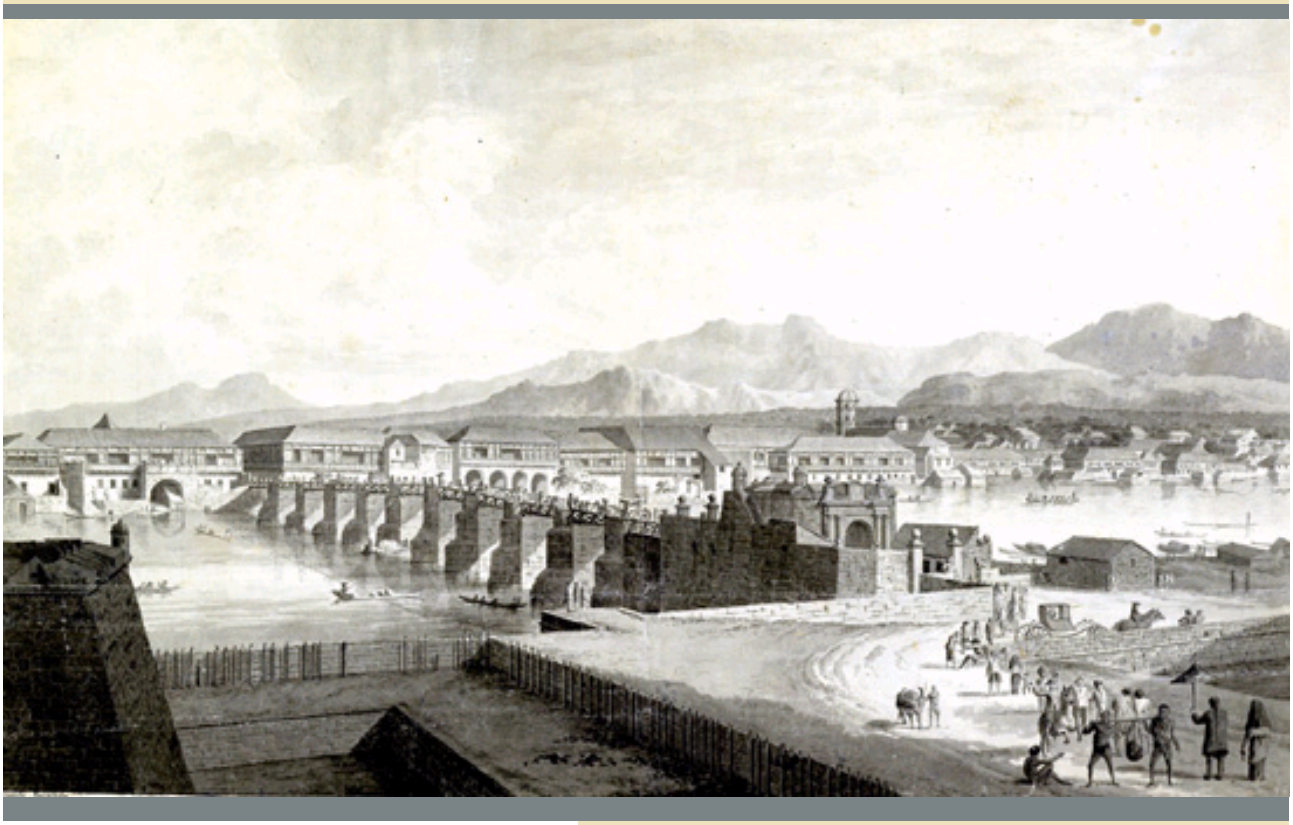
Especificación de interfaz de usuario:

De los requisitos a la generación automática.

Tesis Doctoral



Pedro Juan Molina Moreno

pjmolina@care-t.com

Marzo de 2003

Departamento de Sistemas
Informáticos y Computación

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Especificación de interfaz de usuario: de los requisitos a la generación automática.

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Pedro Juan Molina Moreno
pjmolina@care-t.com

Departamento de Sistemas Informáticos y Computación
UNIVERSIDAD POLITÉCNICA DE VALENCIA

Director: **Dr. Óscar Pastor López**

Valencia, 18 de diciembre de 2002

Diseño de cubierta: Pedro J. Molina
Ilustración de cubierta: Vista del río Pasig (Filipinas) con el Puente Grande de piedra, antes del terremoto de 1863. Fernando Brambila. Colección de dibujos y grabados de la Expedición Malaspina. 1789-1794. Museo Naval, Madrid.

Autoedición y maquetado en LaTeX / MiKTeX y pdfTeX.

Abstract

Developing user interfaces is a high resources consuming task. On the other hand, Conceptual Modeling techniques have been proven to be valuable in order to increase the abstraction level to build systems. However, traditionally, these approaches have forgotten the modeling of the user interface. This PhD thesis provides an extension for the specification of user interfaces on object-oriented conceptual models. The proposed model explores the conceptual pattern languages as tools for requirement elicitation, specification and code generation. As supporting tools, model's editors, specification, prototypes and code generators have been developed. All of them have been refined, fine-tuned and validated in an empiric way in an industrial environment after several iterative development cycles. The user interface development process proposed reduces drastically design times and resources needed due to code generation and rapid prototyping techniques introduced. Analysis based on Conceptual Modeling and supported by code generation techniques narrows the gap to the Automatic Programming Paradigm utopia as stated by Balzer. Moreover, in this way, the developing of software systems change from an artisan process to a software production method where the quality of the process followed can be guaranteed in the final product.

Resumen

El desarrollo de interfaces de usuario es una tarea que consume grandes cantidades de recursos. Por otro lado, las técnicas de modelado conceptual han demostrado ser muy valiosas para incrementar el nivel de abstracción a la hora de construir sistemas, sin embargo, históricamente dejaban de lado el modelado de la interfaz de usuario. La presente tesis proporciona una extensión para la especificación de interfaces de usuario sobre modelos conceptuales orientados a objetos. El modelo propuesto explora los lenguajes de patrones conceptuales como herramientas para la elicitación de requisitos, especificación y generación de código. Como herramientas de soporte, se han construido editores de modelos, especificaciones, prototipos y generadores de código. Todo ello, ha sido refinado y validado empíricamente en un entorno industrial tras varios ciclos de desarrollo iterativo. El proceso de desarrollo de interfaces de usuario propuesto reduce drásticamente los tiempos de desarrollo así como los recursos necesarios gracias a las técnicas de generación de código y prototipación rápida introducidas. El análisis basado en Modelización Conceptual y apoyado sobre técnicas de generación de código acerca un poco más la utopía del Paradigma de la Programación Automática enunciada por Balzer. Más aún, la construcción de sistemas pierde características de artesanía para convertirse en un método de ingeniería de producción de software que asegura la calidad del producto software.

Resum

El desenvolupament d'interfícies d'usuari és una tasca que consumeix grans quantitats de recursos. D'altra banda, les tècniques de modelatge conceptual han demostrat ser molt valuoses per a incrementar el nivell d'abstracció a l'hora de construir sistemes, no obstant, històricament deixaven de costat el modelatge de la interfície d'usuari. La present tesi proporciona una extensió per a l'especificació d'interfícies d'usuari sobre models conceptuals orientats a objectes. El model proposat explora els llenguatges de patrons conceptuals com a ferramentes per a l'elicitació de requisits, especificació i generació de codi. Com a ferramentes de suport, s'han construït editors de models, especificacions, prototips i generadors de codi. Tot això, ha sigut refinat i validat empíricament en un entorn industrial després de diversos cicles de desenvolupament iteratiu. El procés de desenvolupament d'interfícies d'usuari proposat redueix dràsticament els temps de desenvolupament així com els recursos necessaris gràcies a les tècniques de generació de codi i prototipat ràpida introduïdes. L'anàlisi basada en Modelització Conceptual i recolzat sobre tècniques de generació de codi acosta un poc més la utopia del Paradigma de la Programació Automàtica enunciada per Balzer. Més encara, la construcció de sistemes perd característiques d'artesania per a convertir-se en un mètode d'enginyeria de producció de programari que assegura la qualitat del producte programari.

Palabras clave

Keywords:

User interface, specification model, conceptual modeling, code generation, object oriented methods, patterns.

Palabras clave:

Interfaz de usuario, modelo de especificación, modelado conceptual, generación de código, métodos orientación a objetos, patrones.

Paraules clau:

Interfície d'usuari, model d'especificació, modelatge conceptual, generació de codi, mètodes d'orientació a objectes, patrons.

Datos de la tesis

Título de la tesis:	*Especificación de interfaz de usuario: de los requisitos a la generación automática.*
Presentada por:	*Ingeniero* Pedro Juan Molina Moreno, `pjmolina@care-t.com`
Dirigida por:	*Catedrático* Oscar Pastor López, `opastor@dsic.upv.es`
Universidad:	Universidad Politécnica de Valencia.
Departamento:	Sistemas Informáticos y Computación.
Programa de doctorado:	Programación Declarativa e Ingeniería de la Programación.
	Memoria para optar al grado de Doctor en Informática.
Depósito:	Valencia, a 18 de Febrero de 2003.
Defensa:	Valencia, a 14 de Marzo de 2003.

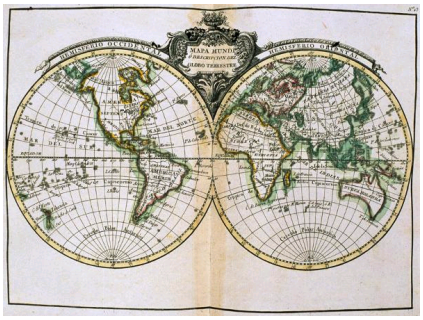
Dedicado

a mi amada Silvia:
por sacrificar tantos fines de semana sin mi compañía,

a mis queridos padres, Pedro y Manoli:
por apoyarme sin descanso en el largo camino para llegar hasta aquí,

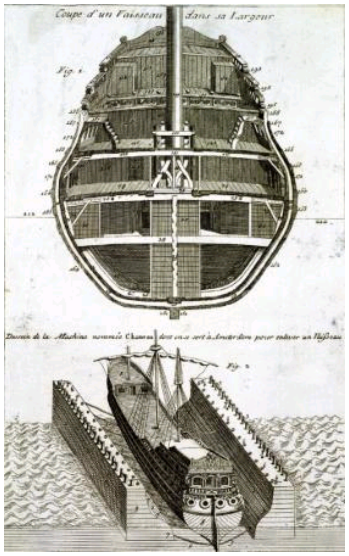
y a mi siempre despierto hermano Pablo:
por la ayuda logística prestada ;-)

Agradecimientos



Enrolarme en el desarrollo de una tesis, como lo sería un viaje en galeón al *Nuevo Mundo* allá por el siglo XVI, ha sido para mi un excitante desafío. Tal aventura conlleva años de preparación y esfuerzo continuado, que sin lugar a dudas, sería difícilmente abordable e imposible de llevar a buen puerto sin la colaboración y el aliento de muchas personas.

A todos aquellos que me han prestado su ayuda de un modo u otro a lo largo de la travesía quiero agradecerles su colaboración y dedicarles un pedacito de está tesis que sin duda merecen.



Gracias en primer lugar al Catedrático Óscar Pastor, capitán experimentado de la *Ruta de las Especias* y mentor de este grumete con el que aprendí a esquivar los embites de corsarios y piratas. Gracias también a los integrantes del Grupo de Investigación de OO-Method y demás tripulación del DSIC donde, con centro de operaciones en el insigne D3, comencé a navegar allá por el año 1998 d.C. Gracias al visionario y mecenas José Iborra por la financiación de aquellas primeras velas con las que abandonamos la tierra firme para echarnos a la mar.

Puedo asegurar sin temor a equivocarme que CARE Technologies S.A. ha sido la nave ideal que nos ha permitido surcar las aguas para alcanzar el ansiado objetivo. Quiero ex-

presar mi agradecimiento a su presidente Siegfried Borho, a su director Emilio Iborra por facilitar los medios para tal empresa, a todos los integrantes de los departamentos de I+D, especialmente a los veteranos marinos con los he combatido mano a mano en mil batallas, y Factoría de Aplicaciones por presentar las condiciones ideales para investigar y probar nuevos materiales para las velas, diseñar y preparar la nave para soportar cualquier contingencia, bien sea tormenta o huracán tropical.

Gracias también a los críticos y sobre todo pacientes colaboradores: Silvia Estébanez, Javier Hernández, José Romero, Ángel Martínez, Pedro J. Jiménez, J. Ignacio Sánchez, Ramón Mollá y Javier Jaen por hacerme rectificar el rumbo cuando me desviaba de la ruta trazada con el sextante respecto a la Osa Mayor.

Mención especial merece el capitán belga Prof. Jean Vanderdonckt que me ha prestado su ayuda y orientación desinteresada durante la navegación en alta mar.

Por último, gracias a los amigos de verdad que han animado a continuar con tesón cuando el azar nos era adverso y las fuerzas podían flaquear.

Hoy hemos avistado un pedazo de tierra. Parece que el viaje llega a su fin: ¿Serán estás las cálidas playas del *Nuevo Mundo*? Pronto lo sabremos.

Gracias a cada uno de vosotros amigos.

Denia, a 18 de diciembre de 2002 d.C.

Pedro J. Molina

Las figuras empleadas pertenecen a la exposición: «*Manila 1571-1898. Occidente en Oriente*» del Centro de Estudios Históricos de Obras Públicas y Urbanismo. Fuente: `http://www.cedex.es/cehopu/expomanila/`.[1][2][3][4]

[1] Mapamundi. Tomás López. 1792. Museo Naval, Madrid.

[2] Corte transversal de un navío y dibujo de un dique seco. Álbum marino de Cesáreo Fernández Duro. Museo Naval.

[3] Astrolabio astronómico del siglo XVI. Museo Naval.

[4] Vista de una torre y parte del pueblo de Samboangan. Fernando Brambila. Colección de dibujos y grabados de la Expedición Malaspina. 1789-1794. Museo Naval.

Índice general

Capítulo 1

Introducción

«Jamás se descubriría nada si nos considerásemos satisfechos con las cosas descubiertas.»

— Lucio A. Séneca, filósofo y pensador cordobés, 4 a.C. – 65.

«En su esencia, la investigación es la búsqueda de la verdad.»

— Eduardo Primo Yúfera, investigador español, [Primo94, 1^{er} capítulo, página 17].

1.1. Motivación

Las aplicaciones basadas en sistemas de información cuidan, cada vez más, sus interfaces de usuario. Atrás quedaron los tiempos donde los ordenadores eran máquinas para una élite: bastaba con que el sistema hiciese bien el trabajo no importando cuan críptica[1] fuese la interfaz de usuario.

Mas allá de su finalidad esencial de interacción entre hombre y máquina, las interfaces de usuario conforman también la cara visible o *escaparate* de las aplicaciones tal y como las percibe el cliente final que las usa.

Como consecuencia directa, las interfaces de usuario entregadas en productos finales están cada vez más y más cuidadas [Constantine99] para ofrecer al usuario sensaciones de seguridad, fiabilidad, ergonomía, sencillez de uso y precisión en la aplicación suministrada. Todas estas cualidades se transmiten subliminalmente a través de la interfaz de usuario.

[1] En los albores de la informática, donde el ENIAC, las válvulas y las tarjetas perforadas eran lo habitual.

Desde el punto de vista del *marketing* podríamos considerar a las aplicaciones informáticas, como cualquier otro producto, «*entran por la vista*». En este sentido, una cuidada presentación es esencial para promover su venta. No es de extrañar que, ante tal panorama, en la producción de software comercial cada vez es más frecuente la inversión de mayores esfuerzos en el desarrollo de interfaces de usuario de alta calidad.

La creciente popularidad de los ordenadores y su uso cotidiano han cambiado la relación de los usuarios frente a la máquina y, por ende, lo que los usuarios esperan de ella. Nuevas formas de ordenadores como asistentes para datos personales (*PDA*), teléfonos móviles, sistemas empotrados o nuevas generaciones de electrodomésticos comienzan a popularizarse, obligándonos a interactuar con sus interfaces que no siempre son todo lo naturales ni intuitivas que uno hubiera deseado.

Los usuarios exigen cualidades al software destinadas a facilitar su trabajo, ahorrar tiempo –de uso y de aprendizaje–, evitar y corregir los errores. Los procesos de desarrollo de aplicaciones dedican cada vez mayores esfuerzos a diseñar y mantener las interfaces de usuario de sus aplicaciones. Gran parte de este trabajo se realiza a mano por diseñadores y programadores que crean y dotan de funcionalidad a las interfaces construidas. Este proceso consume muchos recursos: personal, herramientas, tiempo y dinero.

Por otro lado, la emergente *Sociedad de la Información* de este nuevo milenio incrementa día a día el volumen de datos producidos. Las aplicaciones necesarias para extraer información valiosa del *océano de datos* se convierten en una necesidad imperiosa. Los consumidores de aplicaciones demandan cada vez mejor y más aplicaciones en menos tiempo.

El panorama descrito se beneficiaría de cualquier nuevo avance que permita incrementar la productividad en el desarrollo de interfaces de usuario.

1.2. Planteamiento del problema

La Ingeniería del Software promueve técnicas y métodos de trabajo para lograr que el proceso de producción de Software no sea una tarea artesanal, sino una verdadera tarea de ingeniería. Los lenguajes de modelización conceptual ayudan en este sentido, reduciendo el *gap* semántico (distancia semántica) existente entre el espacio del problema donde reside el usuario y los lenguajes de bajo nivel, pertenecientes al espacio de la solución, entendibles por la máquina.

El *Paradigma de Programación Automática* [Balzer83], sus aproximaciones desde el modelado conceptual (OBLOG [Sernadas87], Troll [Hartmann94], *OASIS* [Pastor92b, Pastor95]) y la generación automática de código (OBLOG

[ESDI93], OO-Method [Pastor96, Pastor97]) logran incrementar la velocidad de desarrollo de aplicaciones de un modo espectacular provocando una revolución comparable al paso del código ensamblador a lenguajes de programación de tercera generación. Sin embargo, los modelos conceptuales suelen centrar la atención sobre el problema en sí: su estructura dinámica asociada, reglas de negocio, es decir, los llamados requisitos funcionales. Las aplicaciones obtenidas de este modo son producidas en tiempos de desarrollo muy bajos y satisfacen las reglas de negocio especificadas, pero carecen de o tienen una muy pobre interfaz de usuario que ha de ser construida a mano por completo o al menos refinada por diseñadores especializados.

Sin embargo, existen otro tipo de requisitos denominados clásicamente como *requisitos no funcionales* que agrupan otro tipo de requisitos importantes del problema tratado, y que son más escurridizos a la hora de capturarlos mediante técnicas de Modelado Conceptual. En particular, entre estos requisitos no funcionales, hallamos los requisitos relativos a la interfaz de usuario, la comunicación entre el hombre y la máquina para llevar a cabo una tarea.

Si tales requisitos de interfaz de usuario logran ser capturados de una manera precisa y se establece una semántica clara para su aplicación, se abre la puerta de nuevo al *Paradigma de Programación Automática* aplicada al ámbito de las interfaces de usuario. Es decir, la producción de modo automático de interfaces de usuario que cumplen con los requisitos expresados.

Esta tesis pretende proporcionar un marco basado en un lenguaje de patrones conceptuales para la especificación y posterior generación de código completa de interfaces de usuario para sistemas de información. Los sistemas de información engloban a las aplicaciones de gestión, sistemas de negocio, logística, control de stocks, distribución, facturación, contabilidad, etc. constituyendo un gran parte del mercado del software comercial. Esos sistemas información suelen estar soportados por interfaces WIMP[2] basadas en formularios que pueden ser generadas de modo automático. Por contra, controles o mecanismos de interacción como gráficos altamente interactivos o gráficos animados caen claramente fuera del alcance de esta tesis.

La especificación abstracta de interfaces de usuario junto con técnicas de generación automática pueden reducir en gran medida el esfuerzo necesario para obtener las interfaces de tales aplicaciones.

A cambio de invertir un poco más de esfuerzo en la fase de análisis (mediante la especificación del correspondiente esquema conceptual y sus características de interfaz de usuario), las fases de codificación, pruebas e integración se reducen en gran medida gracias a técnicas de generación automática de código

[2]Término acuñado para referenciar interfaces para entornos gráficos de usuario IGU, tradicionalmente basadas en ventanas, iconos, menús y punteros (*Window, Icon, Menu, Pointer*).

que permiten derivar las aplicaciones desde las especificaciones.

Los programas derivados de este modo pueden ser empleados como prototipos de la aplicación. Más aún, un prototipo puede convertirse en versión final, lista para ser usado con pocas o ninguna intervención por parte de los diseñadores y programadores.

La presente tesis proporciona una extensión a los modelos conceptuales orientados a objetos clásicos para lograr capturar los anteriormente referidos requisitos de interfaz de usuario. Para tal fin, se explorará una aproximación basada en *patrones* en la que los esfuerzos se concentran en identificar patrones de comportamiento, estructura o funcionalidad (todo ello a nivel de análisis) para capturar su semántica, y al mismo tiempo, dotar desde el análisis de mecanismos para proporcionar homogeneidad y reuso a las interfaces producidas.

El modelo planteado, denominado Just-UI, permite la especificación de requisitos de interfaz de usuario a nivel conceptual basándose en un lenguaje de patrones conceptuales de interfaz de usuario. Se soporta la generación automática a diferentes dispositivos a partir de este modelo. Completado con un método orientado a objetos (p.e. OO-Method), es posible abordar el desarrollo de aplicaciones completas.

El problema planteado cae dentro de lo que se ha denominado *Ingeniería de Dominio* [Cleaveland01], donde el dominio a tratar lo constituyen las Interfaces de Usuario para aplicaciones de gestión.

1.3. Metodología de trabajo

La presente tesis está enmarcada dentro de un trabajo de investigación dilatado en el tiempo que ha compaginado desde su orígenes teoría y practica. Los inicios de este trabajo de investigación se remontan al año 1998, donde en un marco académico como es el grupo de investigación de OO-Method en el Dpto. de Sistemas Informáticos de Computación, se presento un Proyecto Final de Carrera [Molina98] que abrió las puertas a un amplio campo de investigación. Una beca financiada por Consoft S.A. y posteriormente otra beca FPI (Fomento del Personal Investigador) concedida por el Ministerio de Ciencia y Tecnología lo hicieron posible. Posteriormente, la segunda parte ha transcurrido en CARE Technologies S.A., empresa de Investigación y Desarrollo en Ingeniería de Software surgida a raíz del grupo de investigación inicial.

Durante este tiempo, se ha llevado a cabo una investigación activa de experimentación. Se han construido modelos, editores de modelos, prototipos y generadores de código en un ciclo de vida en espiral con refinamientos sucesivos.

A medida que se ha ido desarrollando la base teórica para nuevos conceptos y patrones que enriquecían la expresividad del modelo conceptual, los generadores han sido extendidos de modo incremental para adaptarse a los nuevos requisitos emergentes, implementado los nuevos patrones y rediseñando y reimplementando los patrones previos cuando estos se han visto afectados.

La evaluación de aplicaciones y el estudio teórico ha permitido extraer nuevos patrones útiles. Al poner en práctica los nuevos patrones en el modelo y en los generadores de código se ha podido producir aplicaciones con menos esfuerzo que en la iteración previa.

De este modo, por medio de sucesivas iteraciones se ha conseguido validar los patrones empíricamente sobre casos reales en un contexto industrial para ir destilando en cada iteración el lenguaje de patrones propuesto.

1.4. Resumen de aportaciones

Las principales aportaciones de esta tesis son las siguientes:

1. Describe el estado de la cuestión en la especificación de interfaces basadas en modelos (*véase capítulo 2*).
2. Describe los pilares base en los que se fundamenta esta tesis (*véase capítulo 3*).
3. Explora la aplicación de patrones conceptuales a la especificación de interfaces de usuario. Y se emplean los patrones para construir un nuevo modelo de especificación de interfaces de usuario como medio de captura de propiedades abstractas de las interfaces de usuario (*véase capítulo 4*).
4. Propone un nuevo método para la especificación de interfaces de usuario basado en la extensión de un modelo conceptual orientado a objetos usando patrones conceptuales de interfaz de usuario, manteniendo al mismo tiempo, una única especificación (*véase capítulo 4*).
5. Demuestra como la especificación obtenida es independiente de consideraciones de diseño o implementaciones particulares (*véase capítulo 4*).
6. Se formaliza el modelo presentado usando meta-modelos y especificaciones de tareas (*véase capítulo 5*).
7. Presenta una serie de técnicas de inferencia para facilitar la prototipación rápida de interfaces de usuario a partir de especificaciones incompletas (*véase capítulo 6*).

8. Revisa y propone técnicas para construir generadores de código que deriven automáticamente implementaciones a partir de la especificación para distintas plataformas (*véase capítulo 7*).

9. Propone mecanismos de integración del componente de interfaz de usuario con otros componentes de la aplicación (componentes de lógica de negocio) para disponer de una aplicación totalmente funcional generada de modo totalmente automático (*véase capítulo 7*).

10. A través de la utilización real del método propuesto sobre casos industriales, se demuestra de un modo empírico cómo el método ha incidido positivamente en la productividad, los tiempos de entrega y la calidad de los productos desarrollados (*véase capítulo 8*).

1.5. Estructura de la tesis

Los siguientes apartados describen someramente el contenido de cada capítulo y las convenciones tipográficas empleadas.

1.5.1. Organización

- El presente capítulo 1 introduce la motivación, describe el problema a resolver y declara las convenciones tipográficas empleadas en esta tesis.
- El capítulo 2 describe el estado de la cuestión en los campos industriales y de investigación. Trabajos previos relevantes a la especificación y construcción de interfaces de usuario son descritos en este capítulo.
- Antes de describir con detalle las aportaciones de esta tesis, el capítulo 3 introduce los fundamentos en los que se basa este trabajo. Los conceptos y la terminología descrita se tornan imprescindibles para la correcta comprensión de capítulos posteriores.
- Por fin, el capítulo 4 constituye el capítulo central de la tesis. Describe los patrones conceptuales para describir interfaces de usuario y se describe el modelo de especificación asociado.
- La semántica desde diversos puntos de vista es descrita en el capítulo 5.
- Un mecanismo adicional para favorecer el prototipado rápido es descrito en el capítulo 6.
- Las técnicas de generación automática de código a partir del modelo de especificación son descritas en el capítulo 7.

- La justificación de la utilidad de las ideas presentadas se recoge en el capítulo 8 donde se muestran los resultados empíricos obtenidos tras la aplicación en un entorno industrial con software comercial.
- Finalmente, el capítulo 9 resume las conclusiones y aportaciones del trabajo presentado.
- Los apéndices proporcionan información complementaria: el apéndice A presenta una extensión sintáctica a $\mathcal{OASIS}$ para dar cuenta de los patrones conceptuales descritos en esta tesis, el apéndice B presenta un DTD para describir en XML especificaciones Just-UI, el apéndice C describe en detalle las fórmulas de filtro, el apéndice D hace lo propio para las reglas de dependencia, mientras que el apéndice E muestra un ejemplo de guía de estilo para una plataforma particular. Por último, los apéndices F y G contienen una lista de abreviaturas y un glosario de términos respectivamente.
- La bibliografía junto con los índices de figuras y tablas cierran este trabajo.

1.5.2. Convenciones tipográficas empleadas

Para facilitar en la medida de lo posible la lectura de la tesis se han adoptado una serie de convenciones tipográficas encaminadas a dotar de homogeneidad a los distintos tipos de conceptos empleados. Las convenciones son las habituales en los textos impresos. A continuación, se enumeran cada una de las convenciones junto a un ejemplo.

- Los términos o vocablos foráneos comúnmente aceptados en la jerga informática y que, sin embargo, no son propios del español, aparecen resaltados en cursiva para indicar que se trata de términos procedentes de otra lengua, por ejemplo, el *sort* de interés.
- Las citas o definiciones procedentes de terceras fuentes aparecen en cursiva y convenientemente identadas. Ejemplo de cita:

  > *«Alea iacta est.»*
  >
  > — Gaius Julius Caesar, emperador romano, 100 a.C. – 44 a.C.

- El código fuente proveniente de un lenguaje de especificación formal o un lenguaje de programación aparece en bloques distinguidos con tipografía no proporcional.

```
[User]trabaja.SetValue(a) : a = "No" :
              sueldo.SetActive(false) .
              sueldo.SetValue(0)
```

- Las referencias a funciones dentro de una interfaz de usuario particular se denotará empleando fuente *Sans Serif*. Como por ejemplo:
  - *Combinaciones de teclado:* Presione Ctrl + Alt + Del para ...
  - *Entradas de menú:* Opciones→Preferencias→Fuentes
  - *Botones de acción:* Pulse Aceptar para confirmar la tarea.
- Las referencias o bibliografía aparecen entre corchetes siguiendo el convenio estándar indicando un acrónimo y año de publicación [Alexander64]. La información completa a las referencias se encuentra clasificada alfabéticamente en la sección de Bibliografía.
- Las existencia de notas al pie[3] aparecen etiquetadas con un subíndice numérico (único para cada capítulo) que corresponde con la nota etiquetada con el mismo número que figura al pie de la página.
- Para facilitar la lectura no-lineal y la navegación por el documento, la versión digital (no impresa) de esta tesis contiene hiper-enlaces para las citas bibliográficas, referencias a capítulos o secciones, notas al pie, índice general, índices de figuras y tablas, direcciones de internet (en formato de URL), etc.

  En la sección de Bibliografía aparecen (a modo de referencias cruzadas) las páginas de aparición para cada referencia. Del mismo modo, en la versión digital de este documento dichas referencias cruzadas son hiper-enlaces.

[3] Ejemplo de nota al pie.

Capítulo 2

Situación actual: Construcción de Interfaces de Usuario

«Aquel que duda y no investiga,
se torna no sólo infeliz,
sino también injusto.»

— Blaise Pascal, matemático y pensador francés, 1623 – 1662.

El presente capítulo realiza un recorrido por los trabajos más relevantes en el campo de la especificación y construcción de interfaces de usuario tanto desde el punto de vista de trabajos prácticos implementados en herramientas comerciales y de uso actual en la industria como trabajos teóricos desarrollados en ámbitos académicos. Una vez descrito el cuadro general, será más fácil encuadrar las aportaciones al campo proporcionadas por esta tesis.

2.1. Herramientas para la construcción de Interfaces de Usuario

En el mundo de las aplicaciones comerciales podemos diferencias dos tipos de aproximaciones para la construcción de interfaces de usuario:

1. **RAD y herramientas de autor.** Son herramientas de diseño para construir directamente la interfaz de usuario.
2. **Herramientas basadas en modelos.** Son herramientas basadas en modelos para intentar aumentar el nivel de abstracción. Se apoyan en generadores de código parciales o totales.

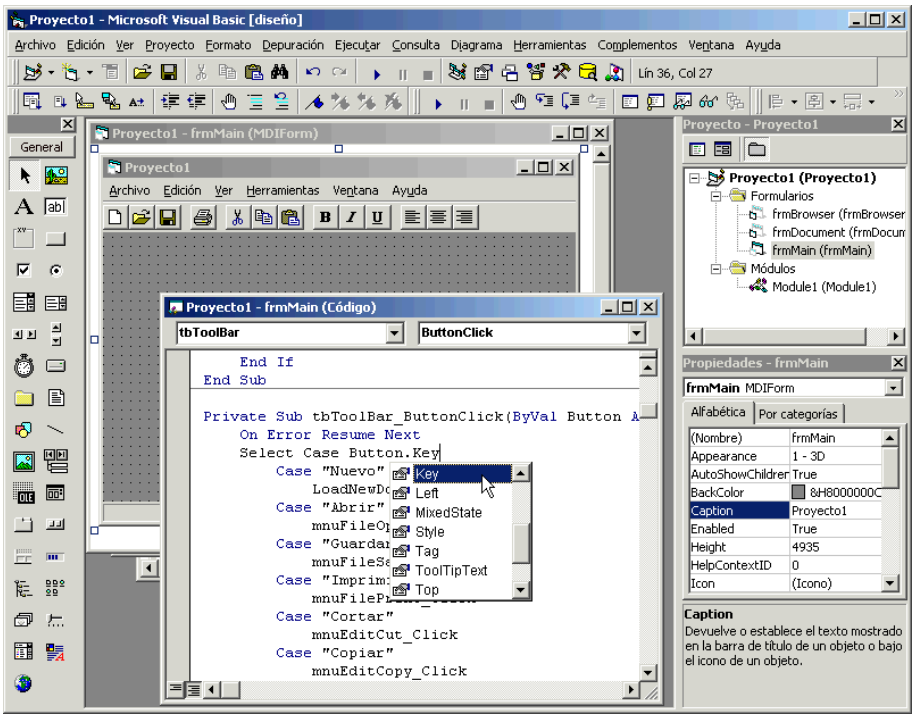


Figura 2.1: Entorno de Desarrollo Integrado de Visual Basic 6.0.

A continuación se describirán las más significativas.

2.1.1. RAD: Entornos de desarrollo rápido

La primera categoría tiene un menor nivel de abstracción. Aquí se encuadran las llamadas herramientas RAD *(Rapid Application Development)*. *Visual Basic (Microsoft Corp.)*, *Power Builder* (Sybase), *Delphi* o *C++ Builder* (Inprise Corp.) son brillantes ejemplos de este tipo de herramientas donde la construcción de interfaces de usuario es muy rápida en comparación con las herramientas y lenguajes de 3ª generación previos. En ellas, el programador y/o diseñador de la interfaz de usuario ayudados por un IDE[1] (como pueden ser *Eclipse* o *Visual Studio .NET*) van construyendo la interfaz de usuario mediante el paradigma WYSIWYG: eligiendo los componentes de la interfaz, ajustando sus propiedades y programando el enlace con la lógica de la aplicación propiamente dicha.

Es precisamente, el paradigma WYSIWYG, también conocido como *programación visual*, el que ha hecho más productiva la fase de producción de inter-

[1]IDE: Entorno integrado de desarrollo. Véase como ejemplo la Figura 2.1

faces de usuario tal y como se venía haciendo antes de la aparición de este tipo de herramientas. Todo ello a pesar de que dichas herramientas están limitadas:

- Dan soporte a los aspectos de presentación, pero no cubren adecuadamente aspectos de diálogo que ha de ser codificado completamente a mano.
- Solo aptas para interfaces WIMP. Por ejemplo, los gráficos dinámicos o gráficos interactivos deben ser programados de modo manual.

Herramientas de autor como Macromedia Director, Macromedia Flash o Macromedia Dreamweaver [Inc.02] son un claro exponente de editores para la construcción de interfaces. Sin embargo, construir buenas interfaces con estas herramientas es complejo. De este modo, la calidad y corrección de la interfaz así construida queda en manos del «saber hacer» del diseñador que la construye. Para garantizar que dicha interfaz sigue estándares de calidad y se ajusta a los requisitos se necesitarán realizar, como mínimo, diversas tandas de pruebas y evaluaciones.

2.1.2. Herramientas basadas en modelos

Para completar este recorrido sobre el estado actual de las herramientas, comentaremos aquellas herramientas comerciales que están siendo usadas en la industria para la producción de software. En particular, se comentarán Rational Rose, TogetherSoft, Argo UML, System Architect, Cool:Plex, ezWay y Genova.

Rational Rose

Rational Rose 2000 [Rat02] (Rational Corp.) es la herramienta CASE de referencia para la especificación gráfica con la notación UML. Las herramientas de Rational Corp. intentan abarcar todo el ciclo de vida para dar soporte a su metodología RUP *(Ration Unified Process)*. Sin embargo, Rational no dispone de herramientas para el soporte directo al desarrollo de interfaces de usuario. Ni en la notación UML, ni en las herramientas de soporte a RUP, se da una respuesta al modelado o diseño de interfaces de usuario.

Together

Together [Tog02] es otra herramienta de modelado de código (modelado y documentación de diseño orientado a lenguajes de 3^a generación) de la compañía TogetherSoft. Similar a Rational Rose, da soporte al modelado de código,

generación de clases y permite mantener sincronizado código y modelo de diseño. En la última versión, incluye características tipo RAD (*UI Builder*) con el cual es posible diseñar interfaces de usuario para el lenguaje Java.

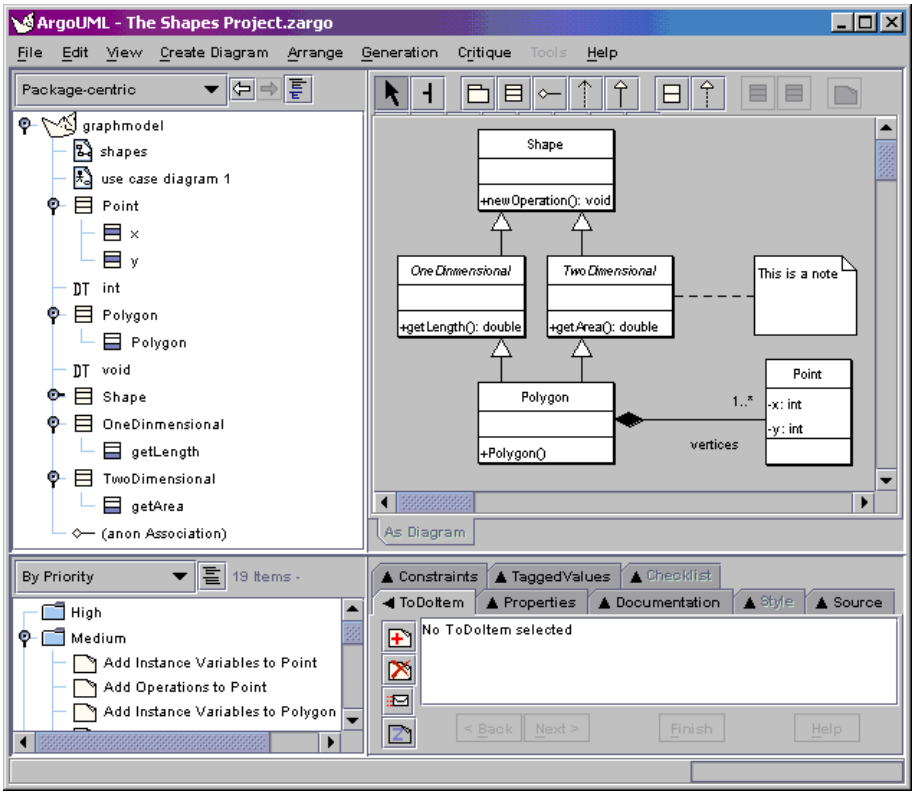


Figura 2.2: Herramienta Argo UML.

Argo UML

Argo UML [Arg02, Robbins97] es la respuesta del mundo del software de código abierto a herramientas como Rational Rose. Implementado en Java y con licencia de código abierto, permite modelar usando notación UML (*véase Figura 2.2*). La apertura de código facilita la creación de módulos adicionales para la generación de código.[2] Al igual que Rational Rose, no dispone de aspectos específicos para el desarrollo de interfaces de usuario.

[2] Argo UML es un proyecto GNU de código abierto en Java. Disponible en `http://argouml.tigris.org`

System Architect

Por otro lado, System Architect [Sys01] (Popkin Software) es otra herramienta CASE que permite construir diagramas según diversas notaciones tales como UML, OMT o Coad & Yourdon entre otras. Su finalidad es la de soportar la fase de análisis de un sistema software y la de proporcionar un marco de trabajo inicial en la fase de diseño. Para este último fin, esta herramienta proporciona un lenguaje de guiones o de *scripts*. Sobre dicho lenguaje se construyen programas que recorren la información obtenida en el análisis y generan plantillas de código que contienen las cabeceras de las funciones a implementar. Lejos de ser *scripts* de generación completos, los *scripts* generan muy poco código y sólo los aspectos más genéricos estructurales. El usuario puede adaptar el *script* para ajustarlo a sus necesidades, pero para ello, deberá aprender a manejar este nuevo lenguaje de *script* y conocer la estructura de la información del modelo que se pretende recorrer.

System Architect cuenta con *scripts* para generar código SQL que produce las tablas e índices para diversos gestores de base de datos, código de cabecera para funciones y clases en C++, Java, etc. Sin embargo, la generación de código en estos entornos para interfaz de usuario es casi inexistente.

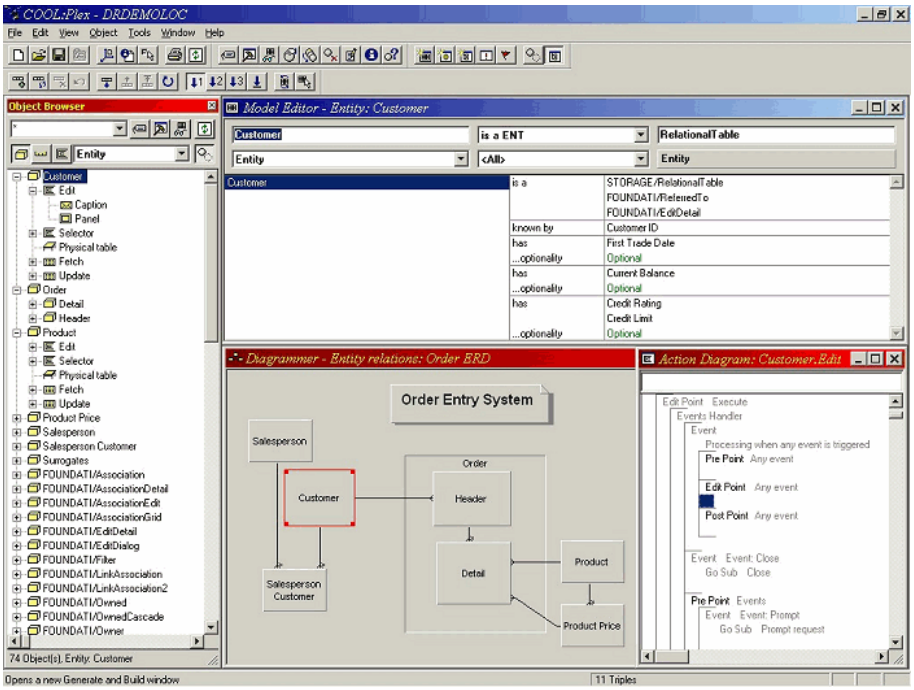


Figura 2.3: Herramienta Cool:Plex.

Cool:Plex

Cool:Plex [Coo01] es un producto de Computer Associates. Es una herramienta comercial destinada al diseño de aplicaciones cliente-servidor independiente de plataforma. Se basa en diagramas de entidad relación extendidos (*véase Figura 2.3*) para especificar la estática del sistema. Usa componentes y patrones de diseño [Gamma95] que son traducidos en la fase de generación a un lenguaje destino en particular. La interfaz de usuario se diseña al estilo WYSIWYG de modo similar a las herramientas RAD. De la interfaz, lo único que se permite diseñar es la estática de cada ventana o panel.

eZway

eZway [eZw01] es un producto de Freeway Inc. Es una herramienta de desarrollo de aplicaciones ligada a plataformas y tecnologías de Microsoft. El proceso de construcción está soportado por las siguientes características:

- Definición de funcionalidad.
- Diagrama de casos de uso. Dibujados usando la herramienta VISIO [VIS02].
- Especificación de los casos de uso, a través de una tabla textual.
- Definición de formularios. Diseño usando formularios para Visual Basic o páginas ASP.
- Componentes de segunda y tercera capa. Permite definir los componentes de segunda y tercera capa para lógica de negocio y persistencia respectivamente.
- Generación de código. A partir de los objetos definidos en cada capa genera código Visual Basic o ASP.
- Gestión del cambio. Se permite almacenar código personalizado Visual Basic para ensamblar con la aplicación final.

eZway, por tanto, está muy ligado a la plataforma Microsoft. Emplea Visual Basic y objetos ActiveX en el proceso de diseño, lo que dificulta la migración del proyecto a otras plataformas como pueda ser Java.

Desde el punto de vista de la interfaz de usuario, el diseño es dependiente de la tecnología Microsoft, por lo cual, no puede ser fácilmente migrado a otros entornos no Windows.

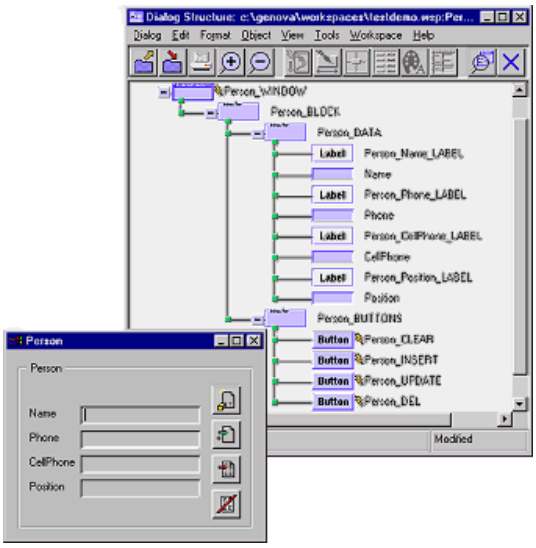


Figura 2.4: Modelo de Diálogo de Genova.

Genova

Genova 7.0 es un módulo de extensión para Rational Rose desarrollado por Genera A.S. [Genera00]. A partir de modelos UML construidos con Rational Rose, Genova permite construir Modelos de Diálogo y de Presentación (*véase Figura 2.4*) de modo parcial que representan la interfaz de usuario como un árbol de composición de AIO[3] [Bodart96]. Genova incluye el concepto de «Selectores de objeto» (*Object Selectors*) para establecer las correspondencias entre AIO → CIO. Algunos aspectos de diseño pueden ser parametrizados (*véase Figuras 2.5 y 2.6*)lo que permite configurar una *guía de estilo* con tal parametrización.

El desarrollador selecciona las clases que desea emplear desde un modelo de clases en Rational Rose y selecciona qué guía de estilo emplear. La guía de estilo asegura la uniformidad de la presentación en la interfaz de usuario generada.

La generación de código de Genova soporta las siguientes plataformas destino: Java, C++, Visual Basic and HTML.

[3]AIO. (*Abstract Interaction Object*) Objeto abstracto de interfaz [Vanderdonckt93] frente a CIO (*Concrete Interaction Object*) Objeto concreto de interacción, también llamado control o *widget*. Ambos se discutirán más adelante con mayor precisión.

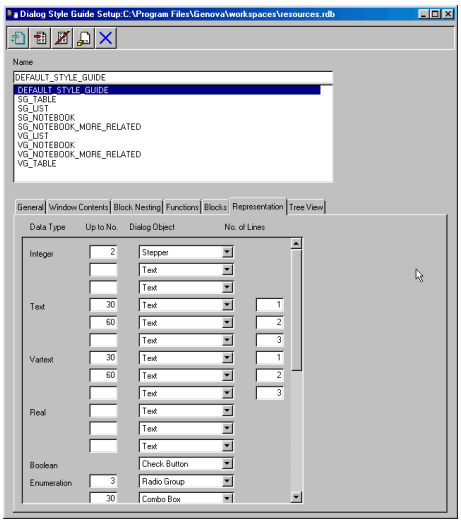


Figura 2.5: Parámetros de diseño en Genova 1/2.

2.1.3. Conclusiones sobre las herramientas comerciales

Las herramientas de tipo RAD para el desarrollo de interfaces de usuario suelen aparecer integradas dentro de IDE (Delphi, Visual Basic, Eclipse, Visual Studio .Net, etc.) de lenguajes de tercera generación. En estas herramientas de programación, el desarrollo de la interfaz de usuario se realiza a un nivel de diseño. La experiencia del diseñador es crucial para obtener interfaces de alta calidad. Las interfaces diseñadas de este modo son dependientes de un lenguaje de programación o librería de controles dada, dificultando la portabilidad a otros entornos.

Por otro lado, las principales herramientas de modelado que incrementan el nivel de abstracción como son Rational Rose, Together, Argo UML, System Architect basadas en UML no proporcionan mecanismos para la especificación de interfaces. Cabe mencionar que, el modelado realizado con estas herramientas, es más cercano al modelado de código que al modelado conceptual ya que muchas introducen dependencias del lenguaje destino en la especificación, imposibilitando de este modo la generación a otras plataformas.

Un segundo tipo de herramientas híbridas como Cool:Plex y ezWay introducen diseñadores de interfaces de usuario. Si bien, el primero sólo permite especificar la estática de la interfaz de usuario. Y el segundo es dependiente de los lenguajes destino Visual Basic y ASP.

Por último, Genova introduce un Modelo de Diálogo y de Presentación siguiendo las ideas de Vanderdonckt [Vanderdonckt93]. Probablemente, Genova

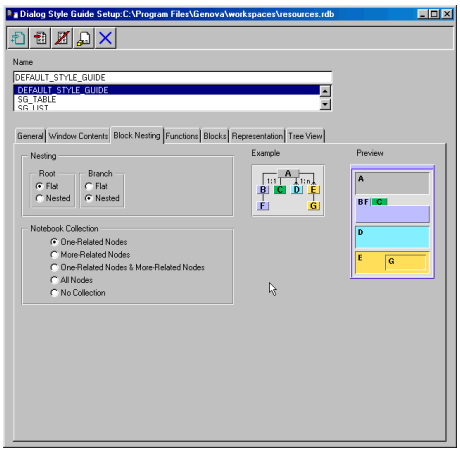


Figura 2.6: Parámetros de diseño en Genova 2/2.

es la herramienta comercial más avanzada para la generación de Interfaz de Usuario a partir de modelos UML.[4]

Genova proporciona un control razonable sobre los parámetros de generación. Sin embargo, al ser un módulo de extensión a Rational Rose, los modelos de diálogo se almacenan de modo separado al modelo UML raíz. Esto puede provocar problemas de sincronización y de desfase de versiones cuando el modelo UML cambia y los Modelos de Diálogo asociados no lo hacen. La generación producida por Genova está basada en pantallas y está orientada a la invocación de uno o más métodos.

A la vista de la revisión a la tecnología, podemos concluir que no existen herramientas que complementen el modelado conceptual y el modelado de interfaz de usuario a nivel conceptual (independiente de plataforma destino) de una manera completamente integrada.

Disponer de un único modelo capaz de contener la descripción de estructura, comportamiento, funcionalidad e interfaz de usuario se convierte en un objetivo de primer orden perseguido por la presente tesis.

2.2. Modelos para la especificación de Interfaces de Usuario

Las herramientas del ámbito académico ponen el énfasis en la especificación de la interfaz de usuario. Dejando de lado los detalles de imple-

[4]Rational Rose por sí solo, no proporciona ningún soporte para la Interfaz de Usuario.

mentación, se centran en la captura declarativa del comportamiento esperado del sistema. Una vez completado el análisis, algunas de las herramientas plantean la posibilidad de generar código para prototipos en uno o más lenguajes destino.

Algunos de los trabajos presentados pueden ser denominados como MB-UIDE,[5] como evolución de los UIMS.[6]

En esta sección se describirán los trabajos más relevantes y se comparará la información que cada uno de ellos recopila. Dicha información es vital para la fase de generación de interfaces de usuario, ya que la idoneidad, completitud y no-ambigüedad de la información condicionan fuertemente qué puede generarse automáticamente y qué no.

A continuación, se describirán los trabajos relacionados en el campo de especificación de interfaz de usuario. En particular, nos detendremos en describir UIDE, JANUS, TRIDENT, OVID, UMLi, CTT, MOBI-D y Wisdom.

2.2.1. UIDE

UIDE es un entorno que permite al analista indicar los requisitos de interfaz de un sistema y diseñar la interfaz de usuario a un nivel de abstracción elevado. UIDE dispone de herramientas adicionales para evaluar la calidad del diseño, generar prototipos de interfaz, etc.

UIDE (User Interface Design Environment) [Foley91] es un sistema soportado por una base de conocimiento para asistir en el proceso de diseño, evaluación e implementación de la interfaz de usuario. UIDE recoge una representación conceptual del diseño de la interfaz de usuario y proporciona un conjunto de herramientas para diseñar, analizar e implementar interfaces de usuario al mismo nivel que las herramientas CASE soportan tareas de ingeniería del software. Esta aproximación encuadra a UIDE dentro del campo general de los Sistemas Gestores de Interfaz de Usuario UIMS *User Interface Management Systems*.

Un UIMS típico permite al diseñador:

- Especificar la organización de las ventanas, iconos y menús.
- Crear la ayuda y mensajes de error.
- Seleccionar técnicas de interacción alternativas, tales como lenguaje de comandos, menús estáticos o menús emergentes *(pop-up menus)*.

[5] *Model based-User Interface Development Environment* Entornos de desarrollo de interfaces de usuario basados en modelos.

[6] *User Interface Management Systems* Sistemas de gestión de interfaces de usuario.

- Especificar la secuencia de acciones del usuario.
- Especificar los nombres de los procedimientos a invocar por el UIMS cuando un comando y sus parámetros han sido introducidos por el usuario.

Los UIMS clásicos tienen un nivel de abstracción muy bajo, tal y como ocurre en la programación visual, donde se manipulan directamente los elementos de implementación (controles) de la interfaz final. UIDE intenta incrementar el nivel de abstracción de un modo notable.

La base de conocimiento de UIDE contiene la siguiente información:

- Jerarquía de clases de objetos que existen en el sistema.
- Propiedades de los objetos (meta-data).
- Acciones que pueden realizarse sobre los objetos.
- Unidades de información requeridas por las acciones (argumentos).
- Pre- y post-condiciones sobre las acciones.

La información recogida en la base de conocimiento (*véase la Figura 2.7*) puede emplearse para:

- Comprobar la consistencia y completitud del diseño.
- Transformar la base de conocimiento e incluso la interfaz de usuario que representa en otra distinta pero funcionalmente equivalente a través de un conjunto de algoritmos de transformación.
- Evaluar la *bondad* (calidad) del diseño.
- Diseñar la disposición de menús y cajas de diálogo.
- Elegir técnicas de interacción apropiadas.
- Generación automática de mecanismos de ayuda inteligente con animación.
- Mantener un modelo del conocimiento del usuario acerca de la aplicación, para facilitar el trabajo a la ayuda inteligente.
- Implementar macros y capacidades deshacer/rehacer *(undo/redo)*.
- Proporcionar entrada a SUIMS, un simple UIMS que implementa la interfaz de usuario a modo de prototipo.

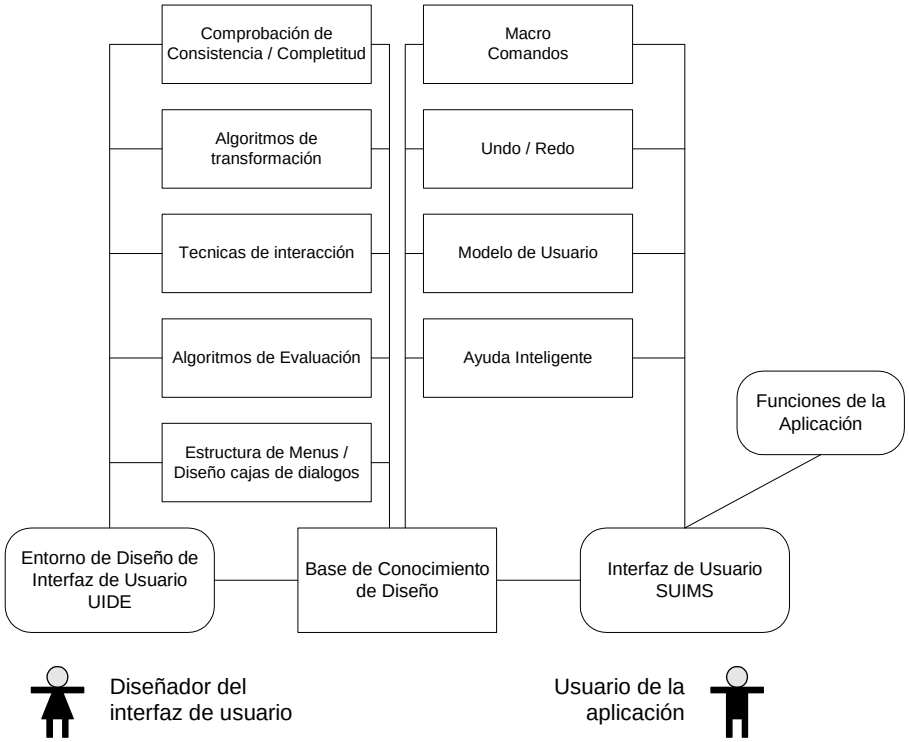


Figura 2.7: Arquitectura de UIDE (adaptado de [Foley91]).

El módulo que interacciona con el usuario, SUIMS sigue el siguiente ciclo de ejecución:

- Establece y actualiza el contenido de la pantalla.
- Comprueba todas la precondiciones y reconoce las acciones que están disponibles.
- Acepta una acción seleccionada por el usuario.
- Procesa cada parámetro de acuerdo a su tipo (explícito, implícito, etc.).
- Acepta los valores de los parámetros en un orden arbitrario, usando cualesquiera de las técnicas de interacción disponibles.
- Confirma implícitamente la acción (si se suministra toda la información necesaria), confirma explícitamente (si así se requiere) o cancela la acción.
- Ejecuta la acción.
- Evalúa las post-condiciones.

SUIMS identifica siete tareas de interacción:

- Selección de comando.
- Selección de clase.
- Selección de instancia.
- Selección de atributo.
- Selección de posición.
- Entrada de texto.
- Entrada entera (argumento de tipo entero).

Para llevarlas a cabo, emplea las llamadas técnicas de interacción. Por ejemplo, las técnicas de interacción *apuntar*, *introducción de texto* y *selección de menú* pueden usarse para llevar a cabo la tarea de interacción *selección de instancia*. No hay restricciones para indicar qué técnicas de interacción pueden ser empleadas en cada tarea de interacción. Si no se selecciona ninguna, es obvio que la tarea no podrá llevarse a cabo. Si están activadas más de una, el usuario puede elegir la que le resulte más cómoda.

UIDE se apoya en un modelo de datos (base de conocimiento) para abordar desde un punto de vista formal la especificación de interfaz. Mediante una aproximación cercana a los principios del modelo orientado a objetos, se recoge

la estructura y la semántica del sistema a modelar con un modelo de datos, lo que permite disponer de una base de conocimiento o información centralizada del sistema a partir de la cual se pueden extraer conclusiones así como proponer implementaciones a la interfaz de usuario.

UIDE constituye una de las primeras herramientas MB-UIDE[7] de primera generación que ha servido como referencia a muchas otras posteriores. UIDE solo soporta el modelo de dominio, careciendo de modelos de tareas, presentación, dialogo o de usuarios.

2.2.2. JANUS

JANUS [Balzert96] es un trabajo de la Universidad de Ruhr en Alemania. Partiendo del Análisis Orientado a Objetos (OOA) [Coad91], genera parte de la interfaz de usuario asociada. El modelo de configuración de clases empleado recoge clases (con métodos y atributos) y relaciones (agregación y herencia). A partir de esta información que debe haber sido recopilada con la ayuda de una herramienta CASE, JANUS es capaz de generar, por ejemplo:

- Un formulario por cada clase para representar una instancia de objeto, con controles para cada atributo y botones par cada método de la clase.
- Un formulario por cada clase para mostrar conjuntos de objetos de una misma clase con una rejilla que contiene tantas columnas como atributos tiene la clase y botones para lanzar los métodos de la clase.

El método OOA proporciona un análisis del sistema pero no captura requisitos de interfaz de usuario propiamente dicho, se apoya solo en el modelo de datos. Esta carencia de información acerca de los requisitos de la interfaz de usuario provocan que la interfaz generada por JANUS no sea muy refinada y sólo contemple la parte de diálogo (parcialmente) de la interfaz.

2.2.3. TRIDENT

TRIDENT[8] [Bodart93, Bodart95d, Bodart95a, Bodart95b, Bodart96] es un proyecto desarrollado en la Universidad de Namur, que propone un modelo de arquitectura para el desarrollo de aplicaciones de gestión interactivas.

El principal objetivo de TRIDENT es permitir al diseñador de interfaz generar, de modo automático, tanto como sea posible, a partir de una especificación

[7] *Model-Based User Interface Design Environment*. Entornos de diseño de interfaces de usuario basados en modelos.

[8] TRIDENT: *(Tools foR an Interactive Development EnvironmeNT)*

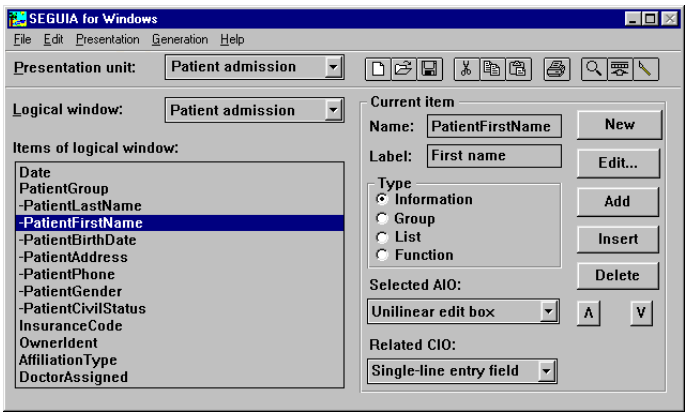


Figura 2.8: Interfaz de la herramienta SEGUIA.

descrita a través de diversos modelos (tales como: modelos de tareas o modelos de entidad-relación orientados a objetos). Aspectos como la selección de los elementos de presentación, elección de objetos de interacción apropiados, distribución en pantalla de los objetos de interacción, secuenciación de diálogos y conversación básica quedan resueltos con esta aproximación.

TRIDENT emplea análisis de requisitos funcionales y análisis de tareas. A partir del primero se obtiene un modelo entidad-relación extendido. El análisis de tareas se realiza construyendo ACG (Grafos encadenados de actividad, *Activity Chain Graphs*) que ligan las tareas interactivas del usuario con la funcionalidad del sistema.

Una de las principales aportaciones de TRIDENT al campo de la especificación de interfaces de usuario es la introducción de del concepto de AIO *(Abstract Interaction Object)* como abstracción de CIO *(Concrete Interaction Object)* o control.

SEGUIA (*System Export for Generating a User Interface Automatically*) [Vanderdonckt99a, Vanderdonckt99b] es un generador de interfaces a partir de modelos construidos por TRIDENT.[9] SEGUIA contiene un sistema experto que selecciona mediante reglas y heurísticos la mejor traducción de AIO a CIO en el proceso de generación además de soportar la disposición y el alineado de los CIO en ventanas y cajas de diálogo (*véase Figura 2.8*).

Grafos encadenados de actividad

Uno de los resultados derivados del análisis de tareas es un modelo que describe las tareas interactivas que el usuario tiene que realizar. Este modelo

[9] Véase `http://www.inyo.ucl.ac.be/bchi/research/seguia`.

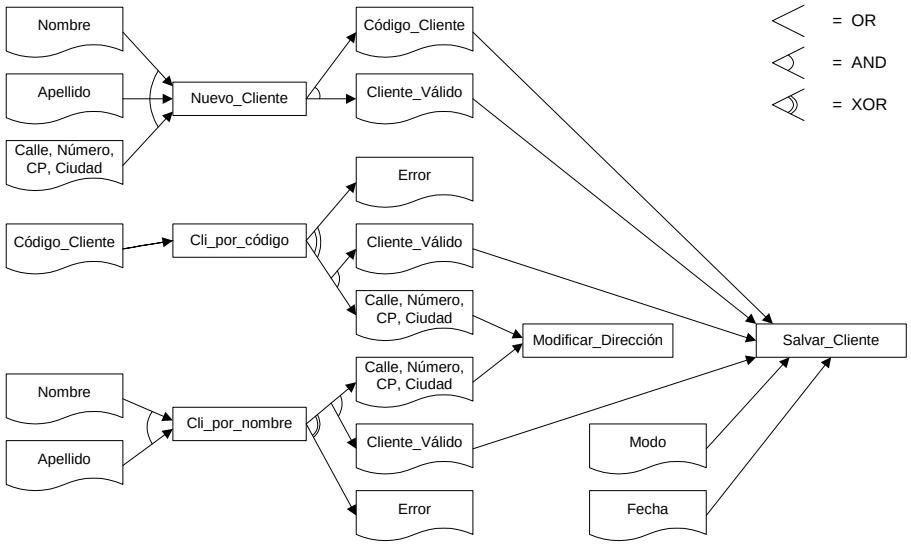


Figura 2.9: Ejemplo de diagrama ACG (adaptado de [Bodart95d]).

puede representarse gráficamente mediante un grafo encadenado de actividad o (ACG). Un ACG despieza un flujo de información en funciones encadenadas para definir una lógica de negocio para alcanzar los objetivos asignados a la tarea interactiva.

Un ejemplo de ACG para una tarea interactiva[10] Salvar_Cliente se muestra en la Figura 2.9. Cada función recibe información de entrada que representa la información necesaria para la correcta ejecución de la función. Por ejemplo, la información Nombre, Apellido, Dirección debe proporcionarse para permitir la ejecución de Nuevo_Cliente. Cada función produce información de salida que puede ir destinada al usuario (por ejemplo: Código_Cliente) o, por el contrario, ser interna e ir destinada como información de entrada para otra función (por ejemplo: Cliente_Válido). La terminación de la ejecución de una función permite que el encadenamiento de funciones progrese un paso más en el AGC. La entrada de información, la visualización de resultados y el encadenamiento de funciones es compatible con enlaces OR (representados sin arco), AND (arcos simples) y XOR (dobles arcos). Por ejemplo, el enlace AND entre Nombre, Apellido y Dirección indica que toda la información es necesaria; el enlace XOR entre Error y el conjunto de informaciones Cliente_Valido y Dirección significa que uno de los dos, el mensaje de error o la combinación de las informaciones será generado.

[10]El ejemplo descrito de ACG es una traducción libre del ejemplo original descrito en [Bodart95d].

Diseño de la presentación

La construcción de los componentes de la interfaz de usuario se basa en los siguientes conceptos:

- **CIO** *(Concrete Interaction Object)*: Es un objeto real, un control perteneciente al mundo de la interfaz de usuario que cualquier usuario puede manipular, como un botón de comando, una lista, una casilla de verificación. Un CIO es compuesto si se puede descomponer en unidades más pequeñas y es simple si no puede ser descompuesto.
- **AIO** *(Abstract Interaction Object)*: Son la abstracción de todos los CIO desde el punto de vista de la presentación y de comportamiento, independientemente del entorno final de implementación.
- **Ventana** *(Window)*: Es la ventana raíz que puede considerarse como una ventana lógica para los AIO y como ventana física, caja de diálogo o panel para CIO. Cada ventana es por sí misma un CIO compuesto de otros CIO simples o compuestos a su vez. Todas las ventanas están geográficamente delimitadas en la ventana del usuario.
- **PU** *(Presentation Unit)*: Constituyen la parte encargada de llevar a cabo la entrada y visualización de los datos de cualquier subtarea de una tarea interactiva. Cada unidad de presentación puede descomponerse en una o más ventanas que pueden no aparecer en pantalla simultáneamente. Cada PU está compuesta por, al menos, una ventana llamada ventana base, a partir de la cual el resto están encadenadas.

Siguiendo el ACG descrito en la Figura 2.9, el usuario puede añadir nuevos clientes o modificar clientes previamente existentes. Para localizar clientes existentes, éstos pueden ser localizados en una base de datos a partir de un código de identificación o a partir del nombre y el apellido. Por lo tanto, dos unidades de presentación deben ser identificadas: una PU1 que cubra las funciones de Nuevo_Cliente, Cliente_por_Código, Cliente_por_Nombre y Modificar_Dirección y otro PU2 que cubra la función Salvar_Cliente.

De acuerdo al ACG, PU1 puede ser dividido en siete ventanas:

1. Una caja de diálogo W11 que permita al usuario seleccionar de entre las tres alternativas (nuevo cliente o cliente existente, identificado en este último caso por nombre o por código). Véase Figura 2.10.
2. Una ventana W12 que permita al usuario añadir un nuevo cliente proporcionando toda la información necesaria. Véase Figura 2.11.

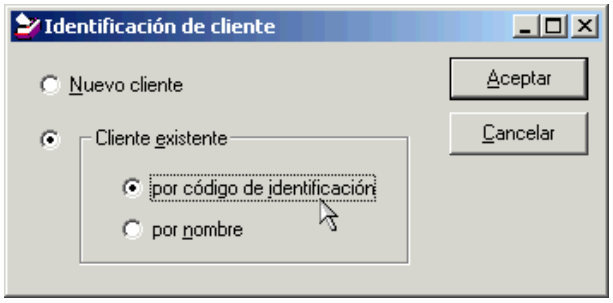


Figura 2.10: Presentación de la ventana W11 (adaptado de [Bodart95d]).

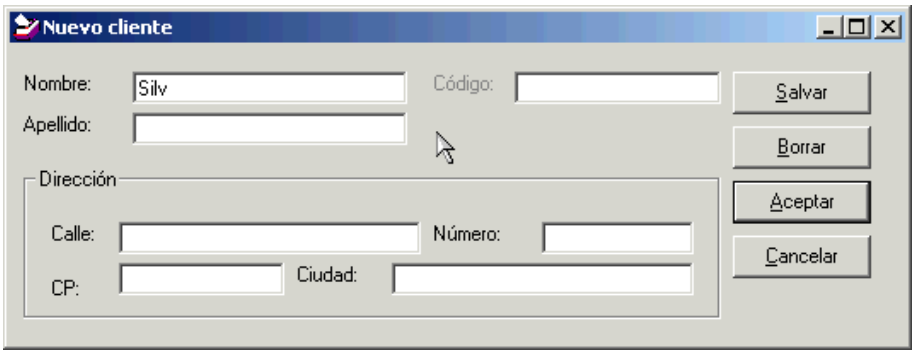


Figura 2.11: Presentación de la ventana W12 (adaptado de [Bodart95d]).

Figura 2.12: Presentación de la ventana W13 (adaptado de [Bodart95d]).

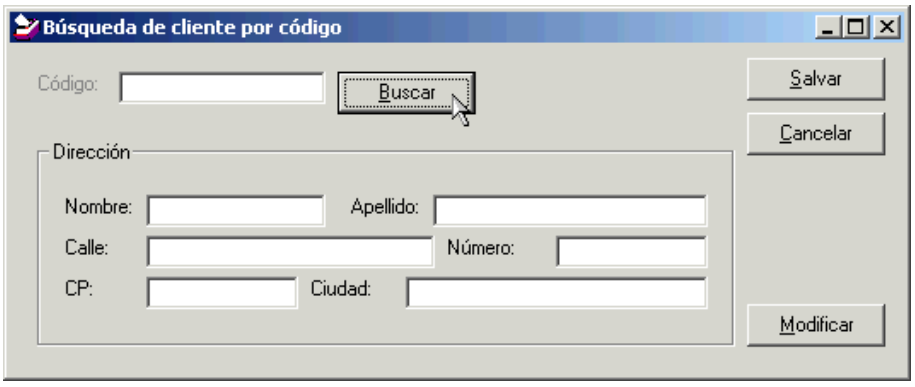


Figura 2.13: Presentación de la ventana W14 (adaptado de [Bodart95d]).

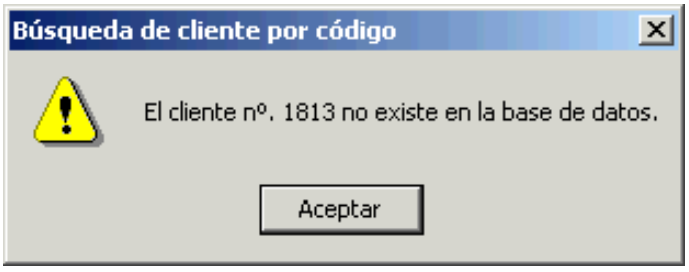


Figura 2.14: Presentación de la ventana W15 (adaptado de [Bodart95d]).

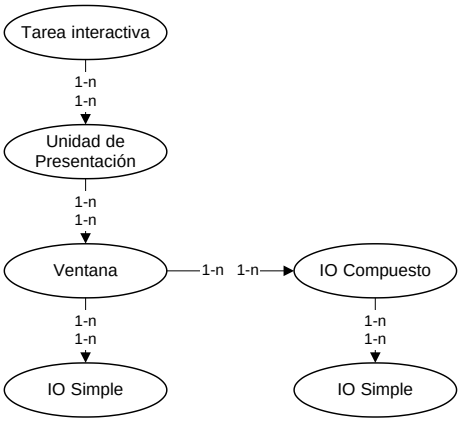


Figura 2.15: Estructura de la presentación en TRIDENT (adaptado de [Bodart95d]).

3. Un mensaje de error W13 para alertar al usuario siempre que falte información en la ventana W12. Véase Figura 2.12.

4. Una ventana W14 que permita al usuario buscar un cliente por su código y modificarlo cuando sea necesario. Véase Figura 2.13.

5. Un mensaje W15 de alerta cuando el código suministrado no corresponda a un cliente existente en la base de datos. Véase Figura 2.14.

6. Una ventana W16 para realizar búsquedas de clientes por el nombre y el apellido, además de permitir su modificación cuando sea necesario (similar a W14).

7. Un mensaje W17 que alerte al usuario cuando se proporcione un nombre y apellido que no corresponda a un cliente existente (similar a W15).

Del mismo modo al indicado, la unidad de presentación PU2 puede componerse de una ventana W21 para permitir la introducción de Modo y Fecha antes de recoger finalmente el Cliente (*véase Figura 2.9*). Esta aproximación define una presentación por refinamientos iterativos empezando desde los objetos generales hasta acabar en objetos terminales de acuerdo a la estructura mostrada en la Figura 2.15.

Dicha estructura de presentación consiste en una jerarquía de CIO para cada tarea interactiva (IT). La Tabla 2.1 presenta a modo de resumen la estructura de la presentación del ejemplo descrito con el nombre del CIO empleado entre paréntesis.

```
IT_Salvar_Cliente
    PU1_Identificar_un_Cliente
        W11_Identificación_de_Cliente (caja de diálogo)
            Tipo_de_Cliente (botón de radio)
            Busqueda_de_Cliente_Existente (botón de radio)
            Aceptar (botón de comando)
            Cancelar (botón de comando)
        W12_Nuevo_Cliente (ventana)
            Nombre (caja de edición)
            Apellido (caja de edición)
            Dirección (grupo)
            Calle (caja de edición)
                ...
            Salvar, Borrar, Aceptar,
            Cancelar (botones de comando)
        W13_Campo_Faltante (mensaje de error)
        W14_Búsqueda_por_Código (ventana)
            ...
        W15_Código_Inexistente (mensaje de alerta)
        W16_Búsqueda_por_nombre (ventana)
            ...
        W17_Nombre_Inexistente (mensaje de alerta)
    PU2_Almacenar_Cliente
        W21_Entrada_Final (caja de diálogo)
            Modo (caja de edición)
            Fecha (caja de edición)
```

Tabla 2.1: Representación textual de la estructura de la presentación en TRIDENT (adaptado de [Bodart95d]).

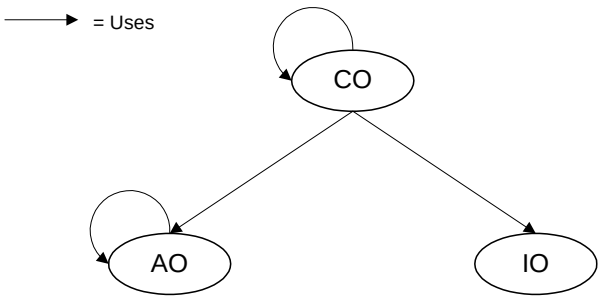


Figura 2.16: Esquema genérico de la arquitectura de TRIDENT (adaptado de [Bodart95d]).

Modelo de la arquitectura

El modelo de arquitectura propuesto para TRIDENT [Bodart95d] consiste en una jerarquía de los elementos genéricos ilustrados en la Figura 2.16.

El modelo de arquitectura se compone de tres clases de objetos cuya descripción es la siguiente:

1. La clase *Control Objects* (CO) es una clase genérica que se descompone en diversos CO de diferentes tipos para manejar el diálogo y asegurar una correspondencia entre las estructuras de datos de la aplicación y las pertenecientes a la presentación. Cada CO es responsable de la gestión de una porción del diálogo y de realizar la correspondencia en esa porción entre la aplicación y la presentación. El comportamiento del CO toma la forma de un *script* escrito en un lenguaje de reglas y puede ser parcialmente representado gráficamente con un diagrama de transición de estados.

2. La clase *Application Objects* (AO) es una clase que no puede ser descompuesta puesto que sus elementos representan las funciones de la aplicación. Es importante asegurar que las funciones necesarias estén incluidas en la arquitectura, ya que los requerimientos funcionales desempeñan un papel importante en la asignación de significado semántico a las tareas interactivas y a sus respectivas subtareas.

3. La clase *Interaction Objects* (IO) es una clase genérica que contiene dos tipos de CIO: CIO encargados de realizar la entrada/salida de información y CIO que están estrictamente implicados en el diálogo (por ejemplo, los botones que disparan las funciones).

La arquitectura de TRIDENT se descompone en tres conceptos (como en MVC[11] [Goldberg83, Bergin02] y PAC[12] [Coutaz87].) La jerarquía de descomposición en TRIDENT es jerárquica a tres niveles, la descomposición de los conceptos esta limitada nivel por nivel. Por en contrario, en PAC, todos los aspectos pueden ser descompuestos jerárquicamente de modo simultaneo.

Las reglas que gobiernan el comportamiento de estas tres clases de objetos son idénticas, del mismo modo que las relaciones permitidas entre cualquier par de objetos en esta jerarquía. Cada objeto es un agente. Si una flecha une un objeto padre con un objeto hijo cualesquiera, se establece una relación entre ellos indicando que el objeto padre usa al objeto hijo. Con más detalle:

- Un objeto hijo envía al objeto padre eventos relacionados con el comportamiento de los estados significativos.
- Un objeto padre invoca las primitivas ofertadas por el objeto hijo para obtener información necesaria para llevar a cabo los pasos del comportamiento que tiene programado.

Veamos con más detalle cada uno de los componentes:

AO. Objetos de aplicación Representan componentes particulares de aplicación en el sentido de que, para cada función que aparece en el ACG, existe un correspondiente AO en la jerarquía. Si la aplicación actual ya existe y está escrita con un lenguaje no orientado a objetos pero se pretende reutilizarla, entonces, cada función será encapsulada en un nuevo objeto AO donde el código de la función será uno de los métodos del objeto. Si por el contrario, la aplicación es nueva, se escribirá en algún lenguaje orientado a objetos y se creará un nuevo AO por cada función de aplicación.

En cualquier caso, el conjunto de todos los objetos de clase AO constituye la *máquina funcional*. La función de disparo perteneciente a esta máquina funcional determina la lógica funcional de negocio representada por el diagrama ACG. Cada AO de la jerarquía propuesta es siempre un objeto hijo de un y sólo un objeto de control (CO), el cual es responsable de llevar a cabo la función común.

CO. Objetos de control Son los responsables de manejar otro componente de interfaz en el sentido de que para cada paso a dar para llevar a cabo la composición de la presentación, existe el correspondiente CO en la jerarquía. La composición de las presentaciones permite ocultar el concepto

[11] Modelo–Vista–Controlador, *(Model–View–Controler)*.
[12] Presentation, Abstraction & Control.

de ventanas (que son dinámicamente secuenciadas) bajo el concepto de unidades de presentación. Las distintas unidades de presentación se materializan en el contexto requerido para llevar a cabo su tarea de interacción. La jerarquía de CO puede construirse a partir del mismo esquema, por lo que los siguientes CO son de utilidad:

- **CO-IT**: el único objeto de control correspondiente a la tarea interactiva *(interactive task)*.
- **CO-PU**: los objetos de control correspondientes a las unidades de presentación *(presentation units)*.
- **CO-W**: el objeto de control correspondiente a la ventana (window).
- **CO-Fc**: los objetos de control correspondientes a las funciones de la aplicación *(application functions)*.

Cada CO correspondiente a una función de aplicación (CO-Fc) es un objeto hijo de un objeto de control ventana (CO-W) que alberga a todos los CIO necesarios para lanzar la función o servicio asociado en la máquina funcional.

IO. Objetos de interacción Los IO constituyen la interfaz de usuario propiamente dicha, de un lado los controles de entrada/salida IO-I/O *(input/ output interaction objects)* y, por otro lado, los elementos de control (botones, iconos, etc.) denominados IO-P *(presentation interaction objects)*.

Estos objetos que son seleccionados por un sistema experto [Bodart94], son equivalentes a los CIO y pertenecen a entornos físicos como Windows, OSF/Motif, OS/2 Presentation Manager, etc.

Cada IO, bien sea IO-I/O o IO-P, es un objeto hijo de un objeto de control de función (CO-F), los cuales envían eventos que representan los cambios de estado significativos. Además, tienen definidas primitivas para permitir que el objeto padre pueda interrogar su estado. En las implementaciones, la mayor parte de las veces, estas primitivas son métodos nativos proporcionados por las librerías de controles empleadas.

Semántica de las relaciones en la arquitectura

La semántica de las relaciones involucradas en la arquitectura (*véase la Figura 2.17*) es la siguiente:

Semántica de la relación CO-IT → CO-PU: El objeto de control correspondiente a la tarea interactiva (CO-IT) gestiona el cambio dinámico de unidades de presentación, que carga y descarga (de modo secuencial o concurrente) cada PU_i de acuerdo a los eventos recibidos de un PU_j.

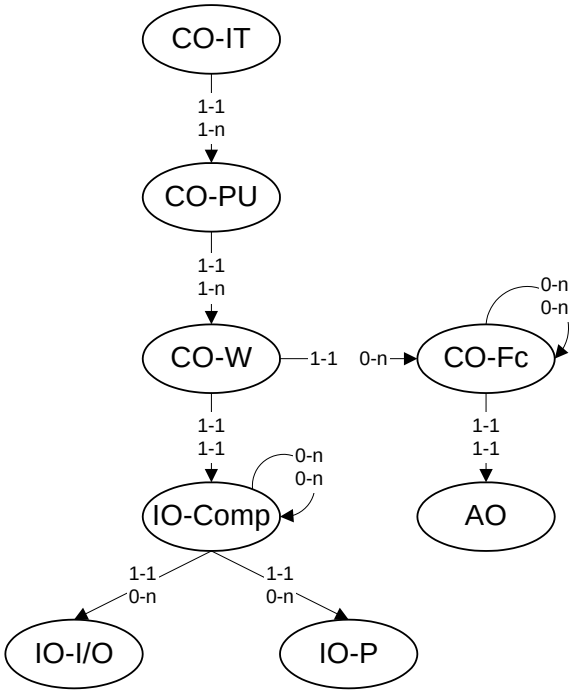


Figura 2.17: Relaciones entre los Objetos de Control en TRIDENT (adaptado de [Bodart95d]).

Semántica de la relación CO-PU → CO-W: El objeto de control correspondiente a una unidad de presentación (CO-PU) gestiona el cambio dinámico de las ventanas, que muestra u oculta la ventana W_i de acuerdo a los eventos recibidos desde otra ventana W_j.

Semántica de la relación CO-W → CO-Fc: El objeto de control correspondiente a una ventana (CO-W) invoca al objeto de control correspondiente a la función de aplicación cuando todas las condiciones requeridas por esta llamada son completadas, ésto es, cuando toda la información necesaria para realizar la función está disponible en los objetos hijos de CO-W. El CO correspondiente a la función de aplicación (CO-Fc) envía al CO-W un evento alertando de la terminación del proceso.

Semántica de las relaciones recursivas entre CO-Fc: Los CO asociados a las funciones de aplicación intercambian eventos entre ellos mismos para mantener la dinámica del ACG consistente y se llaman mutuamente a sus servicios para intercambiar estas informaciones.

Semántica de las relaciones entre IO ... y CO-W: Los IO, sean IO-I/O o IO-P, son los objetos que directamente atienden las reacciones del usuario. Los IO generan eventos destinados a su padre cuando el estado cambia. Los objetos padre llaman a los servicios ofertados por los IO para conocer que acciones han sido realizadas por el usuario.

La clave en la arquitectura TRIDENT consiste en ofertar algoritmos para convertir adecuadamente el diagrama ACG en una jerarquía compuesta por los objetos y relaciones de agente anteriormente descritas. La propia arquitectura separa claramente la interfaz de la funcionalidad, encapsulando dicha funcionalidad en el conjunto de AIO que forman la denominada máquina funcional.

Los conceptos de AIO, CIO y Unidad de Presentación se han revelado como abstracciones muy útiles a la hora de abordar el problema de la interfaz de usuario.

Por otro lado, la experiencia con TRIDENT [Bodart95d] evidencia que el modelo ofrece buenos resultados sobre aplicaciones de gestión de complejidad moderada sin requerimientos de tiempo real. TRIDENT no es un modelo de propósito general.

El analista tiene ahora la nueva responsabilidad de realizar el análisis de tareas y de generar los correspondientes diagramas ACG. Dependiendo de la calidad del análisis realizado, la interfaz se ajustará en mayor o menor grado a lo esperado.

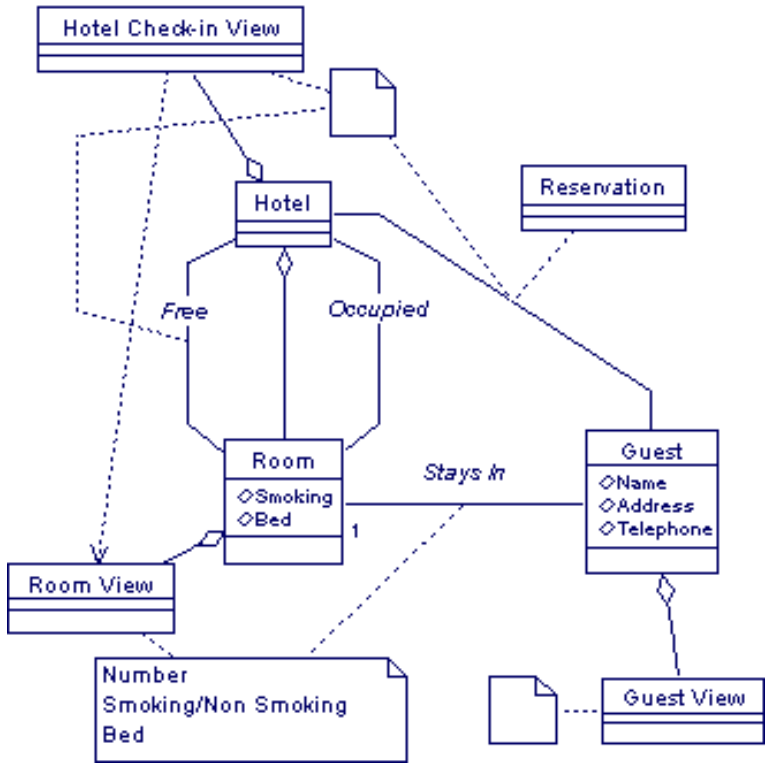


Figura 2.18: Diagrama de clases decorado con vistas [Roberts98].

En resumen, TRIDENT es una herramienta de referencia dentro del mundo de herramientas para el diseño de interfaces de usuario soportadas en modelos. En este proyecto se asentaron las bases de los conceptos de AIO, CIO y Unidad de Presentación. Dada la importancia y el uso que de dichos conceptos se hace en esta tesis, se ha considerado conveniente proporcionar una revisión más detallada de esta herramienta frente a otras también consideradas.

2.2.4. OVID

OVID [Roberts98] (*Objects, Views, and Interaction Designs*, Objetos, vistas y diseños de interacción) es una extensión a UML para abordar el modelado de interfaces de usuario.

Usando el concepto de estereotipo definido en UML, en OVID se define el concepto de vista *«View»* como un estereotipo de clase (*véase Figura 2.18*). Una vista es una presentación de un objeto perteneciente a una clase dada. A partir de modelos UML anotados con vistas, los diseñadores pueden prototipar de modo manual la vista (*véase Figura 2.19*).

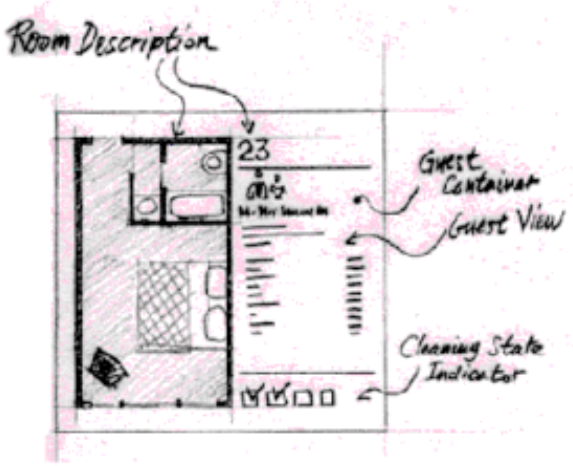


Figura 2.19: Boceto en papel de la vista correspondiente a una habitación de hotel [Roberts98].

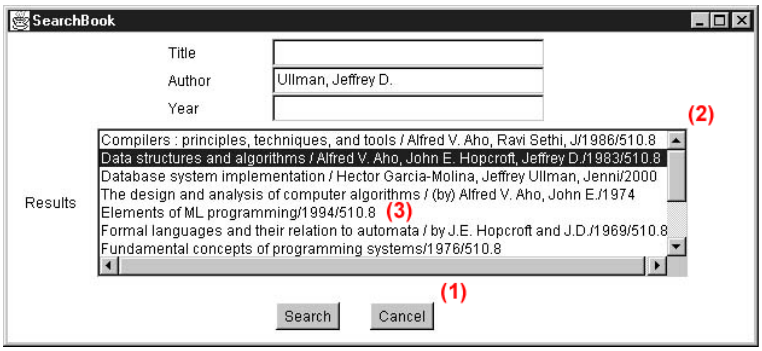


Figura 2.20: Ejemplo de interfaz a modelar [Silva02].

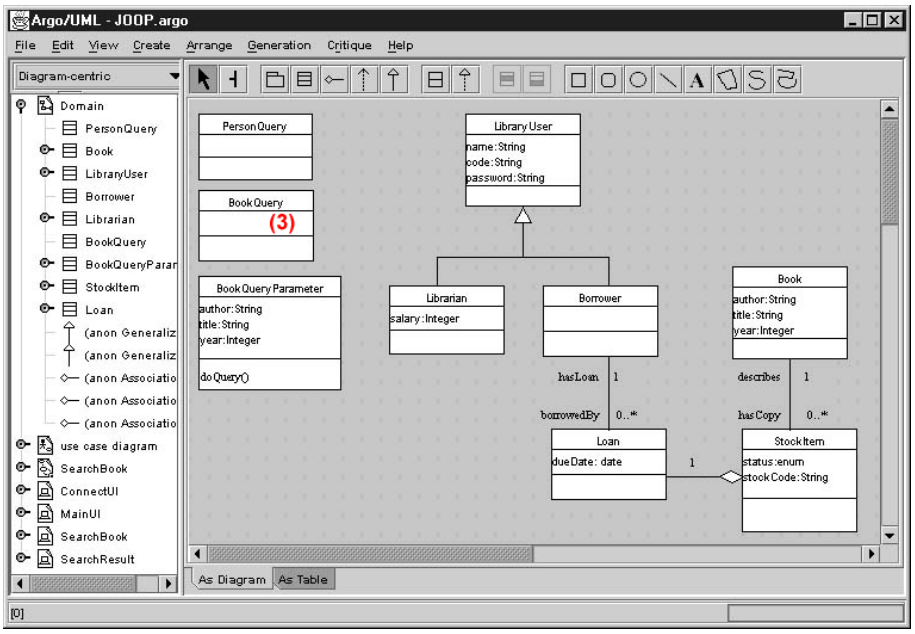


Figura 2.21: Dominio de aplicación [Silva02].

OVID es el primer método que se propone extender UML para la especificación de interfaces orientadas a objetos. Sin embargo, la especificación es poco formal y la correspondencia entre las vistas y la implementación se deja totalmente abierta a criterio del diseñador.

2.2.5. UMLi

UMLi [Silva02, Silva01] es el trabajo de investigación de Paulo Pinheiro da Silva para extender UML con conceptos y diagramas para abordar la especificación de interfaces de usuario en UML.

La Figura 2.20 muestra un ejemplo de interfaz de usuario a modelar. En las sucesivas figuras se muestran los diagramas asociados a la especificación de este ejemplo.

El modelo de dominio (*véase Figura 2.21*) muestra un análisis orientado a objetos clásico de las clases de análisis participantes en el sistema.

UMLi introduce un nuevo diagrama: El diagrama de presentación de interfaz de usuario (*véase un ejemplo en la Figura 2.22*) Las primitivas de este diagrama se denominan *Primitive Interaction Objects* y son las siguientes:

- **Inputter**. Representa un campo de entrada donde el usuario proporciona

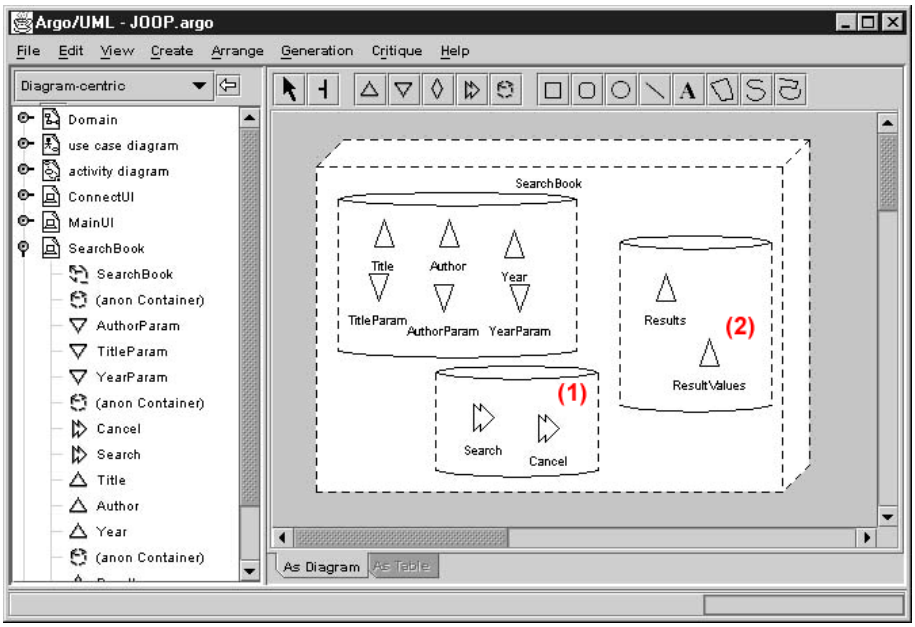


Figura 2.22: Presentación de interfaz de usuario [Silva02].

información al sistema. Se representa gráficamente mediante un triángulo isósceles invertido $\bigtriangledown$.

- **Displayer**. Representa un campo de salida donde el sistema proporciona información al usuario. Se representa gráficamente mediante un triángulo isósceles $\bigtriangleup$.
- **Editor**. Es una primitiva compuesta: se comporta como *Inputter* y *Displayer* al mismo tiempo. Se representa mediante un rombo $\diamond$.
- **Container**. Representa un contenedor de objetos de interacción. Un contenedor puede contener a otros contenedores. Se representa mediante un cilindro de trazo discontinuo.
- **FreeContainer**. Representa un contenedor de primer nivel, es decir, no puede estar contenido dentro de otro contenedor. Se representa mediante un prisma de trazo discontinuo.

También se introducen una serie de estereotipos para extender la interacción de flujo entre objetos en los Diagramas de Actividad (*véase Figura 2.23*):

- **«presents»** Conecta un *FreeContainer* con una actividad, indicando que la actividad es activada cuando se presenta la interfaz descrita por el *FreeContainer*.

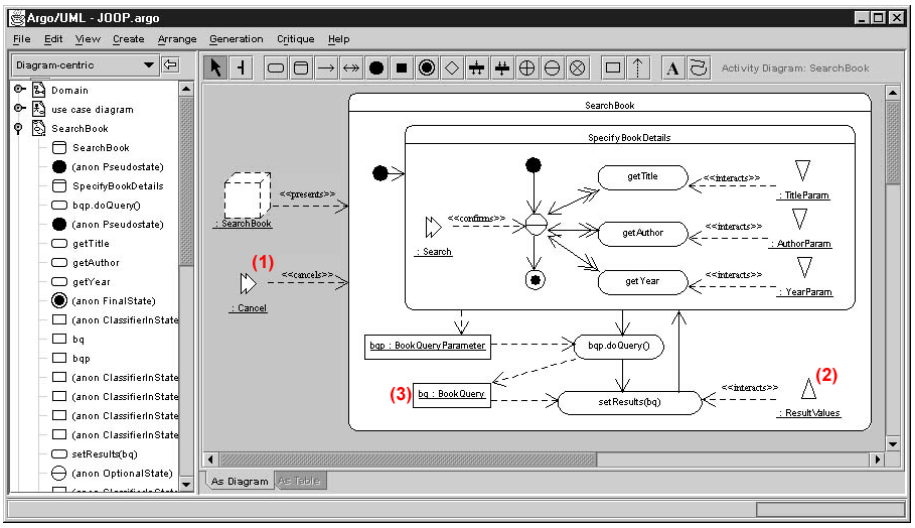


Figura 2.23: Modelo de comportamiento [Silva02].

- **«interacts»** Conecta un *Inputer*, *Displayer* o *Editor* con una tarea. Indica que la tarea está ligada a la interacción con estas primitivas.
- **«cancels»** Relaciona una tarea con un *ActionInkover*. Representa que la acción del usuario cancela la tarea en curso.
- **«activates»**Para disparar una tarea.

Por último, se introducen estereotipos a los estados, denominados *SelectionStates*, también para los Diagramas de Actividad:

- **«OrderIndepenceState»** Indica independencia de orden. Se representa gráficamente mediante el símbolo $\oplus$.
- **«OptionalState»** Indica opcionalidad. Se representa gráficamente mediante el símbolo $\ominus$.
- **«RepeatableState»** Indica posibilidad de repetición. Se representa gráficamente mediante el símbolo $\otimes$.

Las Figuras 2.21, 2.22 y 2.23 muestran cómo queda la especificación del ejemplo mostrado en la Figura 2.20.

UMLi constituye el primer intento serio de extender UML para capturar la interfaz de usuario de un modo formal. Sin embargo, la fina granularidad de los conceptos manejados conlleva problemas graves de escalabilidad. Los problemas de tamaño medio se tornan complejos y tediosos de especificar (*véase*

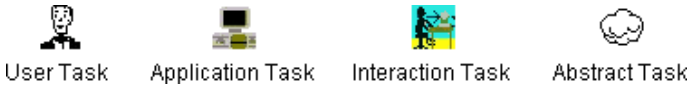


Figura 2.24: Categorías de tareas en ConcurTaskTrees [Paternò00].

Ley de Novak, en la página 98), lo cual dificulta en gran medida su adopción en entornos de desarrollo industriales.

Por otro lado, las primitivas empleadas son útiles para describir los interfaces WIMP. Para otro tipo de interfaces como las interfaces gráficas, las primitivas pueden no ser las adecuadas para representarlas.

2.2.6. CTT

ConcurTaskTree [Paternò00] es una notación ampliamente aceptada en el campo de la especificación de Modelos de Tareas. La notación está soportada por el lenguaje formal LOTOS [ISO88]. Fabio Paternò y su equipo en el CNUCE CNR de Pisa han desarrollado una herramienta CASE (*ConcurTaskTree Environment* [Paternò02]) que soporta el diagramado con esta notación.

La notación define cuatro categorías de tareas:

- **Tareas de usuario:** (*User Tasks*) Tareas llevadas a cabo por el usuario. Normalmente son actividades cognitivas, como pensar una estrategia para resolver un problema.
- **Tareas de aplicación:** (*Application Tasks*) Tareas realizadas por la aplicación. E.g., presentar una serie de resultados al usuario.
- **Tareas de interacción:** (*Interaction Tasks*) Tareas realizadas por el usuario al interactuar con el sistema. E.g., pulsar un botón o una tecla.
- **Tareas abstractas:** (*Abstract Tasks*) Tareas que requieren actividades complejas y que pueden ser descompuestas en otras más sencillas.

ConcurTaskTree usa una representación gráfica para cada categoría de tareas como muestra la Figura 2.24.

Una tarea puede ser descompuesta en subtareas. Las subtareas están relacionadas entre sí por medio de relaciones temporales. La notación ConcurTaskTrees proporciona un conjunto de operadores temporales basados en la semántica del lenguaje LOTOS. Los operadores aparecen descritos en las Tablas 2.2 y 2.3.

Operador	**Descripción**
Ocurrencia independiente $(T_1 \| \| \| T_2)$	Las tareas pueden acontecer en cualquier orden sin restricción. E.g. monitorizar una pantalla y hablar por un micrófono.
Independencia de orden $(T_1 \| = \| T_2)$	Las subtareas pueden acontecer en cualquier orden pero no al mismo tiempo.
Alternativa $(T_1 [\,] T_2)$	Elección entre un conjunto de tareas. Una vez se ha seleccionado una tarea, el resto de tareas no estarán disponibles hasta que la tarea seleccionada haya sido completada. E.g. al iniciar un procesador de textos, el usuario puede abrir un documento existente o crear uno nuevo.
Concurrencia con intercambio de información $(T_1 \| [\,] \| T_2)$	Dos tareas pueden ejecutarse concurrentemente pero han de sincronizarse para intercambiar información. E.g. en un procesador de texto el usuario puede editar y desplazar verticalmente el texto en cualquier orden. Sin embargo, sólo es posible editar la información que es visible en la región actual.
Desactivación $(T_1 [> T_2)$	La tarea T_1 es definitivamente desactivada una vez que la primera acción de la segunda tarea T_2 comienza. Este concepto es usado en las interfaces de usuario cuando el usuario puede deshabilitar un conjunto de tareas y habilitar otro conjunto. E.g. tras pulsar un botón.
Habilitar $(T_1 >> T_2)$	La tarea (T1) al finalizar habilita la realización de la segunda (T2). E.g. un usuario ha de conectarse primero al sistema para poder tener acceso a los recursos.
Habilitar con paso de información $(T_1 [\,] >> T_2)$	En este caso, la tarea T_1 proporciona a T_2 más información que la mera habilitación. E.g. T_1 permite al usuario especificar una consulta y T_2 proporciona los resultados de la consulta que, evidentemente, dependen de la información suministrada por T_1.

Tabla 2.2: Operadores temporales en ConcurTaskTrees 1/2.

Operador	**Descripción**
Suspender-Reanudar $(T_1 \mid > T_2)$	Este operador permite que T_2 interrumpa la ejecución de T_1. Cuando T_2 finaliza, T_1 puede ser reanudado desde el punto en el que quedó antes de la interrupción. E.g. un usuario puede interrumpir sus tareas de edición por la aparición de una ventana modal de impresión. Una vez la impresión ha finalizado, puede retomar la edición justo en el punto donde lo dejó.
Iteración $T*$	La tarea se realiza de modo repetitivo. Una vez finalizada, la tarea comienza desde el principio una y otra vez hasta que es interrumpida por otra tarea.
Iteración finita $T(n)$	Se emplea cuando se conoce a priori el número de repeticiones de la tarea.
Tarea opcional $[T]$	La tarea puede ejecutarse o no. E.g. al rellenar un formulario, algunos campos son opcionales y pueden no ser completados.
Recursión	Significa que en el subárbol generado por esta tarea aparece a su vez la misma tarea como subtarea. Dicho proceso ocurre recursivamente hasta que la tarea es interrumpida por otra.

Tabla 2.3: Operadores temporales en ConcurTaskTrees 2/2.

La Figura 2.25 muestra un ejemplo de árbol de tareas especificado con la notación ConcurTaskTree. El ejemplo muestra las tareas implicadas en una una unidad de interacción orientada a la búsqueda y selección de objetos. Al mismo tiempo, en la figura puede apreciarse el entorno de trabajo que proporciona el editor CTTE (*ConcutTaskTree Environtment*) para la creación de especificaciones de modelos de tareas soportados por la notación CTT.

Una utilidad adicional muy interesante en la herramienta CTTE es la posibilidad de poder animar la especificación para comprobar si el comportamiento es el esperado. La Figura 2.26 muestra el mismo modelo de tareas que el mostrado en la Figura 2.25, durante el proceso de animación.

La herramienta CTTE también permite obtener una especificación LOTOS equivalente al modelo de tareas especificado.

ConcurTaskTree se ha convertido en la notación más usada para el modelado de tareas. La riqueza de los operadores, la sencillez de los conceptos, unido a la disponibilidad de un editor libre de licencia son las principales causas de este hecho.

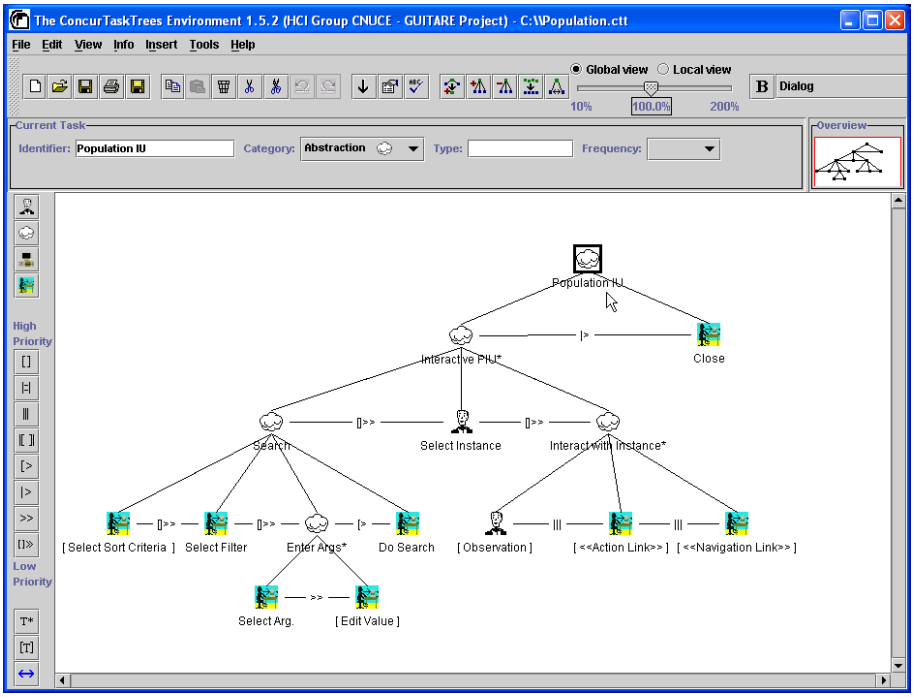


Figura 2.25: Herramienta CTTE.

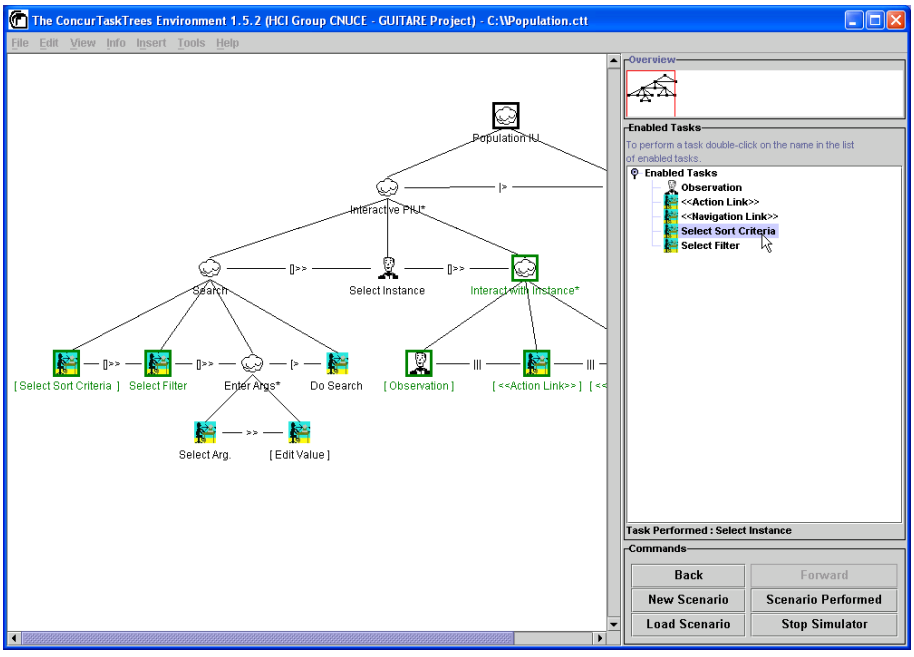


Figura 2.26: Herramienta de animación de modelos CTT.

2.2.7. MOBI-D

MOBI-D (*Model-Based Inteface Designer*) [Puerta96, Puerta97a, Puerta97b, Puerta97c] es un conjunto de herramientas desarrolladas por Ángel Puerta y su grupo de investigación en el proyecto MECANO, desarrollado en la Universidad de Stanford en California y financiado por DARPA.

El proyecto MECANO está soportado por un conjunto de herramientas:

- **MOBI-D** es el editor de especificaciones propiamente dicho. Permite establecer correspondencias (*mappings*) entre diferentes elementos de los diversos modelos.
- **UTel** es una herramienta para el soporte a la elicitación de requisitos que será usada como paso previo a la fase de especificación.
- **TIMM** permite parametrizar la generación de ventanas con aspectos de diseño una vez la especificación ha sido construida.
- **MOBILE** permite diseñar la interfaz de usuario. El diseño está soportado por el modelo de tareas y modelo de usuario construidos con las herramientas previas.

El editor MOBI-D

Consta de un editor de especificaciones (*véase Figura 2.27*) que recoge aspectos relativos a los siguientes modelos:

- Modelo de Dominio
- Modelo de Tareas
- Modelo de Presentación
- Modelo de Diálogo
- Modelo de Usuario

UTel

UTel (*User Task Elicitation*)[Tam98] es una herramienta para el soporte a la elicitación de requisitos (*véase Figura 2.28*). A partir de un texto en lenguaje natural, la herramienta ayuda al analista a construir modelos de dominio, de usuario y de tareas, coloreando en el texto los diferentes elementos localizados. UTel se basa en un diccionario pre-etiquetado dependiente del dominio del

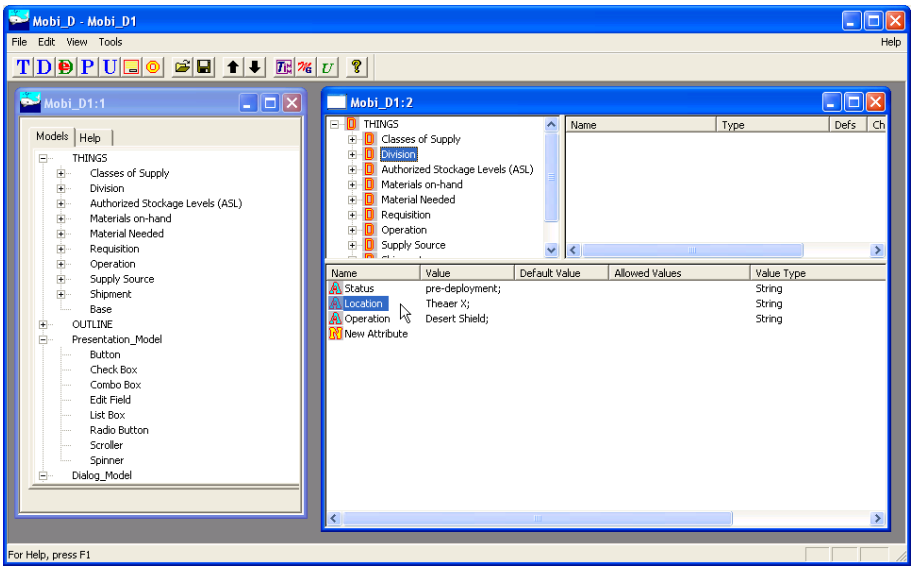


Figura 2.27: Herramienta MOBI-D.

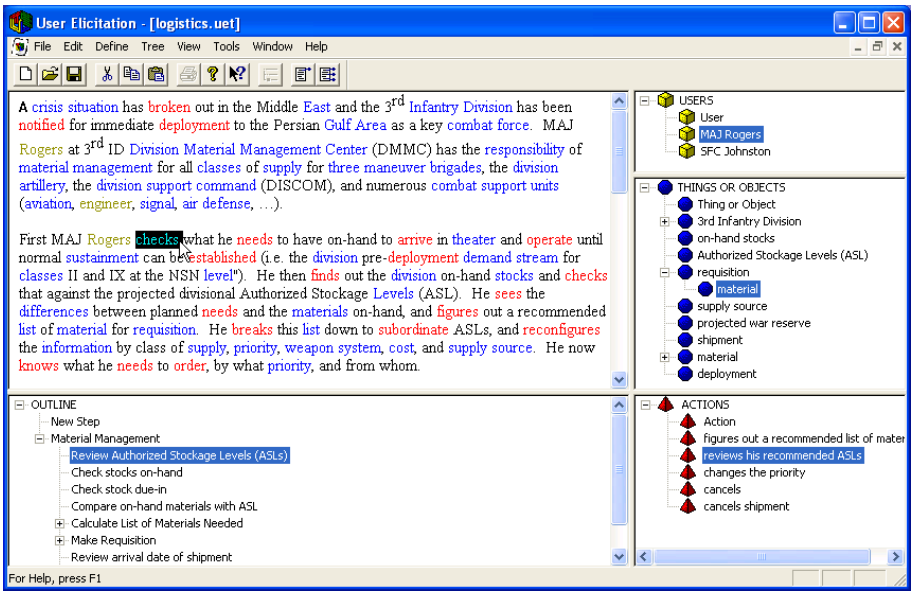


Figura 2.28: Herramienta UTel.

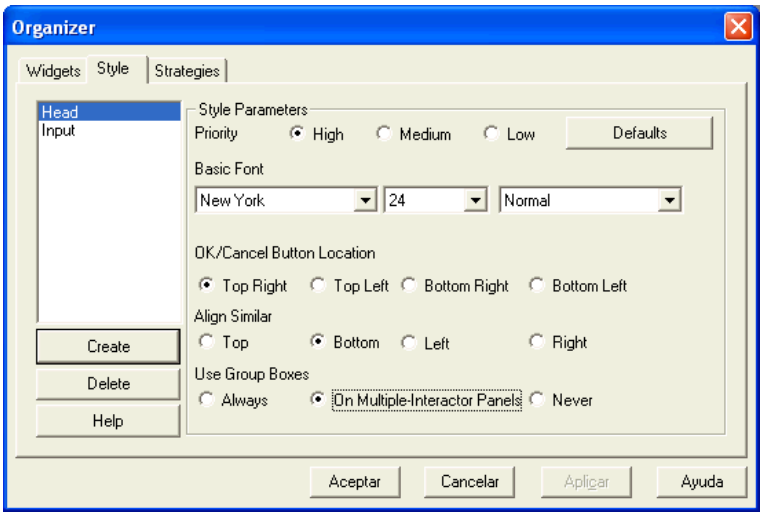


Figura 2.29: TIMM: parámetros de diseño para generar ventanas en MOBI-D 1/2.

problema donde las palabras en lenguaje natural son tipificadas como verbos (tareas), nombres (usuario) y objetos (clases y objetos del dominio). De este modo es posible analizar un texto en lenguaje natural y sugerir los usuarios, objetos del dominio así como las tareas del sistema.

Por tanto, el análisis de requisitos es semi-automático: el sistema proporciona una primera lista de objetos de dominio, tareas y usuarios. Es responsabilidad del analista supervisar el proceso para detectar errores cometidos por el sistema así como organizar de modo jerárquico la información detectada.

TIMM

TIMM (*Task Interface Modeler Mapper*) es un módulo soportado por una base de conocimiento de guias de estilo de interfaces de usuario para ayudar a los diseñadores a tomar decisiones globales. Permite asociar decisiones de diseño a los objetos del modelo que van a ser generados. La interfaz de usuario de TIMM contiene las pestañas (*Widgets*, *Style* y *Strategies*, *véase Figuras 2.29 y 2.30*) Las decisiones de diseño que pueden indicarse abarcan:

- La definición de estilos, comprendiendo fuente gráfica, tamaños, alineación en el formulario, uso de agrupadores.
- Selección de estrategias de generación como son: la ocultación o deshabilitación de controles hasta el momento de su uso, el deshabilitado de ta-

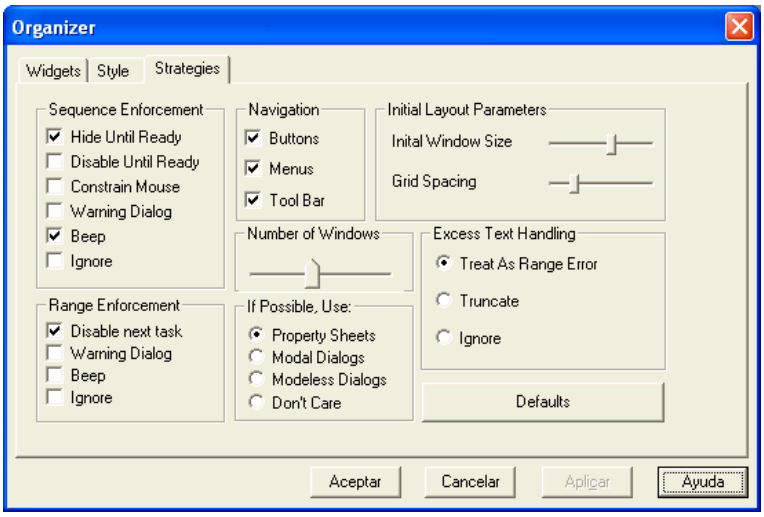


Figura 2.30: TIMM: parámetros de diseño para generar ventanas en MOBI-D 2/2.

reas, la implementación de la navegación (por medio de botones, menús o barras de herramientas), el tamaño y número de ventanas, etc.

Las decisiones de diseño son almacenadas junto con la especificación. Estas decisiones permiten cierto grados de libertad para el afinado de la generación de código de acuerdo a la guía de estilo o particularidades del proyecto considerado.

MOBILE

MOBILE (*Model-Based Interactive Layout Editor*) es un editor de diseño para dotar a interfaz de usuario de aspectos de geometría.

En apariencia externa MOBILE funciona de modo análogo a un entorno para la creación de interfaces de usuario. Sin embargo, hay diferencias adicionales. Las decisiones de presentación y dialogo están guiadas por las tareas de usuario y el modelo de usuario. La base de conocimientos proporcionada por TIMM restringe las decisiones de diseño a aquellas apropiadas a cada paso.

Cada control en la paleta de MOBILE representa a un control ActiveX o un Java Bean. Esta característica es crucial para alcanzar un reconfiguración en interfaces adaptativas.

2.2.8. Wisdom

Wisdom (*Whitewater Interactive System Development with Object Models*) [Nunes00b, Nunes00a, Nunes01] es un método de modelado fuertemente basado en UML para la especificación de sistemas interactivos. Sus autores, Nuno J. Nunes de la Univeridad de Madeira y João Cunha de la Universidad de Oporto, orientan esta propuesta como método de desarrollo para pequeñas compañías de software.

La propuesta persigue alcanzar la aplicación final a partir de ciclos de modelado y de prototipado evolutivo.

Wisdom se asienta sobre tres bases: Proceso, Arquitectura y Notación.

Proceso

El proceso sigue un modelo evolutivo en espiral [Boehm88] centrado en el usuario. Se construyen prototipos que son completados en sucesivas fases hasta alcanzar el producto final. El proceso es dirigido por casos de uso y flujos de tareas.

Arquitectura

Wisdom presenta un modelo de arquitectura para cubrir las cuatro etapas de desarrollo de las que consta: toma de requisitos, análisis, diseño y evaluación.

Los modelos propuestos (*véase Figura 2.31*) son: un modelo de dominio o de negocio, un modelo de casos de uso, un modelo de análisis, un modelo de interacción, un modelo de diseño, un modelo de diálogo, un modelo de presentación y por último, un modelo de implementación. Todos ellos usan algún modelo existente en UML o bien, extienden algún modelo existente con estereotipos.

La arquitectura de interacción propuesta en Wisdom introduce dos estereotipos sobre clases para dar cuenta de dos nuevas dimensiones el diálogo y la presentación en UML. Los estereotipos Tarea «*Task*» y Espacio de Interacción «*Interaction Space*».

Notación

Se hace uso un subconjunto de UML junto con las extensiones basadas en estereotipos de UML como mecanismo de extensión para introducir los diferentes conceptos.

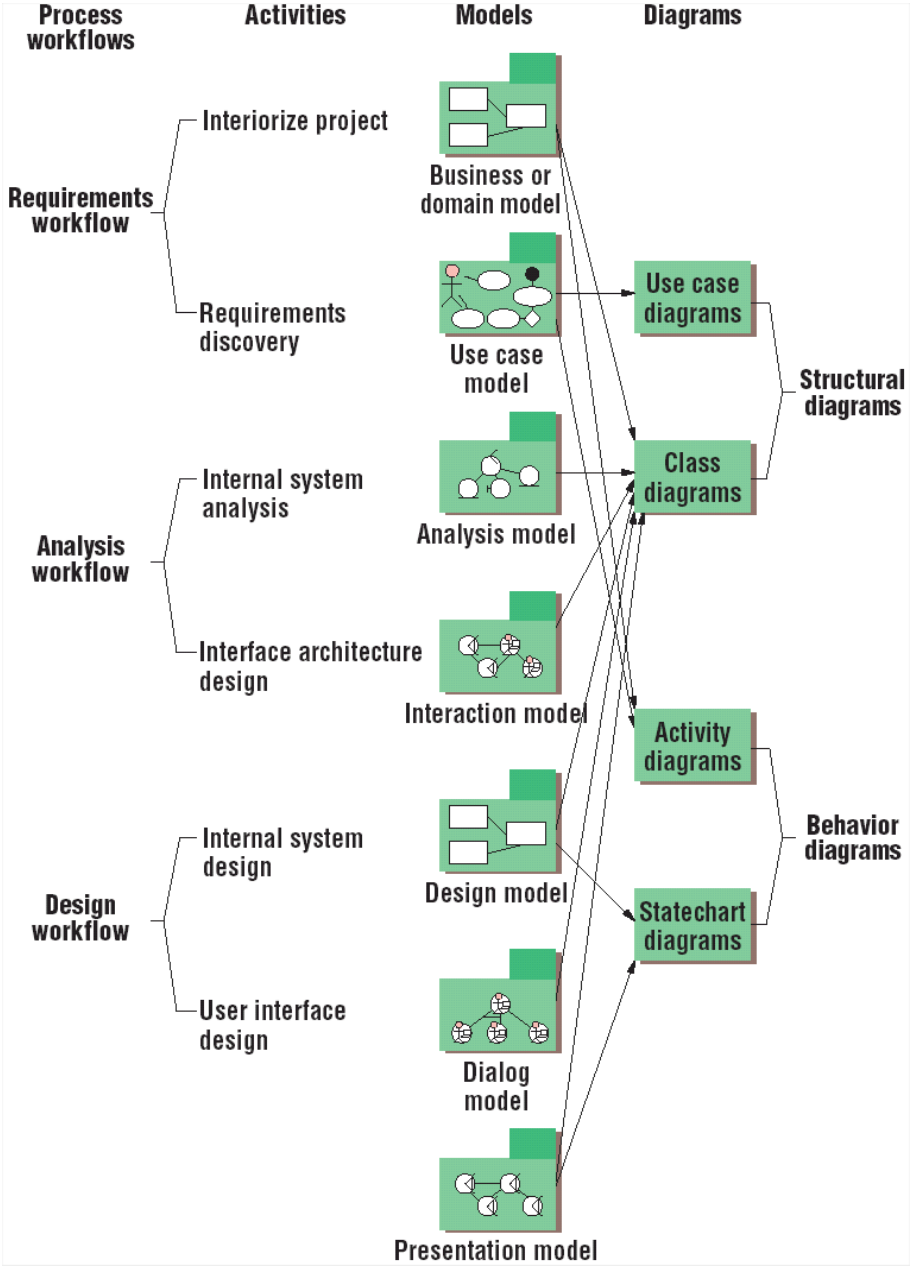


Figura 2.31: Relación entre procesos y modelos en Wisdom (adaptado de [Nunes00b]).

Como trabajo académico, WISDOM todavía no cuenta con herramientas CASE para soportar por completo el método. En particular, las especificaciones deben construirse usando herramientas de terceros como Rational Rose o VISIO. La fase de implementación tampoco está soportada por herramientas: es responsabilidad de los diseñadores y programadores la implementación final de la aplicación. Como se ha comentado con anterioridad, el método está adaptado para pequeñas empresas de software. Por esta razón, para proyectos grandes, el método debe ser revisado teniendo en cuenta consideraciones de escalabilidad.

2.2.9. Otras aproximaciones

En paralelo con este trabajo de tesis, en grupos próximos de investigación se han venido desarrollando trabajos en el campo de la especificación de interfaces de usuario. Los más sobresalientes son los siguientes:

- IDEAS [Lozano01] es un trabajo para la especificación de interfaces de usuario llevado a cabo por Mª Dolores Lozano en la Universidad Politécnica de Valencia. IDEAS propone una serie de modelos de refinamiento para el diseño y la construcción de interfaces de usuario. Una extensión a $\mathcal{OASIS}$ ha sido propuesta para incorporar las ideas de descomposición de tareas y un framework de diseño, ambos propuestos en [Jaaski95].

- Por otro lado, Juan Sánchez [Sánchez01b] en la misma universidad ha investigado con éxito el uso de la especificación de interfaces de usuario a partir de Casos de Uso complementados en una segunda fase por el formalismo MSC (*Message State Chart*) [ITU96].

- OO-$\mathcal{H}$Method [Gómez01] es un trabajo de investigación llevado a cabo en la Universidad de Alicante por Cristina Cachero para la especificación de la interfaz de aplicaciones Web basadas en modelos y herramientas de análisis y diseño.

- Por último, de nuevo en la Universidad Politécnica de Valencia, OOWS (*Object Oriented Web Solutions*) [Pastor00a] (iniciado por Alberto Aparicio y continuado por Silvia Abrahão y Juan J. Fons) es un trabajo para explorar la de especificación de aspectos navegacionales en entornos de aplicaciones Web como complemento a OO-Method en la línea de trabajos como [Rossi01, Ceri00].

2.3. Conclusiones del capítulo

Las herramientas de modelado presentadas (Rational Rose, Together, Argo UML, System Architect) tienen una fuerte penetración en el mercado como herramientas comerciales. Sin embargo, adolecen de soporte para la especificación de interfaces de usuario y su principal uso es la documentación y modelado a nivel de diseño (dependiente del lenguaje, plataforma y arquitectura destino).

Por otro lado, TRIDENT, MOBI-D, CTTE, pueden encuadrarse como herramientas MB-UIDE.[13] Las herramientas de tipo MB-UIDE son la punta de lanza en la investigación para la especificación y desarrollo de interfaces de usuario basadas en modelos. Por contra, son proyectos académicos que han sido poco probados en la construcción de sistemas reales.

Esta separación de la funcionalidad (recogida mediante un modelo de análisis) de la interfaz de usuario (recogida a través de un MB-UIDE) es una separación altamente contraproducente. Obliga a mantener dos especificaciones que comparten contenidos de modo separado y obliga a velar para mantenerlas sincronizadas.

Un modelo de especificación único que dé cuenta de la funcionalidad y la interfaz de usuario al mismo tiempo es más adecuado para abordar el problema que pretendemos resolver –construir aplicaciones completas–, puesto que, todos estos aspectos aparecerán en la aplicación final.

En esta línea, OVID, UMLi y Wisdom son trabajos encuadrados en la línea de extender la notación UML (usada para la especificación de aspectos estructurales y de comportamiento) con aspectos para la especificación de interfaces de usuario.

Sin embargo, si se pretende producir código para la aplicación final a partir de modelo, es necesario poder disponer de una semántica precisa en el lenguaje de modelado. A día de hoy, UML no tiene tal semántica estandarizada, lo cual dificulta la creación de traductores y generadores de código.

En el presente trabajo de tesis se pretende dar respuesta a estos problemas con las siguientes apuestas:

- Proporcionar un modelo único con un soporte formal para la especificación de sistemas a nivel conceptual (sin consideraciones dependientes de plataforma) que integre no sólo de la funcionalidad del sistema, sino también de la interfaz de usuario.

[13] *Model Based-User Interface Development Environment*.

- Explorar la utilización de patrones de interfaz de usuario a nivel conceptual como mecanismo para la elicitación, especificación y posterior diseño e implementación automática de las interfaces de usuario.
- Proporcionar un método soportado en generación automática completa para permitir la evolución de la especificación por medio de ciclos de prototipado muy rápidos.

Capítulo 3

Fundamentos

«Quien a buen árbol se arrima,
buena sombra le cobija.»

— Anónimo, Refranero popular español.

Para facilitar la comprensión de los conceptos tratados en los sucesivos capítulos, se hace necesario introducir previamente las áreas temáticas en las que se basa la presente tesis. El presente capítulo está dedicado a introducir:

- el **Modelado Conceptual:** como herramienta para la especificación de sistemas,
- la **Interfaz de Usuario:** como campo de trabajo donde se definen los conceptos que debemos capturar en la especificación,
- los **Patrones:** como abstracciones útiles para abordar el problema de la identificación de los conceptos de especificación y, por último,
- las **Técnicas de Generación de Código:** como tecnología clave para la obtención de las interfaces finales a partir de las especificaciones.

3.1. Modelado Conceptual

Un esquema conceptual es una descripción **formal** que describe la estructura y el comportamiento de un sistema con un nivel de abstracción superior a los lenguajes de programación, más cercano al modo de razonamiento humano. Las propiedades que proporciona el empleo de un método formal garantizan la corrección y completitud de las especificaciones obtenidas de este modo.

Por el contrario, las especificaciones informales, sean descripciones en lenguaje natural o sean notaciones gráficas sin una semántica definida, conducen a especificaciones propensas a contener un gran número de ambigüedades donde las interpretaciones del modelo pueden variar de analista en analista.

Los lenguajes de especificación permiten construir esquemas conceptuales complejos mediante el empleo de un conjunto de abstracciones proporcionadas por el lenguaje formal que le sirve de base.

OASIS [Pastor92a, Pastor95, Letelier98], Oblog [ESDI93] y Troll [Hartmann94] son lenguajes formales orientados a objetos para la especificación de sistemas. Mediante conceptos como clase, objeto, agregación, herencia, etc. permiten describir sistemas complejos como composición de términos interrelacionados a partir de estas primitivas esenciales.

Estos tres ejemplos de lenguajes formales definen gramáticas y establecen unas restricciones léxicas, sintácticas y semánticas que ha de cumplir una especificación bien escrita en el lenguaje.

Una especificación en un lenguaje formal es una descripción textual en términos del lenguaje y que satisface las restricciones léxicas, sintácticas y semánticas de dicho lenguaje.

3.1.1. UML

UML *(the Unified Modeling Language)* [OMG99] es una notación estándar para el modelado. En primera instancia, UML fue en sus inicios una muy buena idea: tratar de unificar diferentes las notaciones para el modelado de diversas metodologías. Sin embargo, lo que parecía prometedor quedó sólo en eso. UML es una unificación de la notación pero, sin embargo, no define una semántica precisa para la notación.

Tal y como especifica el documento de OMG [OMG99], la especificación recoge sólo la notación. Sin embargo, el Modelado Conceptual requiere, además de notación, de una semántica precisa.

A día de hoy, UML es una notación ampliamente extendida gracias al esfuerzo de estandarización y soportada por multitud de herramientas comerciales. Sin embargo, la aproximación inicial tomada de (*one size fits all*, «UML sirve para todo») se ha tornado en un problema grave, acentuado por la falta de semántica precisa en UML y desembocando en el uso de las notaciones en múltiples contextos diferentes con convenciones semánticas informales en cada contexto. El estándar MDA (*Model Driven Architecture*) [OMG01] ha venido a intentar organizar la especialización y el uso de sub-modelos de UML para desarrollos en dominos particulares.

Diferentes grupos de investigación en el mundo han detectado esta carencia y tratan de paliar el problema desarrollando propuestas para dotar de semántica precisa a UML: *Action Scrip Language* propuesto por un consorcio [ASL01], el grupo académico *Precise UML* [Pre02] o la propuesta del consorcio *2U* [2U02] son algunos ejemplos de propuestas para ser incluidas en UML v. 2.0.

Por otro lado, Rational Corp. patrocina RUP *(Rational Unified Process)* como método para soportar el desarrollo de aplicaciones. Junto con RUP, Rational proporciona toda una *suite* de herramientas para soportar su método. Sin embargo, la semántica sigue sin ser precisada.

En cualquier caso, ni en UML 1.3 (versión vigente) ni el el futuro UML 2.0 se ha avanzado hacia la incorporación de notación y semántica para la especificación de interfaces de usuario. Esta tarea concreta es uno de los objetivos perseguidos por esta tesis.

WAE–UML

Conallen [Conallen00] propone una extensión a UML para la especificación de aplicaciones Web (*WAE, Web Application Extension*).

WAE usa los mecanismos de extensión propuestos en UML (estereotipos, extensión de propiedades y restricciones: véase Tabla 3.1) para proponer nuevas primitivas adaptadas a la idiosincrasia de las aplicaciones Web. WAE define una serie de estereotipos para diseñar páginas Web. Para el programador, el nivel de abstracción se eleva: en lugar de trabajar con páginas y *scripts*, dispone de modelos gráficos para representar esos mismos conceptos de implementación.

WAE proporciona semántica para cada nuevo concepto introducido. Esto es un paso adelante si lo comparamos con la ausencia total de semántica de la notación UML.

Sin embargo, es reprochable en WAE que los estereotipos estén tan fuertemente ligados a la tecnología subyacente (*véase Figura 3.1*, p. e.: `<JSP Page>`, `<ASP Page>`, `<JavaScript Object>`, `<IIOP>`, `<RMI>`, `<Servlet>`, etc.) y no se haya intentado elevar un poco más el nivel de abstracción para evitar la moda tecnológica. Obviamente, cuando la tecnología cambie, los estereotipos tendrán que ser extendidos con nuevos elementos, otros tendrán que ser adaptados y, finalmente, algunos de ellos desechados por obsolescencia.[1]

Por tanto, de cara a abordar un sistema, parece razonable abordar el análisis sólo con UML y en la fase de diseño e implementación, una vez hemos fijado la

[1] Si algo podemos asegurar en informática sin miedo a equivocarnos, es que las tecnologías y lenguajes cambian, evolucionan. Todavía más aprisa si cabe, dentro de la Web, donde los expertos hablan de años–Web para referirse a ciclos de tres meses.

Clase	Estereotipo	Descripción
Class	<Server Page>	Página que se ejecuta en el servidor. Las operaciones representan las funciones de la página y los atributos las variables en el alcance de la página.
Class	<Client Page>	Página HTML con código *script* que se envía al navegador.
Class	<Form>	Los atributos de la clase representan los campos de un formulario (<FORM>) en HTML .
Class	<Frameset>	Contenedor de varias páginas Web.
Class	<Target>	Define un elemento al cual se puede navegar.
Class	<JavaScript Object>	Objeto cliente implementado en Javascript
Class	<ClientScript Object>	Objeto *script* descargado por el cliente.
Association	<Link>	Un <Link> es un enlace a otra página para definir la navegación.
Association	<Targeted Link>	Similar a <Link>, pero al cruzar el destino la página alcanzada se muestra en otro destino (<Target>).
Association	<Frame Content>	Es una asociación que expresa el contenido de un <Frame>.
Association	<Submit>	Expresa la relación de envío entre un <Form> y un <ServerPage>.
Association	<Builds>	Una <Server Page> construye una <Client Page>.
Association	<Redirect>	Una <Server Page> redirecciona a otra.
Association	<IIOP>	Expresa comunicación IIOP (*Internet Inter-ORB Protocol*) entre objetos cliente y servidor.
Association	<RMI>	Expresa comunicación RMI (*Remide Method Invocation*) entre objetos Java cliente y servidor.
Attribute	<Input Element>	Expresa un elemento <INPUT> de HTML.
Attribute	<Select Element>	Expresa un elemento de selección en HTML.
Attribute	<Text Area Element>	Expresa un elemento <TEXTAREA> de HTML.
Component	<Web Page>	Representa una página Web.
Component	<ASP Page>	Página Web que ejecuta código ASP (*Active Server Pages*).
Component	<JSP Page>	Página Web que ejecuta código JSP (*Java Server Pages*).
Component	<Servlet>	Componente Java Servlet.
Component	<Script Library>	Librería de subrutinas o que pueden ser usadas desde las páginas.

Tabla 3.1: Estereotipos definidos en WAE–UML.

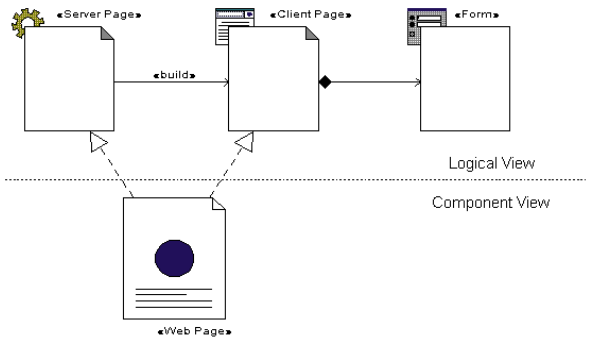


Figura 3.1: Ejemplo de diagrama WAE.

arquitectura, plataforma destino y tecnologías de implementación, abordar el diseño usando la extensión WAE–UML, sabiendo que tal diseño, está muy ligado a la implementación (otras aproximaciones como, por ejemplo, [Winckler02] logran elevar el nivel de abstracción). En este sentido, no podemos decir que WAE–UML sea apto para la especificación o análisis de un sistema, sino que, más bien, modela el código de la aplicación web final.

3.1.2. OASIS

OASIS [Pastor92a, Pastor92b, Pastor95, Letelier98] *(Open Active Specification of Information Systems)* es un lenguaje de especificación formal orientado a objetos que sirve de base formal a la metodología OO-Method. Entre las características del lenguaje podemos destacar:

- Una especificación *OASIS* se formaliza como teorías de primer orden que evolucionan con el tiempo por la ocurrencia de eventos y que tienen una base formal natural, correcta y completa, dentro del contexto de la Lógica Dinámica.
- Se define también un álgebra de procesos para objetos (APO) con el objetivo de caracterizar con propiedad y entender los objetos como procesos observables.

Definición de clases

Una clase se compone de un nombre de clase, una función de identificación para instancias (objetos) de la clase y un tipo o plantilla que todas las instancias comparten.

El nombre de la clase puede ser cualquier nombre que identifique unívocamente a la definición de la clase. La función de identificación caracteriza el mecanismo de nombramiento usado por las distintas instancias de una clase. Ésta da un conjunto de identificadores pertenecientes a un *sort* (género).

Estos *sorts* se importan en la definición de la clase. Los más usados están predefinidos como `int`, `nat`, `real`, `bool`, `char`, `string` o `date`. Representan números, valores lógicos, caracteres, cadenas y fechas en un formato particular. Se pueden añadir nuevos dominios en una especificación definiendo el correspondiente Tipo Abstracto de Datos.

Un tipo es una plantilla que recoge todas las propiedades (estructura y comportamiento) compartidas por todos los objetos potenciales de la clase considerada. Sintácticamente, se puede formalizar como una signatura, que contiene:

- *Sorts*: funciones de identificación, atributos y eventos,
- un conjunto de axiomas, que son fórmulas en una lógica dinámica (derivaciones, precondiciones, evaluaciones, restricciones de integridad y disparos) y
- una sección de procesos, que es un conjunto de ecuaciones nombradas cada una de ellas por variables del *sort* proceso y que son expresadas en el álgebra de procesos para objetos. Cuando estas variables se instancian, tenemos los términos *básicos* que representan las posibles vidas de las instancias (objetos).

Signatura de clases

Una signatura de clase contiene:

- Un conjunto de *sorts* con una relación de orden parcial. Entre estos conjuntos de *sorts*, tenemos en particular el *sort de interés* (el nombre de la clase) asociado con la clase que está siendo definida.
- Un conjunto de funciones incluyendo:
  - La función de identificación cuyo *sort* es el *sort de interés*. Esto nos proporciona valores de un *sort* dado para identificar objetos.
  - Un conjunto de atributos (constantes, variables y derivados). Todos ellos tienen como dominio el *sort* de la clase, y como codominio el *sort* dado asociado al atributo considerado.
  - Un conjunto de eventos, con el *sort* de la clase como dominio (más algún *sort* adicional representando la información de eventos) y con el *sort* de la clase *(sort de interés)* como codominio.

Cada ocurrencia de evento estará etiquetada por el actor al que se le permite iniciarlo. Cuando se trabaja con la noción de actor, escribimos $i : a$ si el actor i inicia el evento a, y decimos que $i : a$ es una acción. El actor i podría ser el entorno o cualquier objeto de una clase del sistema.

No se concibe un evento sin actor. Cuando definimos un evento, el diseñador está obligado a decir quién podrá activarlo. Como consecuencia, asumiremos que está definido un conjunto A de acciones, obtenido desde y unido al conjunto inicial de eventos.

En $\mathcal{OASIS}$, las ideas de tomar la iniciativa en la ejecución de los eventos y visibilidad de los mismos, están separados introduciendo las nociones de actor e interfaz respectivamente. Con la declaración de interfaces representamos qué eventos son visibles a qué actores. Los actores representan quién ejecuta los eventos, mientras que las interfaces expresan qué eventos son visibles a qué actores.

Para profundizar sobre aspectos particulares del lenguaje en lo referente a sus propiedades sintácticas y semánticas consúltese [Letelier98].

3.1.3. OO-Method

OO-Method es una método orientado a objetos de producción automática de software que tiene como objetivo fundamental fusionar las buenas propiedades de los lenguajes de especificación formales con el pragmatismo de los métodos informales de análisis y diseño de software. OO-Method emplea diagramas gráficos semejantes a la notación UML con una semántica precisa soportada por el lenguaje formal $\mathcal{OASIS}$ [Pastor95, Letelier98]. De este modo, por medio de diagramas gráficos se logra ocultar la complejidad subyacente de una especificación formal textual. Los analistas son más productivos y proclives a usar herramientas gráficas de especificaciones. Por el contrario, son mucho más reacios a emplear herramientas textuales que no escalan bien para grandes proyectos con complejidad creciente.

OO-Method aborda la tarea de construcción de software en dos fases:

- El **Modelo Conceptual**, situado en el espacio del problema, para el cual se proporciona un potente lenguaje gráfico que permite recoger las propiedades de los requisitos del problema en cuestión. Con este modelo se cubre totalmente la fase de análisis. A partir de esta información, y utilizando mecanismos de traducción determinados por el método, se genera automáticamente una completa especificación formal en $\mathcal{OASIS}$ que actúa como repositorio de alto nivel del sistema.

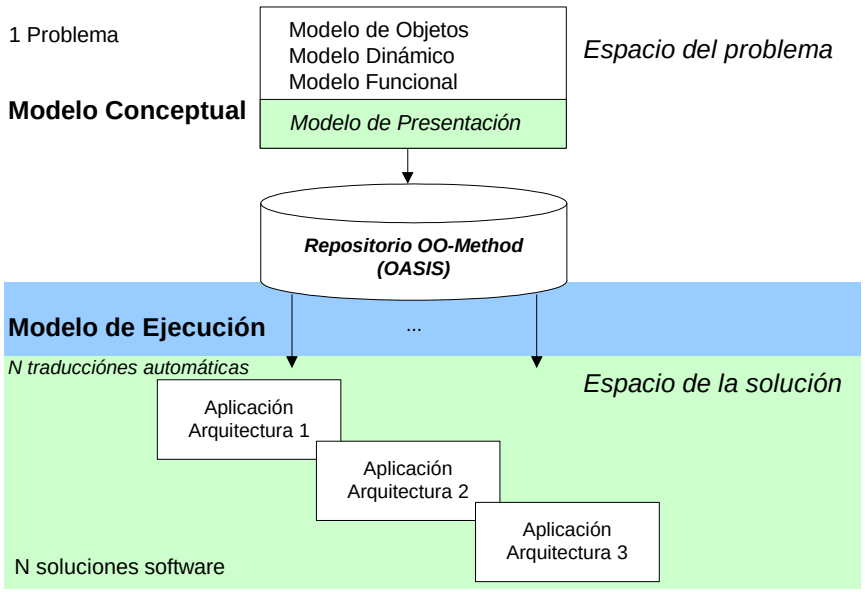


Figura 3.2: Fases de OO-Method para la producción de aplicaciones.

- El **Modelo de Ejecución**, situado en el mundo de la solución, fija los detalles de implementación (en cuanto a control de acceso, interacción con el usuario, persistencia de objetos, etc.) para cada uno de los elementos del Modelo de Conceptual en un entorno de desarrollo particular. De esta manera se obtiene un producto funcionalmente equivalente a las especificaciones del problema recogidas en el Modelo Conceptual.

En la Figura 3.2 se ilustra la relación entre los distintos modelos del método. En particular, *el Modelo de Presentación* que aparece resaltado en color es el que se introduce como principal aportación de esta tesis. Este Modelo de Presentación constituye de por sí, el cuarto modelo de OO-Method.

Modelo Conceptual

El modelado conceptual es la primera fase del proceso de construcción de software. En ella se recoge información del dominio del problema (sin tener en cuenta aspectos de implementación) y se construyen tres modelos que permiten especificar claramente la estructura y el comportamiento de los componentes de la Sociedad de Objetos estudiada, que son: el Modelo de Objetos, el Modelo Dinámico y el Modelo Funcional.

- El *Modelo de Objetos* es un Modelo Semántico de Datos, extendido con

la declaración de eventos y agentes activadores de eventos, que fija el marco arquitectónico del sistema modelado desde una perspectiva estática. Con él quedan declaradas las clases componentes tanto elementales como complejas (clases definidas en función de los operadores de herencia y agregación) y su estructura interna (servicios y atributos). La declaración de agentes especifica dentro del modelo qué objetos (vistos como clientes) están autorizados a activar los servicios ofrecidos por las clases (vistas como servidoras).

- El *Modelo Dinámico* establece cuáles son las vidas válidas para los objetos de una clase dada, ordenando temporalmente la secuencia de ocurrencia de servicios (eventos y transacciones) y especifica la interacción inter-objetual.

- El *Modelo Funcional* permite especificar la funcionalidad de cada *evento*[2], definiendo cual es su efecto sobre el estado del objeto implicado en términos de los atributos que modifica.

La perspectiva estática del Modelo de Objetos se complementa con los modelos Dinámico y Funcional y fija de esta forma la estructura y el comportamiento de los objetos. Los tres modelos en conjunto definen desde puntos de vista complementarios las propiedades (estáticas y dinámicas) de los objetos identificados en el sistema estudiado.

Modelo de Ejecución

La segunda fase del método consiste en la generación automática de código en un entorno de desarrollo dado que implementa la estructura y el comportamiento del sistema, aspectos estos recogidos en la fase de análisis.

Sea cual sea la plataforma destino, el código producido sigue una pautas comunes que permiten garantizar la equivalencia funcional entre implementaciones y conforme al modelo origen. Estas pautas constituyen el llamado Modelo de Ejecución [Gómez98, Gomez99].

El Modelo de Ejecución considera el sistema como una *Sociedad de Objetos*.[3] El usuario mismo, juega el rol de objeto al ser considerado como un objeto instancia de una clase agente.

Para alcanzar este comportamiento, el sistema tiene que seguir los siguientes pasos:

[2] Un *evento* en OO-Method es un método atómico (simple) de una clase.
[3] Un sistema *vivo* en el cual los objetos interaccionan entre si por medio de paso de mensajes.

1. **Identificar al usuario** (control de acceso). Acceso al sistema por parte del usuario (*login*) donde tras un proceso de autenticación un objeto agente impersonará al usuario conectado ante la sociedad de objetos. El sistema proporcionará una vista a medida para el usuario conectado dependiendo de los permisos (visibilidad sobre atributos, servicios y roles[4]) de la clase agente considerada.

2. **Permitir la activación de servicios.** Una vez el usuario tiene una vista clara del sistema de objetos, este puede activar cualquier servicio que tenga disponible.

Entre los servicios se incluyen observaciones sobre los objetos (consultas) o eventos y transacciones ofertados por otros objetos. Cualquier activación de servicio tiene dos pasos: construir el mensaje y ejecutarlo (si tal extremo es posible). Para construir el mensaje el usuario tiene que proporcionar la siguiente información:

1. **Identificar el objeto servidor.** La existencia del objeto servidor es una condición implícita para la ejecución de cualquier servicio, a menos que estemos considerando un servicio de creación.[5]

2. **Proporcionar los argumentos al servicio.** Los argumentos adicionales para el servicio deben ser proporcionados (siempre que sean necesarios).

Una vez que el mensaje es enviado, la ejecución de un servicio viene caracterizada por la ocurrencia de la siguiente secuencia de acciones en el objeto servidor:

1. **Comprobar la transición de estado.** Consiste en verificar si en el diagrama de transición de estados (DTE) asociado a la objeto servidor existen una transición válida en el estado actual del objeto.[6] Si no existe dicha transición válida, se producirá una excepción y el mensaje será ignorado.

2. **Satisfacción de precondiciones.** La precondición asociada al servicio debe cumplirse. Si no es así, al igual que el caso previo, se producirá una excepción y el mensaje será ignorado.

[4] **Rol:** Nombre que etiqueta cada uno de los extremos de una relación de agregación.

[5] En cuyo caso el objeto servidor seria el meta-objeto (*la clase*) que oferta el servicio de creación como método factoría para la creación de instancias. Existe un meta-objeto para cada clase. Dicho meta-objeto contiene como atributo principal la población de la clase, el siguiente OID y los citados servicios de creación.

[6] Existe una transición válida en el estado ϕ para el servicio s si existe una transición con origen en ϕ etiquetada con el servicio s. Es decir: $\exists\ \phi\ [s]\ \phi'$.

3. **Satisfacción de valuaciones.** Las modificaciones inducidas por los eventos (indicadas en el Modelo Funcional) son satisfechas alterando de este modo, el estado del objeto servidor.

4. **Comprobación de las restricciones de integridad en el nuevo estado.** Para asegurar que la ejecución de un servicio conduce a un estado válido, las restricciones de integridad (estáticas y dinámicas) son verificadas sobre el estado final del objeto. Si las restricciones no son satisfechas, una excepción será disparada y los cambios realizados en el estado son completamente ignorados descartando el mensaje original; devolviendo, por tanto, el sistema su estado original previo a la recepción del mensaje.

5. **Comprobación de disparos.** Después de producirse el cambio tras la ocurrencia de un servicio las reglas de condición-acción que representan la actividad interna del sistema (*triggers*) deben ser comprobadas. Si alguna condición se satisface, el servicio especificado sera disparado.

Los pasos descritos constituyen la guía de implementación de cualquier programa derivado de una especificación OO-Method. Esta guía permite asegurar equivalencia funcional entre la especificación del sistema de objetos descrito en el esquema conceptual y su correspondiente reificación en una implementación dada.

Es importante resaltar que el Modelo de Ejecución descrito en OO-Method es independiente del entorno de desarrollo. Por tanto, se podrán obtener implementaciones particulares de este Modelo (como traductores de Modelos Conceptuales) para cualquier entorno de producción de software independientemente de la arquitectura de aplicación destino. Como ejemplo podemos comentar que el Modelo de Ejecución ha sido implementado para diversas arquitecturas COM y Java/EJB dentro de entornos Internet/Intranet. Y que del mismo modo, ha sido implementado con éxito en implementaciones monolíticas, cliente/servidor (bicapa), tres capas (persistencia, lógica de negocio, interfaz de usuario) y cinco capas lógicas (aplicaciones Web como extensión a la arquitectura previa considerando temas de distribución en Web).

Las técnicas de Modelado Conceptual constituyen el primer pilar sobre el que se basa esta tesis. En esta sección se ha introducido un rápido recorrido donde se ha mostrado UML, como la principal notación para el modelado usada actualmente, *OASIS* y OO-Method, como precedentes directos de este trabajo.

3.2. Interfaces de Usuario

3.2.1. Definición

Descomponiendo un sistema en capas lógicas según su función, podemos ver a la interfaz de usuario como a la capa lógica encargada de dar soporte al diálogo con el usuario. Su funciones, son básicamente dos:

Entrada: (*adquisition*) adquirir las órdenes, información y comandos expresados por el usuario a través de diversos dispositivos de interacción.

Salida: (*restitution, rendering*) presentar resultados, retroalimentación y cooperar para facilitar al usuario la realización de las tareas que pretende resolver el sistema.

3.2.2. Evolución de las interfaces de usuario

En los albores de la ciencia informática, las interfaces de usuario eran espartanas, muy limitadas por la tecnología existente. Conforme las capacidades de los equipos han ido creciendo en prestaciones, hemos pasado de la época de los terminales de texto, a los menús textuales hasta llegar a las actuales interfaces gráficas de usuario *(GUI, Graphical User Interface)* que añaden capacidades multimedia a la presentación tradicional de datos lo que constituye un salto cualitativo en la calidad de interacción Hombre-Máquina.

La llegada de las interfaces de usuario gráficas, cambiaron la forma en la que los informáticos diseñaban las interfaces de usuario. La interfaz de usuario comenzó a tener más peso en el proceso de desarrollo, lo que favoreció la aparición de herramientas de desarrollo rápido de aplicaciones donde se ayuda al diseñador a ir construyendo visualmente la interfaz final: *Delphi*, *Visual Basic*, *Power Builder* son ejemplos de herramientas RAD apoyados sobre lenguajes de 3ª generación.

Las herramientas RAD, aunque facilitan la creación de interfaces de calidad, no bastan para asegurar la calidad de la interfaz producida, si no que depende en gran medida de las capacidades y criterio del diseñador que las usa. Son pues, herramientas útiles en el ámbito de diseño y de la creación artística.

A la vez que se recogen requisitos y se especifican sistemas, antes de crear el sistema propiamente dicho, los requisitos de interfaz deben ser también recogidos para dar lugar a una especificación que pueda ser validada. En este sentido, la comunidad investigadora en este campo comenzó el desarrollo de UIMS *(User Interface Management Systems)* que gestionan aspectos como especificación, validación, animación, y prototipación.

WIMP

Todos los entornos gráficos de usuario, desde el pionero *Xerox Star* [Johnson89] desarrollado en PARC *(Palo Alto Research Center)* por Xerox Corp. hasta los actuales como Windows 2000 [Cowart00], System X [Mac01] o X11 [X00] tienen muchos elementos en común. Estas características comunes están encerradas bajo las siglas WIMP acuñadas en Palo Alto que son el acrónimo en inglés de ventana, icono, menú y puntero *(Window, Icon, Menu, Pointer)*.

Window Ventanas y distribución del espacio de trabajo.

Icon Representación gráfica de un objeto manipulable.

Menu Selección de objetos / acciones.

Pointer Puntero para seleccionar objetos y/o acciones a través de dispositivos como: ratón, teclado, lápiz óptico, pantallas táctiles, guantes de datos, etc.

Veamós con un poco más de detalle que aporta cada uno de los cuatro elementos:

Ventana Las ventanas constituyen las unidades de trabajo más sencillas de identificar por parte del usuario (*véase figura 3.3*). Normalmente, las ventanas suelen apilarse unas encima de otras, simulando un efecto 3D para dar sensación de profundidad simulando ser hojas de papel sobre un escritorio. Hay una serie de características reseñables sobre las ventanas:

- **Multiventana:** Los entornos gráficos son interfaces de múltiples ventanas. Las ventanas son empleadas para dividir la pantalla en un conjunto de áreas donde varias aplicaciones pueden ejecutarse. Todos los entornos permiten mostrar simultáneamente muchas ventanas que pueden solaparse, cubrirse por completo o colocadas una tras otra formando un mosaico para compartir el espacio de trabajo.
- **Libertad de contexto:** La presentación simultánea de diversas ventanas permite trabajar en varias aplicaciones al mismo tiempo, que es la expresión visual de la multitarea.

  Además, esta interacción simultánea tiene restricciones. Cada ventana debe ser etiquetada correctamente para el que usuario pueda identificarla inmediatamente sin ambigüedad (si varias ocurrencias de una misma aplicación están ejecutándose al mismo tiempo, debe ser posible diferenciarlas cuando sea necesario). Del mismo modo, una aplicación no puede acaparar todos los recursos del sistema; por contra, debe diseñarse de modo que se permita su uso simultáneamente con otras aplicaciones.

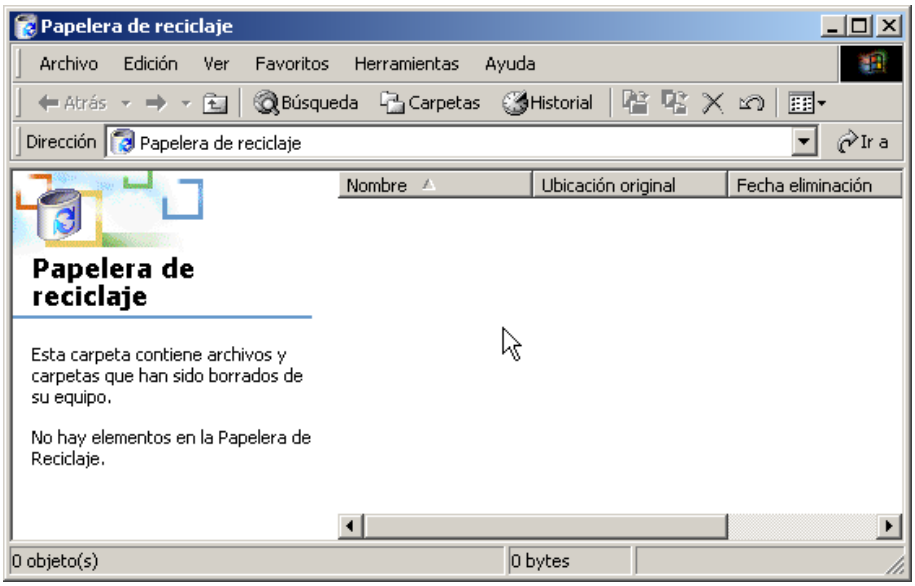


Figura 3.3: Ejemplo de ventana.

El usuario puede cambiar de un contexto (cada aplicación ejecutándose en una ventana representa un contexto) a otro seleccionando una ventana, obligando a que la ventana se sitúe en primer plano (sólo una ventana está en primer plano, todas las demás permanecen en segundo plano) y se active. La ventana de primer plano se activa automáticamente obligando a que todos los eventos de teclado y ratón sean dirigidos hacia ella.

- **No-modalidad:** Las aplicaciones y los documentos son siempre no-modales. Un diálogo modal confina al usuario a un modo de operación preestablecido, es decir, no puede mover el ratón o situar el foco en otra ventana hasta que concluye su interacción con esta ventana. En una interfaz no-modal el usuario siempre puede abrir tantas ventanas como desee e ir cambiando de una a otra a voluntad.
- **Espacio virtual ilimitado:** El usuario dispone de un espacio virtual ilimitado que es mucho mayor que el tamaño físico de la pantalla. Se pueden simular varias pantallas, reorganizarlas con funciones de mosaico y trabajar con barras de desplazamiento horizontales y verticales sobre un espacio virtual ilimitado.

Icono Los objetos se representan mediante pequeños pictogramas (*véase figura 3.4*) en lugar de sólamente etiquetas. El uso de estas metáforas hace que el entorno de trabajo se asemeje al entorno real del usuario.

Figura 3.4: Juego de iconos.

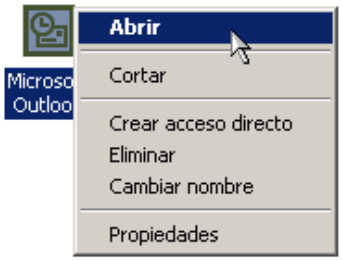


Figura 3.5: Ejempo de menú emergente.

Las personas interpretan la representación visual de objetos más fácilmente que su representación verbal. Por ejemplo, un icono que representa una impresora indica sin ambigüedad la función de impresión.

Menu El uso de menús (*véase figura 3.5*) hace posible el proceso basado en el modelo de objeto-acción: en primer lugar se selecciona el objeto y después seleccionar la acción a realizar sobre el objeto. La principal ventaja del modelo objeto-acción frente al modelo de acción-objeto radica en que una vez seleccionado el objeto o conjunto de objetos a afectar se puede aplicar sobre ellos varias acciones sin tener que volver a seleccionar de nuevo los objetos destino.

El proceso objeto-acción es el modo de interacción con la interfaz de usuario. Incluye tres fases: selección de uno o más objetos (por ejemplo: seleccionar un párrafo de texto), seleccionar la acción (acción Cortar del menú Edición) y finalmente producir el resultado (el párrafo desaparece del texto).

Puntero El puntero (generalmente controlado por un ratón aunque no es necesario) se usa para seleccionar objetos, seleccionar comandos de menú o

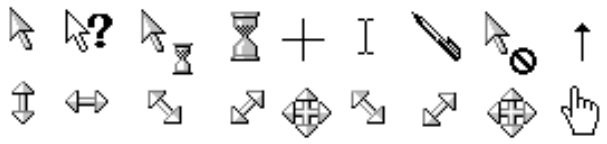


Figura 3.6: Juego de punteros.

cambiar entre contextos de una ventana a otra.

La apariencia del puntero sobre la pantalla puede variar (*véase figura 3.6*) de acuerdo al contexto (la herramienta seleccionada) o cuando se mueve sobre áreas sensibles de la interfaz (el borde de una ventana que puede alterar su tamaño).

El paradigma WIMP demostró con los ensayos preeliminares de Xerox y posteriormente con el éxito comercial de Apple con el Macintosh, la sencillez de uso que proporcionaba la nueva idea de entorno gráfico donde se buscaba mimetizar el escritorio de trabajo de una oficina con símbolos gráficos para el escritorio, carpetas, ficheros, papeleras, tijeras, pegamento, etc.

La gran mayoría de aplicaciones destinadas a usuarios finales no-informáticos emplean interfaces de usuario construidas sobre entornos gráficos. Es por ello que prestaremos especial atención a este tipo de entornos sin olvidar otros más recientes como la Web o interfaces para dispositivos móviles.

Los paradigmas de interacción: verbo–nombre vs. nombre–verbo

Foley [Foley82] definió los dos grandes paradigmas de interacción con un ordenador: *verbo–nombre* (o *acción–objeto*) frente a *nombre–verbo* (*objeto–acción*).

En el primero de ellos, *verbo–nombre*, el usuario selecciona en primer lugar la acción a realizar, posteriormente indica que objeto sufrirá dicha acción. Las aplicaciones basadas en interfaces de menu textuales suelen seguir este paradigma de interacción.

Por el contrario, en el paradigma *nombre–verbo*, el usuario selecciona primero el objeto con el que quiere trabajar y acto seguido, indica que acción sufrirá el objeto. Este modo de trabajo se puso de relieve con la aparición de interfaces de usuario gráficas donde las interfaces están más orientadas a la manipulación de objetos. Objetos que son dispuestos en el escritorio en forma de iconos y que el usuario puede manipular. Un claro y sencillo ejemplo es la operación de *drag & drop* (arrastrar y soltar) de un fichero a la papelera con la sencilla semántica asociada de *querer borrar el fichero*.

Los psicólogos y expertos en usabilidad sugieren que el trabajo de modo *nombre–verbo* es más cercano a la manera de pensar de los usuarios: primero *reconocimiento* (manipular el objeto para ver que es posible hacer con él), después *acción* (una vez conocidas las posibles acciones realizables sobre el objeto, realizar alguna de ellas). Así mismo, permite que los objetos puedan ser disociados de las acciones, así como definir acciones genéricas aplicables a todos los objetos.

Sin embargo, no hay una regla clara. Dependiendo del tipo de tarea a realizar, uno u otro paradigma puede ser más adecuado:

- Desde el punto de vista ergonómico, el paradigma *verbo–nombre* es más adecuado cuando una misma acción se va aplicar repetidamente sobre distintos objetos.
- Por el contrario, el paradigma *nombre–verbo* es más adecuado cuando sobre un objeto o conjuntos de objetos se van aplicar distintas acciones consecutivas.

Las aplicaciones modernas suelen permitir de modo simultaneo el trabajo con los dos paradigmas de modo que el usuario es libre de usar en cada momento la que le sea más cómoda. Por ejemplo, en un editor de textos, el usuario puede marcar negrita y después escribir el texto (*verbo–nombre*) o bien seleccionar un párrafo y marcar la acción poner en negrita (*nombre–verbo*).

3.2.3. Problemas de las interfaces inadecuadas

El abuso o uso inadecuado de las características gráficas de un entorno puede conducir al fracaso de la interfaz diseñada. Algunos problemas más frecuentes [Europea95, Molina98] son:

- **Confusión:** El usuario está perdido en la aplicación, no sabe dónde ésta ni que es lo que puede hacer. Este es el principal problema de las aplicaciones cuya profundidad o nivel de detalle es excesivo o que no proporcionan mensajes de información durante la ejecución de la operación en curso.
- **Frustración:** El acceso a las acciones no es intuitivo y el usuario no sabe como hacer lo que desea. La percepción que el usuario tiene en mente de la aplicación no se corresponde con su interfaz real, las metáforas usadas son confusas.
- **Pánico:** El usuario está atemorizado ante la posibilidad de destruir inadvertidamente ficheros o datos debido a que la aplicación no requiere

(o peor todavía: no siempre requiere) confirmación para acciones irreversibles.

- **Estrés:** El usuario se ve superado por la carga de trabajo para llevar a cabo las acciones y comete errores, que aunque pequeños, con la repetición a menudo se vuelven irritantes. Puede ser debido a que los diálogos son demasiado complicados (demasiada información aumenta el estrés) o a que la carga mental es excesiva (la cantidad de información que necesita ser memorizada para poder seguir trabajando sin necesidad de volver a consultar algo dejado atrás en otro escenario).
- **Aburrimiento:** Surge cuando el usuario tiene que repetir una y otra vez los mismos pasos para llevar a cabo una acción debido a que la aplicación es inflexible y no dispone de atajos o no está ajustada al trabajo que realiza el usuario.
- **Desaprovechamiento:** La aplicación no se usa en su totalidad debido a que es demasiado compleja.
- **Sobreexplotación:** La aplicación es tan ambigua que es necesario recorrer las ventanas y cajas de diálogo atrás y adelante para asegurarse que la acción a realizar es la correcta. El usuario tiene que abrir y cerrar un gran número de cajas de diálogo o ventanas antes de poder acceder a la funcionalidad que buscaba.

3.2.4. Ventajas de las interfaces correctas

El uso apropiado de las capacidades del entorno conduce a aplicaciones con interfaces agradables que satisfacen a los usuarios [Europea95, Molina98]. Algunos ejemplos de las ventajas producidas son:

- **Mejor aceptación:** El usuario acoge con agrado la aplicación en lugar de rechazarla y la considera herramienta esencial para su trabajo.
- **Mejor uso:** Cuando la interfaz representa fielmente la imagen mental de los usuarios, la aplicación se usa correctamente, minimizando los errores, y por tanto de modo eficiente. El rendimiento general es el correcto y los usuarios se benefician de una herramienta adecuada para su trabajo.
- **Mínimo entrenamiento:** La necesidad de entrenar a los usuarios se minimiza desarrollando aplicaciones consistentes ya que los usuarios pueden reaprovechar los conceptos ya adquiridos en otras aplicaciones y herramientas. Del mismo modo, si los usuarios tienen unos conocimientos mínimos en el manejo de aplicaciones, pueden descubrir las capacidades de la nueva aplicación rápidamente.

- **Documentación mínima:** Cuando el acceso es intuitivo y natural y la aplicación se corresponde con la imagen mental del usuario, la aplicación es autosuficiente. Si ofrece información por si misma acerca de sus funciones se reduce considerablemente la necesidad de consultar documentación adicional.
- **Reducción en el tiempo de desarrollo y facilidad de mantenimiento:** Una aplicación bien diseñada con una interfaz de usuario adecuada sigue principios de homogeneidad y reuso minimizando el número de escenarios (ventanas) con operatoria diferente y reduce el tiempo de desarrollo.

Todas ellas si están presentes en las interfaces de usuario ayudan a asegurar la calidad de percibida por el cliente de la aplicación.

Las cinco medidas cuantitativas mas usadas para medir la calidad de una interfaz de usuario son:

1. Tasa de errores cometidos por el usuario.
2. Tiempo de consecución de tareas de usuario.
3. Tiempo de aprendizaje.
4. Satisfacción del usuario.
5. Retención a lo largo del tiempo.

3.2.5. Principios básicos

A la hora de construir interfaces de usuario, bien sea de modo manual o bien sea de manera automatizada por medio de un generador de código, debemos tener muy presentes una serie de principios para intentar alcanzar los niveles de calidad exigibles a las aplicaciones WIMP actuales.

La presente sección pretende poner de manifiesto estos principios básicos ya que guiarán posteriormente el proceso de especificación de los requisitos de interfaz de usuario e inspiran buenas soluciones en la fase de construcción de interfaces de usuario. La motivación principal es detectar los atributos de calidad de cara a asegurar desde una fase temprana (análisis) la calidad de las interfaces de usuario especificadas y posteriormente obtenidas.

Usabilidad

La Organización Internacional de Estándares (*ISO*) proporciona dos definiciones de usabilidad en sendos estándares:

> *«La usabilidad se refiere a la capacidad de un software de ser comprendido, aprendido, usado y ser atractivo para el usuario, en condiciones específicas de uso.»*
>
> — ISO/IEC 9126[7]

> *«Usabilidad es la efectividad, eficiencia y satisfacción con la que un producto permite alcanzar objetivos específicos a usuarios específicos en un contexto de uso específico.»*
>
> — ISO/IEC 9241[8]

Ambas definiciones recalcan la idea de que la usabilidad es una medida las cualidades percibidas en el software por usuarios muy concretos en un contexto de uso muy localizado.

La usabilidad por tanto no puede ser medida en abstracto (o a priori), sino que ha de ser valorada a posteriori mediante un estudios empíricos.

Aún así, a través de la experiencia acumulada por los diseñadores a lo largo de los años, existen recomendaciones, principios, mejores practicas y guías de estilo que sí que permiten garantizar un mínimo de usabilidad a priori. Es en este punto cuando podemos hablar de ergonomía como una propiedad abstracta.[9]

Ergonomía

> **Ergonomía:** *«Conjunto de estudios, métodos y disposiciones para hacer el trabajo más humano en función de las capacidades fisiológicas y psicológicas del individuo.»*

La ergonomía es la *cualidad sin nombre* [Alexander64] imprescindible en el desarrollo de las interfaces de usuario. Algo tan simple como el empleo de fuentes gráficas o colores uniformes tiene un peso muy importante en la productividad que el usuario final es capaz de alcanzar con las aplicaciones. Por tanto, es preciso seguir un mínimo de principios ergonómicos para evitar que el uso de las aplicaciones se convierta en una pesadilla para los usuarios.

[7] ISO/IEC 9126. Disponible en: `http://www.usability.serco.com/trump/resources/standards.htm#9126-1`

[8] ISO/IEC 9241. Disponible en: `http://www.usability.serco.com/trump/resources/standards.htm#9241-11`

[9] Por supuesto, ello no exime de seguir verificando la usabilidad de lo que se construye sobre el terreno con el contexto de uso y sus usuario muy presentes.

Del mismo modo que en mecánica se establece un convenio universal para el sentido de giro de tornillos y tuercas, o en electrónica donde se siguen estándares de color para los cables, es lógico pensar que en informática deban estandarizarse aspectos de interfaz, desde como cerrar o abrir una ventana hasta como debe comportarse un botón o donde debe situarse el menú. En este sentido a finales de la década de los años ochenta y noventa han aparecido numerosas guías de estilo que tratan de fijar estos estándares en la creación de interfaces de usuario. Ejemplos remarcables de estas guías son Apple Desktop Interface [Computer87], CUA [IBM92], MS Windows 95 [Corp.95], OSF/Motif [Foundation93], KDE [Donohoe98, KDE99], Sun Open Look [Microsystems89], Java Look & Feel Design Guidelines [Microsystems99] y GNOME [GNO02]. El Workshop *Tool for Working with Guidelines* celebrado en 2000 es un claro exponente de la investigación en este campo [Vanderdonckt01].

Las guías aportan un modo de proceder preciso acerca de cómo diseñar interfaces de usuario en entornos o plataformas específicas. Estas guías favorecen que la curva de aprendizaje del usuario con una nueva aplicación sea mucho menor.

Dado que CUA sigue siendo la referencia para muchas guías de estilo, merece la pena detallar los principios en los que CUA se basa. El siguiente apartado está destinado a describir su espíritu, que debería estar presente en toda guía de estilo que pretenda proporcionar ergonomía.

CUA

El estándar CUA *(Common User Access)* [IBM92] fue desarrollado por IBM para servir de guía de ergonomía para aplicaciones que deben ejecutarse sobre entornos gráficos.

CUA describe los objetos gráficos, su papel y su comportamiento así como ciertos principios que deben seguirse en el desarrollo de aplicaciones en modo gráfico.

Una aplicación que cumple con los estándares de CUA es más sencilla de usar y es consistente con el entorno gráfico. Esta consistencia facilita la integración de la nueva aplicación con el entorno y con otras aplicaciones. Una aplicación es consistente si se muestra en todas las ocasiones con la presentación y comportamiento estándar del entorno gráfico.

CUA establece principios básicos que deben ser asimilados por los responsables del diseño de la interfaz de la aplicación. Aunque CUA proporciona gran cantidad de información acerca de ventanas, objetos, tipos, etc. no dice cuando usarlos, como secuenciar las ventanas o como construir un menú.

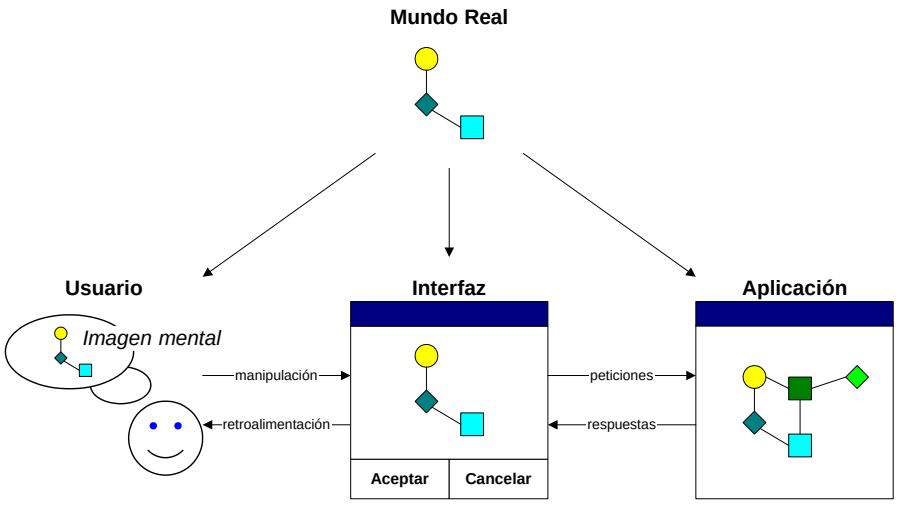


Figura 3.7: Imagen mental del usuario ante la interfaz de usuario.

Imagen mental del usuario

El principio fundamental de CUA pude resumirse en un concepto:

«Reproducir la imagen mental del usuario.»

Los usuarios generalmente esperan que la aplicación opere de acuerdo a su propia naturaleza. Por ejemplo, una hoja de calculo se emplea para manipular números que son dispuestos en celdas. La interfaz de usuario debe confirmar esta imagen mental produciendo los resultados esperados por el usuario.

La Figura 3.7 ilustra todos los elementos del problema. El problema presente en el mundo real ha sido mecanizado por medio de una aplicación que presenta una interfaz de usuario. En la medida en que el usuario conoce el problema a resolver a partir de su experiencia en el mundo real y en la medida en que la interfaz proporciona de modo natural una organización estructural próxima al problema original, el usuario irá construyendo una imagen mental acerca de qué se puede hacer en la aplicación, qué no y cómo operar para conseguirlo.

Los procesos de manipulación y de retroalimentación son vitales para construir el modelo mental del usuario. Por tanto, el diseñador de la interfaz de usuario debe intentar descubrir la imagen mental que el usuario tiene del mundo real para intentar reproducirla en la interfaz de la aplicación. La consistencia

Figura 3.8: La calculadora de Windows reproduce la imagen mental de lo que un usuario espera de una calculadora.

entre la imagen mental del usuario y la interfaz de la aplicación es la clave para el éxito y correcto uso de la aplicación.

La calculadora proporcionada como accesorio en el entorno Windows (véase Figura 3.8) es un buen ejemplo de una aplicación que cumple con este principio. La interfaz de usuario trata de mimetizar tanto estéticamente como el comportamiento de un artefacto que el usuario ya ha tenido la ocasión de manipular en la vida real. Esta forma de mimetismo se aprovecha del conocimiento previo del usuario de la vida real para interaccionar un objeto virtual que lo simula.

Principios comunes a toda buena guía de estilo

El principio de alto nivel: *Reproducir la imagen mental del usuario*, puede ser refinado y/o complementado en base a otros principios más específicos. A continuación se describen.

1. **Consistencia**
   La consistencia implica un conjunto de convenciones que han de ser seguidas dentro de la aplicación (intra-aplicación) y entre las aplicaciones (inter-aplicaciones). Los objetos con apariencia o comportamiento similar son interpretados de manera consistente. La consistencia temporal (o estabilidad) también es recomendada por la mayoría de las guías de estilo. La mayoría alcanza este grado de consistencia mediante la no-modalidad.[10]

[10] El concepto de no-modalidad se define en el apartado 3.2.2, *página 66*.

Los beneficios aportados por interfaces consistentes pueden resumirse en los siguientes: sistemas más predecibles, mejora de la curva de aprendizaje, uso de distintas aplicaciones sin instrucciones adicionales.

2. **Familiaridad**
   Las metáforas del mundo real [IBM92], el lenguaje orientado al usuario [Computer87], las metáforas visuales (iconos), etc. aportan familiaridad al usuario. El principio de la familiaridad puede ser vista como la consistencia con el mundo real. Éste tipo de mimetismo, ayuda a los usuarios a usar su conocimiento del mundo real para interactuar con el sistema, simplificando el proceso de aprendizaje ya que se apoya en conceptos del mundo real que el usuario ya ha experimentado antes.

3. **Simplicidad**
   Cuanto más simple mejor. El usuario no debe sentirse desbordado en ningun momento por los detalles. Cuanto más simple es la interfaz, más fácil es que el modelo mental del usuario coincida con el de la aplicación. La simplicidad facilita mucho el aprendizaje del sistema. Sin embargo, la simplicidad puede ser un lastre para los usuarios experimentados que dificulta otro principio, a veces contrapuesto como puede ser el principio de la eficiencia. En estos casos, hay que alcanzar un equilibrio entre ambos principios.

4. **Control en manos del usuario**
   Engloba dos conceptos:

   - **Manipulación directa:** Sobre todo, en entornos WIMP *(Windows, Icon, Menu, Pointer)*, la manipulación directa tiene un peso especifico. En estos entornos, los usuarios están acostumbrados a identificar objetos y herramientas de manera visual. Los objetos son manipulados por medio de las herramientas. Se intenta que el usuario tenga el control del sistema en lugar de que el sistema indique que acción es la siguiente a tomar.
   - **Flexibilidad:** Las interfaces deben adaptarse a las necesidades del usuario. La personalización del sistema permite ajustar el sistema para un usuario especifico con unas determinadas necesidades. La mayoría de guías de estilo sugieren que debería proporcionarse más de una manera de llevar a cabo una misma tarea. CUA [IBM92] va más allá, y requiere que cualquier tarea que sea posible realizarse mediante el uso del ratón, ha de ser también realizable mediante el teclado.

5. **Retroalimentación**
   Todas las acciones del usuario han de tener como respuesta una rápida

retroalimentación que sea consistente y entendible para el usuario. De este modo, los usuarios permanecen centrados en su tarea y sienten tener el sistema bajo su control. Técnicas de interacción como WYSIWYG *(What You See Is What You Get)* [Computer87] hacen un uso intensivo de este principio.

6. **Aproximación gradual**
   Cuando tomamos el automóvil para viajar, p.e. desde Valencia a Chinchilla de Montearagón,[11] no esperamos encontrar miles de carteles que nos indiquen la dirección a tomar para cada pueblo de España. Al contrario, uno espera ver carteles indicadores *génericos* con los nombres del las ciudades más importantes: Madrid, Castellón, Alicante, Albacete. Esto limita mucho nuesta posibilidad de elección y/o error. Esta vez decidimos, con buen criterio, tomar rumbo hacia Albacete. Conforme nos vamos acercamos a Albacete, nos encontramos con carteles cada vez más *especificos*: Almansa, Bonete y finalmente Chinchilla de Montearagón.

   Al igual que en el ejemplo de los carteles de información de las carreteras, las interfaces de usuario deben organizar y presentar la información al usuario de manera gradual. Siguiendo la regla nemotécnica de 7 ± 2 elementos, debemos cuidar no abrumar al usuario cuando deba tomar una decision con un número excesivo de opciones ya que el número de elementos que los humanos podemos recordar en la memoria inmediata está en torno a nueve elementos.

   La agrupación o clasificación en grupos y subgrupos se convierte en necesidad cuando el número de opciones crece. Los menús de aplicaciones desplegables o los controles arborescentes son un buen ejemplo del *Principio de Aproximación Gradual* [Barfield93].

7. **Diseño visual**
   El diseño visual engloba la posición y la apariencia de los objetos. Las guías de estilo favorecen la simplicidad y la claridad en detrimento de diseños recargados o barrocos con vistosos efectos visuales. De este modo, la atención recae sobre la tarea que el usuario realiza evitando despistes innecesarios sobre los detalles y adornos de la interfaz.

8. **Robustez**
   Los usuarios son propensos a cometer errores, unas veces por la propia naturaleza humana y otras por que el usuario prefiere aprender mediante el método de prueba y error antes que recurrir a leer el manual. Por tanto, se espera que los sistemas estén diseñados para prevenir los errores:

   - guiando a los usuarios,

[11] Ciudad histórica «*muy noble y muy leal*» situada en la provincia de Albacete.

- siendo flexibles en sus expectativas sobre el comportamiento de los usuarios y
- proporcionando avisos que alerten al usuario acerca de las acciones peligrosas.

En el caso de que se produzcan acciones indeseadas, la posibilidad de deshacer las últimas acciones llevadas a cabo *(Undo)* puede enmendar gran parte de los errores cometidos. Los usuarios valoran mucho las prestaciones de tipo deshacer puesto que les proporciona sensación de seguridad el hecho de saber que si comenten un error pueden enmendarlo con facilidad.

9. **Eficiencia**
   La eficiencia mide cuan rápido y preciso un usuario puede realizar las tareas en el sistema. Un buen diseño de un sistema debe realizar un análisis de tareas para favorecer el rápido desarrollo de aquellas tareas más frecuentes e importantes. La eficiencia se consigue con la ayuda de otros principios como la consistencia, simplicidad, familiaridad y el diseño visual. La eficiencia redunda en una mejor productividad de los usuarios usando el sistema. En éste sentido, cabe reseñar que los usuario experimentados esperan características avanzadas en el sistema, pero sin embargo, el sistema no debe ser a la vez, demasiado complejo para los usuarios noveles.

Un recorrido rápido sobre las interfaces de usuario acaba de ser presentado en esta sección. Las propiedades deseables y principios de construcción han sido también detallados. En la medida en la que el diseño de la interfaz de usuario va ha ser derivado a partir de un modelo conceptual, su conocimiento es torna imprescidible para comprobar si en el resultado final, dichas propiedades están presentes o no.

3.3. Patrones

En marco del desarrollo del software, los patrones son un tema candente donde se encuentran trabajando multitud de grupos de investigación en todo el mundo. Dentro de la Ingeniería del Software, los patrones emergieron en primer lugar dentro de la comunidad orientada a objetos. Esta nueva forma de ingeniería trata de buscar en la realidad problemas comunes para abstraer la esencia del problema y de la solución. De este modo, se consigue obtener un catalogo de problemas-soluciones donde cada problema tiene un nombre y posibles soluciones. Aunque los problemas tengan difícil solución, el simple

hecho de documentarlos y darles nombre permite a la comunidad científica dotarse de una jerga común donde todos los que se refieren a un mismo problema usan el mismo nombre para referirse a él.

El proceso de identificación de patrones requiere de grandes dosis de observación para encontrar semejanzas y de abstracción para descartar detalles irrelevantes y describir con precisión la esencia del problema. De este modo, el proceso de identificación de patrones puede verse como un tipo de ciencia empírica y de abstracción donde a partir de los resultados se intenta obtener leyes que rijan dichos resultados.

El uso de patrones, supone una ciencia de carácter estadístico y empírico; puesto que los patrones, por sí mismos y de modo aislado, no resuelven todos los problemas posibles. Sin embargo, una colección de patrones cuidadosamente escogidos puede caracterizar del orden de 80 % de los pares problemas/soluciones para un dominio dado. Es más, la decisión de la incorporación o no de un nuevo patrón a un lenguaje de patrones puede verse justificada por la frecuencia/infrecuencia de aparición del problema descrito por dicho patrón.

3.3.1. Origen

Los patrones dentro de la comunidad informática comenzaron a tomar fuerza a partir de la relevancia obtenida por el libro «*Design Patterns: Elements of Reusable Object-Oriented Software*» [Gamma95] de Eric Gamma, Richard Helm, Ralph Jonson y John Vlissides (frecuentemente citados como *the Gang of Four* (la banda de los cuatro) o más frecuente todavía, con su acrónimo *GoF*). Éste libro recopila una serie de patrones de diseño orientados a objetos que ayudan a la construcción y posterior reuso de componentes de software.

Sin embargo, el origen de los patrones es previo. El uso del termino actual *patrón* fue descrito por el arquitecto Christopher Alexander que escribió varios libros sobre planificación urbana y construcción de edificaciones: *Notes on the Synthesis of Form* [Alexander64], *The Oregon Experiment* [Alexander75], *A Pattern Language: Towns, Buildings, Construction* [Alexander77] y *The Timeless Way of Building* [Alexander79].

Aunque los textos de Alexander tratan sobre arquitectura, las ideas que en ellos se intentan captar pueden ser aplicadas a otras disciplinas, en particular, a la arquitectura y construcción de programas.

Alexander argumenta que los métodos arquitectónicos actuales conducen a productos que no satisfacen las necesidades reales ni los requisitos de los usuarios. No cumplen con el propósito final de todo diseño o esfuerzo de ingeniería: mejorar la condición humana.

Alexander pretende crear estructuras que sean adecuadas para las personas, mejorando su confort y calidad de vida. En *The Timeless Way of Building*, Alexander trata de captar ideas de diseño atemporales *(timeless)* para tratar de alcanzar estos objetivos. El paradigma para arquitectura propuesto por Alexander se basa en tres conceptos:

1. **La cualidad** *(the Quality Without a Name)* Es la esencia de las cosas útiles que imprime en ellas cualidades como: libertad, completitud, confort, armonía, habitabilidad, durabilidad, apertura, resistencia, variabilidad, adaptabilidad. Es lo que nos hace sentir satisfechos de un producto o diseño y finalmente mejora la condición humana.

2. **La puerta** *(the Gate)* Es el mecanismo que permite alcanzar *la cualidad*. Se manifiesta como un lenguaje de patrones común que permite crear múltiples diseños para satisfacer diferentes necesidades. Es el universo etéreo de los patrones y sus relaciones permitidas en un dominio dado. *La puerta* conduce a *la cualidad*.

3. **El camino** *(the Timeless Way)* Usando *el camino*, los patrones definidos en *la puerta* se aplican y se combinan progresivamente para obtener diseños que poseen *la cualidad*.

Por tanto, *el camino*, en su recorrido, atraviesa *la puerta* para alcanzar *la cualidad*.

3.3.2. Definición

Una vez introducido el concepto de patrón, merece la pena revisar algunas definiciones del concepto de diversos autores para contrastar diferentes puntos de vista.

Rhiele y Zullighoven [Riehle96] proponen una definición de patrón como sigue:

> *«Un patrón es la abstracción de una forma concreta que se mantiene en contextos específicos y no arbitrarios.»*
>
> — Rhiele y Zullighoven, 1996.

Los autores de esta definición apuntillan que, dentro de la comunidad informática, la noción de patrón es *guiada hacia la resolución de problemas*. Es decir, la forma concreta que se repite como solución a un problema recurrente.

Una definición más compacta que refleja este contexto es la de Appleton [Appleton97]:

> *«Un patrón es una semilla nominada de información instructiva que captura la estructura esencial y la perspicacia de una familia de soluciones probadas a un problema recurrente que surge dentro de un cierto contexto entre intereses o fuerzas contrapuestas.»*
>
> — Appleton, 1997.

Los patrones también son concebidos como tipos de arquitecturas u organización de partes constituyentes para producir entidades más complejas. En este sentido, una tercera definición de patrón podría ser la propuesta por el mismo Alexander [Alexander79]:

> *« Cada patrón es una regla compuesta por tres partes, que expresan una relación entre un cierto contexto, un problema y una solución.*
>
> - *Como elemento en el mundo, cada patrón es una relación entre un cierto contexto, un cierto sistema de fuerzas que ocurren repetidamente en ese contexto, y una cierta configuración espacial que permite resolver las fuerzas por sí mismas.*
> - *Como elemento del lenguaje, un patrón es una instrucción que muestra como la configuración espacial debe ser usada, una y otra vez, para resolver el sistema de fuerzas dado siempre que el contexto sea relevante.*
>
> *El patrón es, en resumen, al mismo tiempo una entidad, que ocurre en el mundo, y una regla que nos dice como crear esa entidad, y cuando debemos crearla. Es a la vez un proceso y una entidad. La descripción de una entidad que está viva, y la descripción del proceso que genera esta entidad.»*
>
> — Christophe Alexander, arquitecto.

Por último, Coplein [Coplein96] propone un símil de la definición previa respecto a los patrones de costura:

> *«Podemos explicar como hacer un vestido especificando la ruta de las tijeras sobre la tela en términos de ángulos y longitudes de corte. O bien, podemos dar un patrón. Leyendo la especificación de corte, nadie tendría idea acerca de que se está construyendo, al menos hasta que esté construido. Sin embargo, el patrón presagia el resultado: es la regla para crear el vestido, pero también, en gran medida, es el vestido en sí mismo.»*
>
> — James Coplien, 1996.

Como se ha podido comprobar a través de la las definiciones previas, el patrón involucra una descripción general de una solución recurrente a un problema recurrente que tiene objetivos y restricciones (fuerzas) definidos. Además de todo esto, un patrón hace mucho más que identificar una solución, también justifica por qué la solución es necesaria.

3.3.3. Formatos

A la hora de describir patrones se han usado diversos formatos de especificación. El formato alejandrino (también denominado formato canónico) es empleado por Alexander en sus trabajos. Otros textos siguen el formato definido por GoF [Gamma95], que se denomina *GoF format*. Ambos se diferencian en la inclusión de algunas secciones extra destinadas a facilitar la compresión del patrón.

A continuación se describirán las secciones que debe contemplar la especificación de un patrón según el formato alejandrino.

Nombre: Debe ser un nombre significativo. Corto, una palabra simple o una frase breve que permita referir el patrón de manera no ambigua. Si en la literatura se utilizan distintos nombres para un mismo patrón, debe hacerse constar otros nombres empleados para el mismo concepto.

Problema: Una frase o párrafo que describa su intención: las metas y objetivos (requisitos) que se persiguen dentro del contexto y fuerzas. A menudo, las fuerzas se oponen a los objetivos o los objetivos se excluyen mutuamente de algún modo.

Contexto: Las precondiciones bajo las cuales el problema y su solución son recurrentes y por tales precondiciones la solución es la deseable. El contexto informa acerca de la *aplicabilidad* del patrón. El contexto puede ser visto como la configuración inicial del sistema antes de que el patrón le sea aplicado.

Fuerzas: Una descripción de las fuerzas relevantes y restricciones y como interaccionan o entran en conflicto entre sí y con los objetivos que se persiguen. Las fuerzas revelan la complejidad del problema y define los tipos de *intercambios* que deben ser considerados en la presencia de tensiones o disonancias creadas por las fuerzas. Una buena descripción de un patrón debe reflejar todas las fuerzas que tienen impacto sobre el patrón.

Solución: Relaciones estáticas y reglas dinámicas que describen como alcanzar el resultado deseado. A menudo consiste en la descripción de una

serie de pasos que permiten construir el producto necesitado. La estructura estática describe la forma y organización del patrón, mientras que el comportamiento dinámico es el que da la semántica al resultado.

Ejemplos: Consiste en uno o varios ejemplos de aplicaciones del patrón que ilustran: un contexto inicial, cómo se aplica el patrón y cómo transforma el contexto. Los ejemplos ayudan a entender el uso del patrón y su aplicabilidad.

Contexto resultante: Describe el estado o configuración del sistema una vez el patrón ha sido aplicado incluyendo las consecuencias (pros y contras) derivados de la aplicación del patrón. Incluye los problemas y patrones que pudieran aparecer dentro de ese nuevo contexto y describe las postcondiciones y efectos laterales del patrón. A este apartado a menudo se le denomina *resolución de fuerzas* debido a que describe qué fuerzas han sido resueltas, cuáles permanecen sin resolver y qué patrones son nuevamente aplicables en el estado actual.

Fundamento: Una explicación que justifique los pasos o reglas descritos en el patrón. Debe explicar como las fuerzas y restricciones son orquestadas para conseguir los requisitos deseados. Es la justificación de utilidad del patrón: explica cómo funciona, por qué funciona y por qué es una *buena solución*.

Patrones relacionados: La relación con otros patrones si la hubiera. Los patrones dentro de un mismo lenguaje de patrones suelen tener fuerzas compartidas.

Usos conocidos: Describe ocurrencias conocidas del patrón y su aplicación dentro de sistemas existentes. Esta ayuda valida un patrón verificando que efectivamente es una *solución probada* a un *problema recurrente*.

3.3.4. Cualidades de un patrón

Un patrón bien descrito debe exhibir las siguientes cualidades deseables según Lea [Lea99]:

Encapsulación y abstracción: Cada patrón encapsula un problema bien definido y su solución en un dominio particular. Los patrones deben proporcionar fronteras claras que ayuden a separar claramente el espacio del problema del espacio de la solución parcelando el patrón en fragmentos interconectados.

Apertura y Variabilidad: Los patrones deberían ser abiertos para soportar extensiones o parametrización por parte de otros patrones de modo que puedan trabajar juntos para resolver problemas mayores. La solución de un patrón debe ser capaz de ser realizada mediante diferentes implementaciones.

Generabilidad y Composicionabilidad: Un patrón una vez aplicado genera un contexto resultante que encaja con el contexto inicial de uno o más patrones en un lenguaje de patrones. Los subsiguientes patrones se aplican para, poco a poco, alcanzar la solución completa.

> «*Los patrones se aplican de modo incremental* (piecemeal growth). *La aplicación de un patrón proporciona un contexto inicial para la aplicación del siguiente patrón.*»
>
> — Extraído del *Call for Papers* del congreso PLoP'97[12].

Equilibrio: Por último, cada patrón debe realizar algún tipo de balanceo entre las fuerzas y restricciones involucradas. Puede deberse a que hay invariantes o heurísticos que son empleados para minimizar el conflicto en el espacio de la solución. Los invariantes a menudo tipifican problemas fundamentales que pueden ser resueltos mediante un principio de resolución para el dominio particular.

Por otro lado, Winn y Calder [Winn02] han propuesto una lista de propiedades esenciales que son observables en un patrón. Si bien esta lista de propiedades no garantizan la condición de «*ser*», la usencia de ellas proporcionan indicios claros de su «*desclasificación*» como patrón.

1. **Un patrón implica un artefacto.** El patrón presagiar la forma del resultado del mismo modo que el patrón de costura presagia la prenda [Coplein96].

2. **Un patrón cruza varios niveles de abstracción.** Un patrón no es ni un diseño concreto, ni una idea totalmente abstracta. Al contrario, facilita la progresión de un nivel de abstracción al siguiente.

3. **Un patrón es a la vez funcional y no funcional.** Los temas funcionales tratan la posibilidad, determinan las decisiones en un determinado contexto. Los temas no-funcionales tratan la viabilidad: las decisiones deseables en contextos particulares. Un patrón incluye ambos tipos de consideraciones.

[12] PLoP: Proceedings of Language of Patterns.

4. **Un patrón es una solución manifiesta.** Allí donde un patrón haya sido usado consciente o inconscientemente para resolver un problema, el patrón estará presente y será reconocible en la solución diseñada.

5. **Un patrón captura los puntos calientes de un sistema.** Los patrones facilitan un buen diseño capturando lo que Wolfgang Pree denomina los puntos calientes de un sistema (*system "hot spots"* [Pree95]). Los elemento clave son aquellas partes de una solución que pueden cambiar conforme el sistema evoluciona. Los patrones capturan los invariantes y los puntos calientes y proporciona una estructura para manejar la interacción entre la parte estable y los elementos cambiantes del sistema.

6. **Un patrón es parte de un lenguaje.** Cada patrón aparece conectado con otros patrones. Los patrones forman parte de redes de patrones interrelacionados. El mismo Alexander los concibió en origen de este modo [Alexander77].

7. **Un patrón es validado por su uso.** Normalmente, los patrones son descubiertos a través de experiencias concretas más frecuentemente que a partir de disquisiciones abstractas. Sin embargo, un patrón no puede ser verificado o validado desde un planteamiento solo teórico. En el extremo, la demostración de la existencia del patrón recae en la aparición recurrente e identificable del patrón en soluciones construidas.

8. **Un patrón está vinculado a un dominio.** Un patrón no es una unidad aislada. Si no que es definido el un contexto con otros patrones (lenguajes de patrones). Un patrón aplicado fuera del dominio inicial puede no ser viable.

9. **Un patrón captura una gran idea.** Los patrones no son soluciones a problemas triviales. Cada solución a un problema particular no garantiza la existencia de un patrón.

En resumen, un patrón debe proporcionar un equilibrio. Debe proporcionar una solución especifica a un problema específico.

3.3.5. Lenguajes de patrones

Un lenguaje de patrones es un conjunto de patrones que comparten un mismo ámbito o contexto, es decir, comparten un mismo dominio y además están relacionados entre si. Un lenguaje de patrones puede servir de base para un marco de trabajo o *framework*. Este marco de trabajo es reutilizable y puede ser aplicado para resolver problemas dentro de ese ámbito bien definido.

Los patrones pueden clasificarse de acuerdo al nivel de abstracción al que pertenecen. De acuerdo a esta clasificación y dentro del ámbito de sistemas de información tenemos:

1. **Patrones arquitectónicos:** Los patrones arquitectónicos expresan estructuras de organización de sistemas de información. Proporcionan un conjunto de subsistemas predefinidos donde se especifican sus responsabilidades y se incluyen reglas y guías para organizar las relaciones entre los subsistemas.

   Una arquitectura tres capas o cliente/servidor son ejemplos de patrones arquitectónicos.

2. **Patrones de diseño:** Un patrón de diseño proporciona un esquema para refinar componentes o subsistemas de un sistema de información o las relaciones entre ellos. Describe una estructura recurrente de componentes que se comunican que resuelve un problema general de diseño en un contexto particular.

   Aunque los patrones de diseño suelen proporcionar ejemplos sobre un lenguaje de programación como C++ o Java, son independientes de cualquier lenguaje de programación.

   Los patrones están la punza de lanza de la investigación para sistemas de información. Desde la publicación de *Design Patterns* [Gamma95] muchos otros libros, talleres (*Workshops*) y conferencias se han realizado acerca de los patrones de diseño.

   Para cualquier diseñador de sistemas, los patrones de diseño son una fuente de conocimiento que no debe ser despreciada, ya que contienen el conocimiento y experiencia de otros diseñadores de sistemas sobre problemas que aparecen una y otra vez.

3. **Patrones de programación, Idioms:** Un patrón de programación o *idiom* es un patrón de bajo nivel, muy especifico para un lenguaje de programación dado [Coplein98]. Describe como implementar aspectos particulares de componentes o sus relaciones entre ellos usando las características particular de un lenguaje de programación dado. Es decir, un *idiom* no puede ser reusado en otro lenguaje.

Otra clasificación de los patrones propuesta por Riehle [Riehle96] da lugar a:

1. **Patrones conceptuales:** Un patrón conceptual es un patrón que se describe en términos de y conceptos de un dominio de aplicación (dominio del problema).

2. **Patrones de diseño:** Un patrón de diseño es un patrón descrito en términos de construcciones de diseño como por ejemplo, objetos, clases, herencia, agregación y relaciones de uso.

3. **Patrones de programación:** Un patrón de programación está descrito en términos de construcciones de un lenguaje de programación.

3.3.6. Uso de los patrones en esta tesis

Una vez establecido el concepto de patrón y sus posibles clasificaciones, podemos encuadrar a los patrones que se van a ilustrar en la presente tesis como patrones conceptuales de interfaz de usuario que serás descritos con profusión en el capítulo 4.

1. Son **patrones conceptuales** porque los patrones están descritos en términos del espacio del problema orientados a la especificación, independientemente del diseño que pueda llevarse a cabo.

2. Son **patrones de interfaz de usuario** porque los patrones se enmarcan dentro de un ámbito o contexto bien definido, la interfaz de usuario.

De este modo, el lenguaje de patrones que se propone en el capítulo 4 sirve como base para desarrollar el modelo para la especificación de interfaces de usuario (Just-IU). En particular, OO-Method [Pastor97] ha sido extendido con Just-IU iniciado en [Molina98] y continuado dentro del grupo de trabajo de OO-Method y CARE Technologies S.A. para enriquecer su expresividad de modelado y propiciar la consecución de generación automática de interfaces de usuario.

3.3.7. Patrones: Panorámica general

Durante los últimos años, los talleres (*Workshops*) organizados a lo largo de sucesivos años en el marco de la conferencia CHI (*Computer Human Interaction*) organizados por ACM/SIGCHI (grupo de interes en Interación Persona Ordenador de la *Association for Computer Machinery*) han sido el principal foro para la discusión acerca de la aplicación de los patrones al desarrollo de interfaces de usuario:

- En el año 1997, Thomas Erickson y John Thomas sentaron las bases para la búsqueda de un lenguaje de patrones para la interacción [Erikson97].[13]

[13] Resumen disponible en `http://www.pliant.org/personal/Tom_Erickson/Patterns.WrkShpRep.html`

- En la edición del año 2000, Richard Griffits, Lyn Pemberton, Jan Borchers y Adam Stork organizaron un segundo taller: «*Pattern Languages for Interaction Design: Building Momentum*» [Griffits00].[14]

- En la edición del año 2002, Martijn van Welie, Kevin Mullet y Paul McInerney organizaron otro taller orientado a patrones [Welie02].[15]

- Durante la edición de 2003 se va ha celebrar otro taller orientado hacia herramientas de soporte a patrones: «*Perspectives on HCI patterns: concepts and tools*» organizado por Sally Fincher, Janet Finlay, Sharon Greene, Lauretta Jones, Paul Matchen, John Thomas y el propio autor de esta tesis [Fincher03].[16]

Como resultado de estos talleres, se aprecia un interés creciente en el campo de los patrones software aplicados al area de la construcción de interfaces de usuario.

Dado el diferente bagaje de los asistentes a estos talleres, se han obtenido una visión enriquecedora al contrastar el uso que se esta haciendo de a los patrones desde diferentes puntos de vista como por ejemplo: psicología cognoscitiva, aprendizaje, como herramienta de diseño, como lenguaje común para los desarrolladores y usuarios, como soporte a la especificación y generación de código a partir de modelos.

Colecciones existentes

Dependiendo de su dominio y nivel de abstracción, podemos situar las diferentes colecciones y lenguajes de patrones. En particular, la Figura 3.9 muestra una clasificación teniendo en cuenta las fases de desarrollo y al grado de abstracción empleado. Los *Patrones de Diseño* [Gamma95] son patrones de propósito general (no necesariamente interfaces de usuario) que pueden ser aplicados para resolver problemas de diseño. Por contra, los *Patrones de Análisis* [Fowler96] son considerados en las fases de requisitos y especificación. Fowler proporciona diversos patrones para diferentes dominios como son la banca, la sanidad o la contabilidad. Dentro del mundo de las interfaces de usuario, las colecciones de patrones suelen enfocarse a la fase de diseño [Tidwell99, Welie00a, Trætteberg00, Trætteberg02, Javahery02].

La aproximación presentada en esta tesis, propone:

[14] Resumen disponible en http://www.stanford.edu/ borchers/hcipatterns/chi2000ws.html y www.acm.org/sigchi/bulletin/2000.1/borchers.pdf

[15] Resumen disponible en http://www.welie.com/patterns/chi2002-workshop/

[16] Solicitud de contribuciones en http://www.dsic.upv.es/~pjmolina/CHI2003WS/

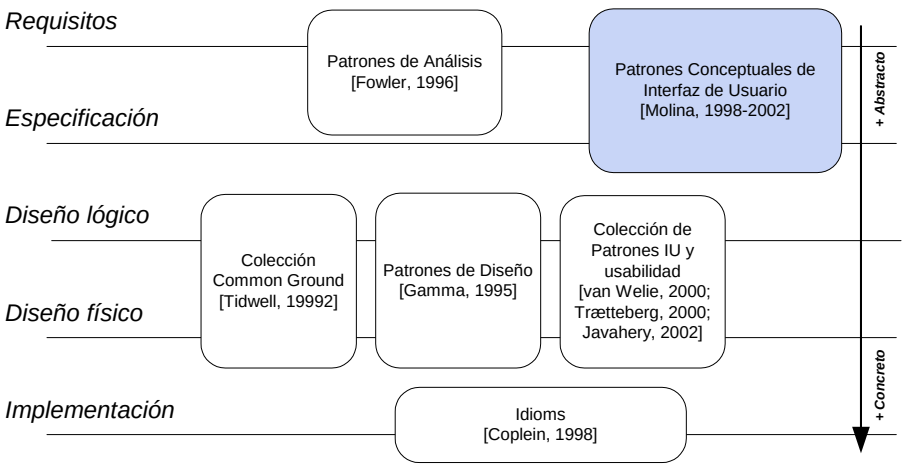


Figura 3.9: Panorámica de patrones según niveles de aplicación.

- Just-IU [Molina02d]: como lenguaje de patrones de interfaz de usuario a nivel conceptual.
- Cada patrón identificado a nivel conceptual puede tener asociadas más de una reificación a nivel de diseño lógico, físico y finalmente de implementación [Molina02b, Molina02c]. Asociando a cada patrón una colección de implementaciones, es posible generar código para diversas plataformas.

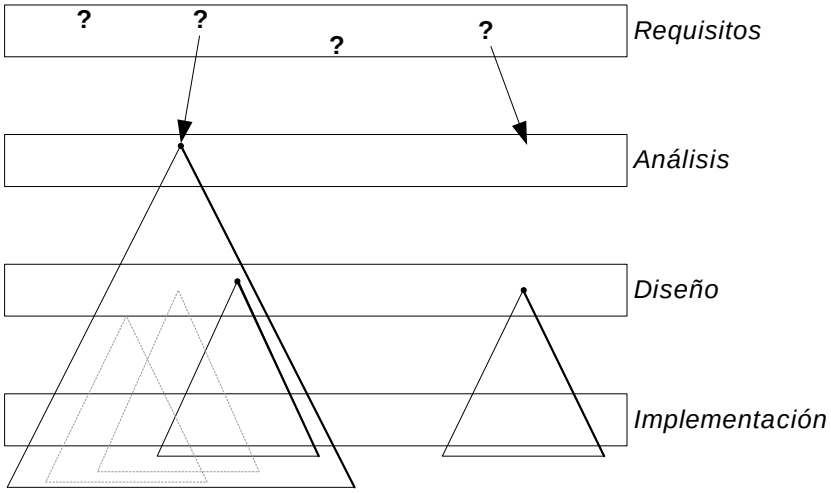


Figura 3.10: Efecto Δ (Delta).

3.3.8. Reificación y el Efecto Δ (Delta)

Cuanto más abstracto es un patrón, más amplio es el posible campo de aplicación. Al mismo tiempo, el patrón puede ser instanciado para resolver el problema con múltiples formatos. A raíz de esta observación, las siguientes preguntas surgen de modo natural en fases de diseño:

- *¿Qué implicaciones impone la elección de un determinado patrón en la fase de análisis?*
- O más específicamente: *¿Cómo los patrones de conceptuales aplicados en la fase de análisis influyen en las fases siguientes?*

Un patrón proporciona restricciones semánticas, de comportamiento y estructurales. Dichas restricciones tienen un alcance cónico o Efecto Δ (Delta) [Molina02b, Molina02c] (*véase Figura 3.10*): la elección tiene implicaciones desde su definición hasta los refinamientos en las sucesivas fases. Estos refinamientos están por tanto supeditados a las decisiones tomadas en fases precedentes.

Los símbolos de interrogación mostrados en la Figura 3.10 representan preguntas o requisitos formulados en la fase de requisitos. Dichas preguntas son respondidas por medio de la elección de un patrón conceptual en la fase de análisis. La Figura 3.10 muestra el alcance de las decisiones en forma de triángulos Δ donde aspectos de subsiguientes fases están restringidas o guiadas por dichas decisiones previas.

En este sentido, decisiones tomadas en las primeras fases (selección de patrones conceptuales en la fase de análisis) pueden guiar o ayudar a tomar decisiones en términos de diseño (donde los patrones de diseño pueden ser aplicados). Una herramienta de tipo asistente (*wizard*) puede ayudar a seleccionar los patrones en la fase de diseño a partir de la información capturada en las fases previas.

Por ejemplo: un asistente puede sugerir la reificación del Patrón *Maestro/Detalle* [Molina02d] en la fase de diseño mediante los patrones de diseño *Container Navigation* [Trætteberg00, Trætteberg02] o *Navigation Spaces* [Welie00a, Welie00b].

De este modo, los patrones pueden ser resueltos en cascada. Los patrones conceptuales pueden ser implementados usando patrones de diseño y estos, a su vez, aplicando *idioms*.

El Efecto Δ (Delta) así observado, permite restringir y guiar la selección de opciones de diseño en una fase de construcción en función de las elecciones tomadas en una fase previa. De este modo, se puede dar soporte a un refinado guiado de las especificaciones hasta la implementación final.

En esta sección se ha introducido el concepto de patrón y se ha presentado un recorrido por sus aplicaciones en el campo de la Ingeniería del Software. En la presente tesis se pretende explorar el uso de los patrones de interfaz de usuario a nivel conceptual como herramientas centrales para la especificación de interfaces de usuario abstractas.

3.4. Tecnologías de Generación de Código

Por último, y como cuarto pilar de esta tesis, se presenta a continuación un resumen a las técnicas y métodos de generación de código basado en modelos.

La generación de código proporciona sistemas de reificación automática que en líneas general permiten ahorrar grandes cantidades de recursos: esfuerzo, tiempo y dinero.

La presente sección comienza presentando las ventajas e inconvenientes de la generación de código. A continuación se describe FAST como una de las metodologías para abordar sistemas de producción de aplicaciones basado en generación de código. Por último se revisan los tipos de generación de código existentes en el mercado para la producción de aplicaciones a partir de tecnología orientada a objetos.

3.4.1. Ventajas de la generación de código

Las principales razones para usar técnicas de generación de código son [Cleaveland01]:

- **Productividad.** Incrementa considerablemente la productividad. El código generado se produce en segundos o minutos.
- **Separación de responsabilidades.** Permite una separación clara del *qué* del *cómo* (*Separate of Concerns*).
- **Calidad.** Reduce el número de errores introducidos en la codificación. El proceso es repetible y controlable (auditable). Se pueden imponer reglas de estilo al código generado para aumentar su homogeneidad y legibilidad.
- **Corrección.** Corrección del producto generado y robustez. Es más fácil controlar la corrección del código generado (lo que se ha generado correctamente, volverá a ser generado correctamente si no se han introducido cambios en el generador).

- **Mantenimiento centralizado.** Permite mantener sincronizadas las especificaciones respecto a la documentación y las implementaciones. Este proceso a menudo se convierte en el más difícil y caro cuando el software evoluciona. Mediante procesos de regeneración automática, es posible garantizar la consistencia gracias a que el mantenimiento se concentra solo en las especificaciones.

- **Portabilidad.** Independencia de tecnologías de implementación. El trabajo se centra en el análisis. Una nueva tecnología puede ser abordada usando un nuevo generador de código para una nueva plataforma.

Por contra, las desventajas achacables a las técnicas de generación de código son:

- **Dominios reducidos.** La generación de código tradicionalmente se ha aplicado a dominios muy específicos y bien conocidos donde es posible anticiparse a la variabilidad de problemas que pueden encontrarse en ese dominio. Los dominios o áreas de aplicación demasiado grandes o fuera de ámbito no se benefician de la aplicación de técnicas de generación de código.

- **Grado de sofisticación elevado.** El diseño o mantenimiento de un generador de código es una tarea de envergadura que requiere de personal cualificado tanto en el dominio de aplicación como en las herramientas para la construcción del generador.

- **Ajuste final.** (*Round-Trip Problem*) El código generado puede necesitar ser adaptado (*Tweaking* [Bergman02]) (muy frecuentemente en código destinado a interfaces de usuario final) y por tanto cambiado de modo manual. Cuando la especificación cambia, el código es regenerado y los cambios manuales deben volver ser aplicados.

- **Motivos psicológicos.** Hay programadores convencionales que ven la generación de código como una amenaza para su trabajo. Ante lo cual, toman una postura defensiva de rechazo a toda suerte de código generado. Sistemas expertos semi-guiados de generación de código como SEGUIA [Vanderdonckt99a, Vanderdonckt99b] permiten en gran medida paliar este aspecto permitiendo al usuario tomar parte de las decisiones de generación. De este modo, el usuario sigue teniendo la sensación de «*control*» sobre la máquina.

3.4.2. El método FAST

FAST (*Family-oriented Abstraction, Specification & Translation*)[17] [Weiss99] es una metodología desarrollada a principios de la década de los noventa.

Tiene como origen la teoría de Familias de Programas propuesta por Parnas [Parnas76] a mediados de la década de los setenta.

Dado un dominio de trabajo, podemos observar que existen diferentes programas diseñados para ese dominio que comparten muchas similitudes. El estudio de variabilidad y de parte común (que permanece constante) permite caracterizar una familia de programas. En aras a mejorar la productividad, se propone construir especificaciones para dar cuenta de la parte variable; implementar librerías, plantillas, componentes, etc. para la parte común y construir generadores de código para traducir la especificación a código que será enlazado con la parte común para producir la aplicación final.

La metodología FAST define un par de términos que nos serán muy útiles a a hora de describir el método:

Ingeniería de Dominio Se encarga de estudiar el dominio, caracterizarlo. Se construyen modelos, herramientas de especificación, traductores, compiladores, enlazadores, librerías, etc. como herramientas de soporte para definir un proceso de producción de aplicaciones.

Ingeniería de Aplicación Usa el método propuesto por la ingeniería de dominio para producir aplicaciones. Se realizan análisis de requisitos a partir de los cuales se construyen especificaciones y se genera código. Se completa la aplicación y se pone en producción.

FAST establece una serie de actividades bien diferenciadas (*véase Figura 3.11*) así como un árbol de artefactos producidos en cada fase (*véase Figura 3.12*).

[17] FAST: Abstracción, especificación y traducción orientada a familias de programas.

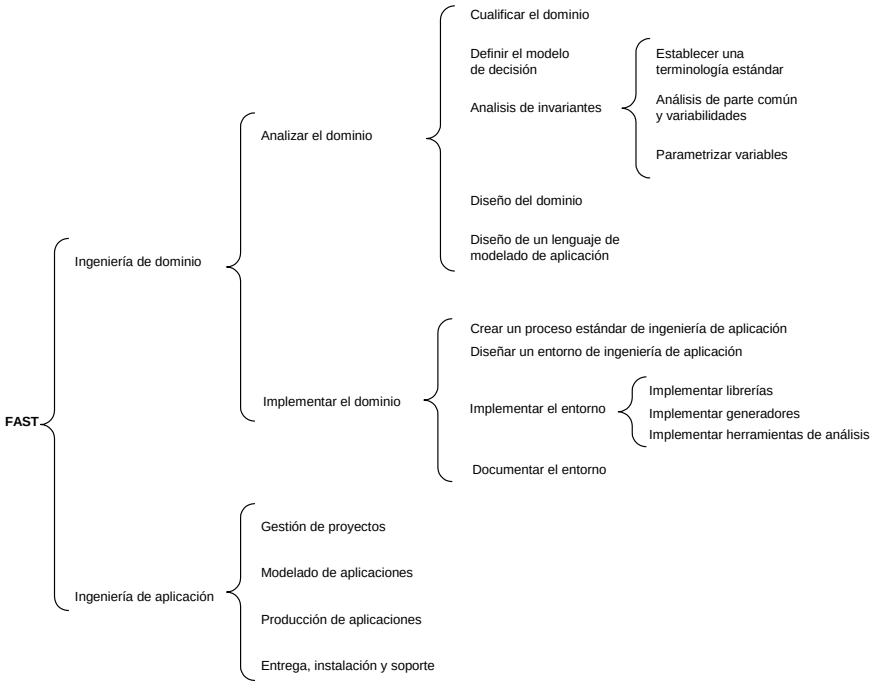


Figura 3.11: Actividades en FAST (adaptado de [Weiss99]).

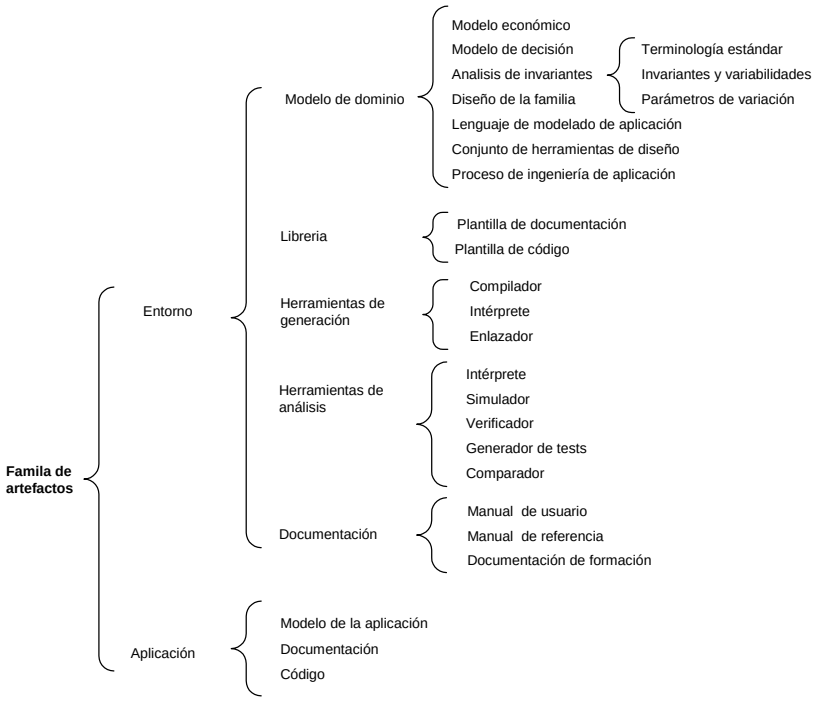


Figura 3.12: Artefactos en FAST (adaptado de [Weiss99]).

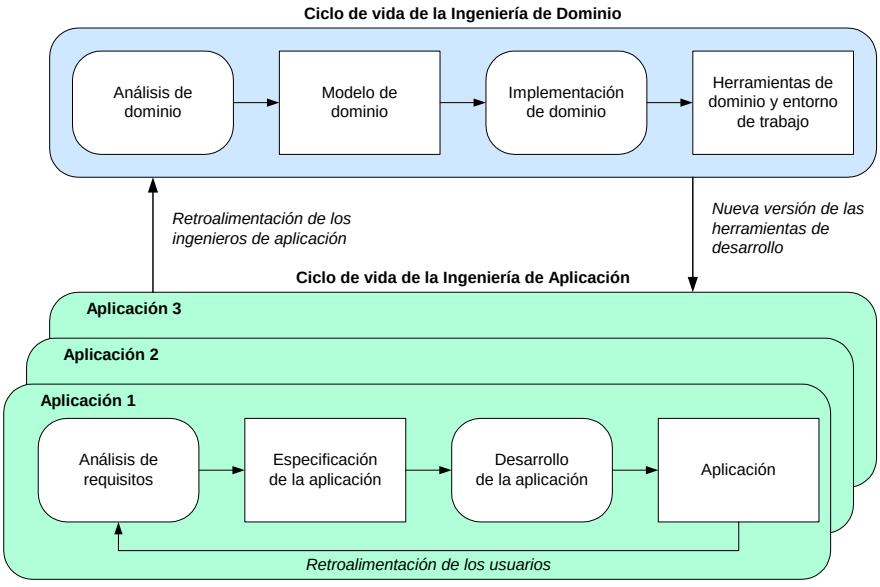


Figura 3.13: Ciclo de vida genérico de un método FAST (adaptado de [Weiss99]).

La Figura 3.13 ilustra el ciclo de vida de desarrollo de software con una metodología de tipo FAST. Los ingenieros de dominio producen nuevas versiones del proceso y de las herramientas de soporte para mejorar y optimizar el proceso. Los ingenieros de aplicación usan el proceso definido y las herramientas para producir aplicaciones. La retroalimentación en el esquema acontece a dos niveles:

- primero los usuario de la aplicación proporcionan retroalimentación a los ingenieros de aplicación,
- en segundo lugar, los ingenieros de aplicación proporcionan retroalimentación a los ingenieros de dominio.

Análisis de invariantes

El análisis de invariantes (*Commonality Analysis*) consiste en estudiar el dominio para caracterizar la familia buscando la parte común (invariantes) en las instancias de la familia y la parte variable, cambiante en cada miembro. Un documento es el producto final de dicho análisis. Dicho documento, contiene la siguiente estructura:

1. **Introducción.** Declaración de objetivos y ámbito.
2. **Descripción general.** Introducción a la familia que se pretende estudiar.
3. **Diccionario de términos.** Recoge de un modo exhaustivo la terminología y los conceptos empleados en el dominio del problema.
4. **Parte común (invariantes).** Relación de aspectos del dominio que permanecen inmutables en los distintos miembros de la familia.
5. **Estudio de Variabilidad.** Relación de aspectos del dominio que varían en los distintos ejemplares de la familia.
6. **Parámetros de variación.** Describe los límites o valores admitidos de variación para cada parámetro.
7. **Temas** (*Issues*). Describe cualquier otro potencial problema que pueda presentarse en el dominio. Justifica las decisiones tomadas acerca de por qué considerar un determinado concepto como inmutable o como variable.
8. **Apéndices.** Cualquier otra documentación para completar la descripción del dominio o el documento.

Como resultado de este estudio se obtiene una definición de una familia de programas donde la parte invariante es *generalizable* (implementable en componentes, librerías, plantillas, etc. una sola vez), mientras que la parte variable es *especificable* (y posteriormente generable).

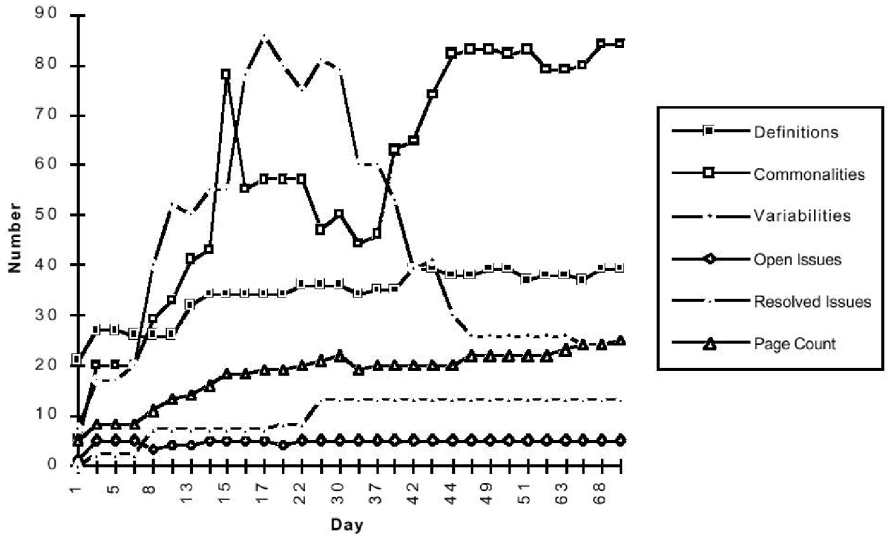


Figura 3.14: Ejemplo de Análisis de invariantes (fuente: [Weiss99]).

La Figura 3.14 muestra un ejemplo de análisis de invariantes llevado a cabo durante setenta dias. En la figura puede apreciase la evolución del documento de análisis de invariantes.

Los conceptos considerados como estables o inmutables crecen a lo largo del tiempo conforme son descubiertos. Por contra, puede apreciarse como las variabilidades crecen muy rápidamente para después volver a reducirse. Conceptos considerados inicialmente como variables son después catalogados como invariantes (*commonalities*). Un menor número de variabilidades redunda en un proceso menos complejo, aunque reduce el tamaño de la familia.

La Figura 3.15 muestra las implicaciones de cambiar la consideración de un concepto como variabilidad o como inmutable. Una nueva variabilidad implica un proceso de *parametrización* que:

- Reduce el código común de la familia.
- Incrementa el tamaño de la familia.
- Incrementa la complejidad del sistema.

Por contra, una nueva consideración como concepto inmutable o fijo conlleva un proceso de *estandarización* que implica:

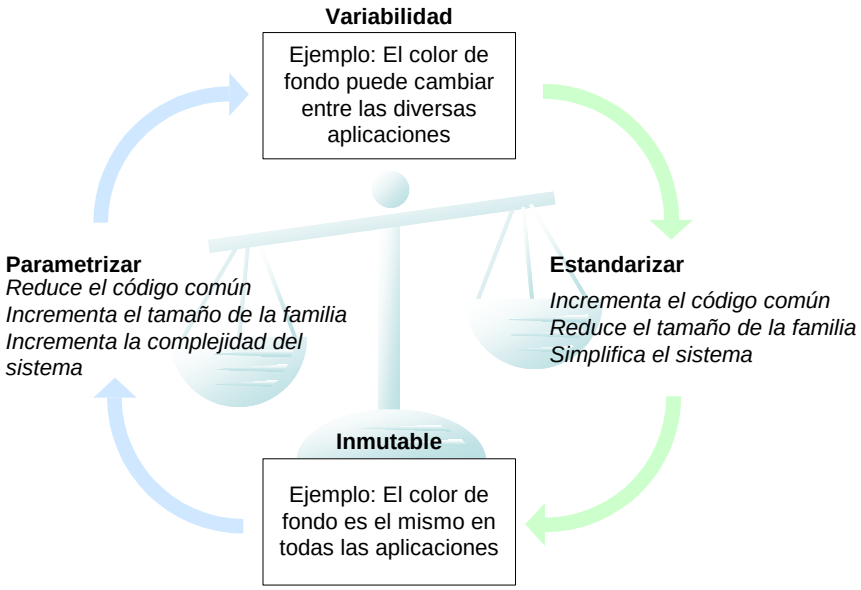


Figura 3.15: Decisión: Variable vs Inmutable (adaptado de [Cleaveland01]).

- Incrementa el código común de la familia.
- Reduce el tamaño de la familia.
- Simplifica el sistema.

Por estos motivos, lo más difícil es establecer el equilibrio entre conceptos inmutables y variabilidades. Hay que dotar al lenguaje de especificación de toda la potencia necesaria para disponer de una familia rica; sin embargo, y al mismo tiempo, hay que mantener el proceso de producción de aplicaciones lo más sencillo posible buscando preservar la facilidad de uso sin perder potencia de expresividad y, por tanto, de generación.

Minimizar los conceptos variables e incrementar en exceso la parte común banaliza la especificación y convierte la familia en un pequeño conjunto de variaciones. Por el contrario, minimizar los conceptos comunes e incrementar en exceso la variabilidad tiene el peligro de disponer de un lenguaje de programación igual de complejo que el empleado en la generación, del cual pretendíamos abstraernos en una capa superior.

Por supuesto, hay que evitar a toda costa este último peligro. Citando a Novak:[18]

[18]Novak: `http://www.cs.utexas.edu/users/novak/index.html`.

> *«Automatic Programming is defined as the synthesis of a program from a specification. If automatic programming is to be useful, the specification must be smaller and easier to write than the program would be if written in a conventional programming language.*[19]*»*
>
> — G. S. Novak

Es decir, si la especificación es más compleja que la programación convencional, nadie usará técnicas de programación automática.

Los criterios para argumentar por qué la programación automática puede ser mejor que la programación tradicional pueden aportarse en términos de coste, esfuerzo, recursos, tiempos de desarrollo, tiempos de entrega (*time to market*), curva de aprendizaje, reuso, tasa de errores, robustez del código, calidad, etc.

En algunos casos, aún a pesar de disponer de especificaciones más complejas de desarrollar que la programación manual equivalente, el hecho de disponer de especificaciones que pueden ser animadas y verificables otorga ventajas adicionales que pueden ser tener más peso para decantar la balanza, por ejemplo, para favorecer su posterior mantenimiento.

Modelo económico de la ingeniería de dominio

Los métodos basados en FAST son adecuados cuando se necesitan producir muchas aplicaciones que forman parte de una familia bien definida. El coste de invertir en un proceso de ingeniería de dominio no siempre está justificado. Por tanto, antes de decidir la aplicación de un método tipo FAST, conviene realizar un análisis económico de coste/beneficio para estimar si realmente merece la pena aplicar el método.

La Figura 3.16 muestra un esquema del ciclo de vida simplificado que pone de relieve que es necesario invertir recursos en ingeniería de dominio. A cambio, el retorno de la inversión llegará en forma de aplicaciones producidas y entregadas a los clientes (venta del producto).

Con el objetivo de analizar con más detalle los costes asociados, definiremos:

- $\overline{C_T}$ como el *coste medio* de desarrollo de aplicaciones por proyecto aplicando alguna metodología *tradicional* (no FAST): $\overline{C_T} = \frac{\sum_i^{1..n} C_{Ti}}{n}$

[19] Traducción: *«La Programación Automática se define como la síntesis de un programa desde una especificación. Para que la programación automática sea útil, la especificación debe ser más pequeña y fácil de escribir que si el programa hubiera sido escrito en un lenguaje de programación convencional.»*

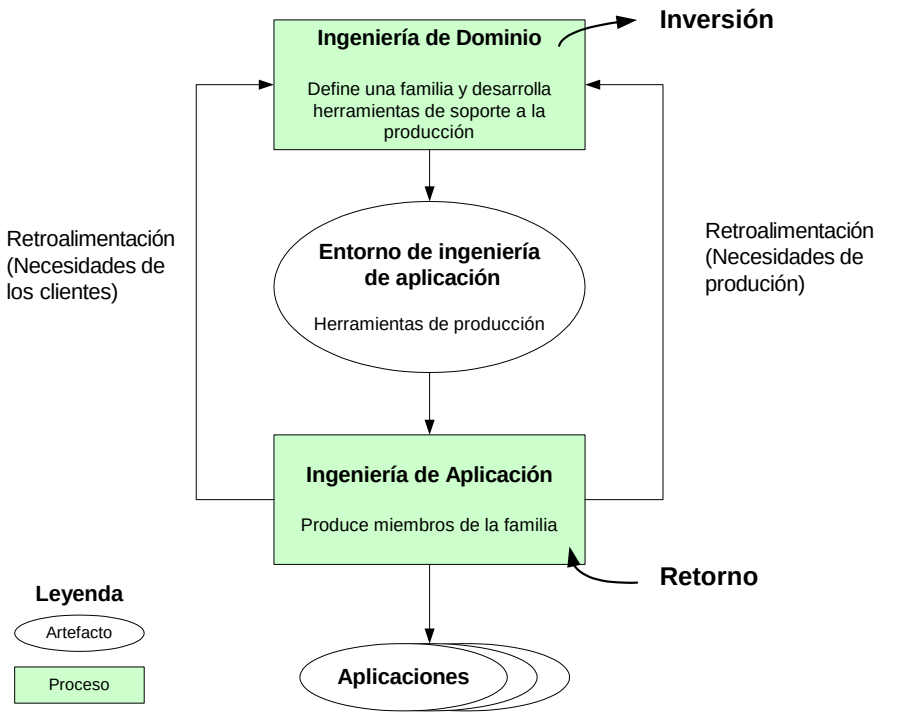


Figura 3.16: Inversión y retorno de capital en FAST (adaptado de: [Weiss99]).

- $\overline{C_F}$ como el *coste medio* de desarrollo de aplicaciones por proyecto aplicando una metodología *FAST*: $\overline{C_F} = \frac{\sum_i^{1..n} C_{Fi}}{n}$
- I como la *inversión inicial* a realizar en ingeniería de dominio para desarrollar el método de producción de aplicaciones usando un método FAST.
- Los costes acumulados de producción para N proyectos serían:

$$\begin{cases} Ca_T(N) = N * \overline{C_T} & \text{usando una metodología tradicional.} \\ Ca_F(N) = I + N * \overline{C_F} & \text{usando una metodología tipo FAST.} \end{cases}$$

Obviamente (según el postulado de Novak) ha de cumplirse que $\overline{C_F} < \overline{C_T}$. De lo contrario, no merecería la pena automatizar la producción. Si tal premisa es cierta, entonces existe un punto (*break even point*) a partir del cual usar la metodología basada en FAST es más barato que la metodología convencional. En particular, existirá un número de proyectos n_R tal que $Ca_F(n_R) < Ca_T(n_R)$. Es decir, a partir de n_R proyectos producidos se recupera la inversión inicial I.

La Figura 3.17 muestra una representación gráfica donde se muestra el punto de retorno de la inversión.

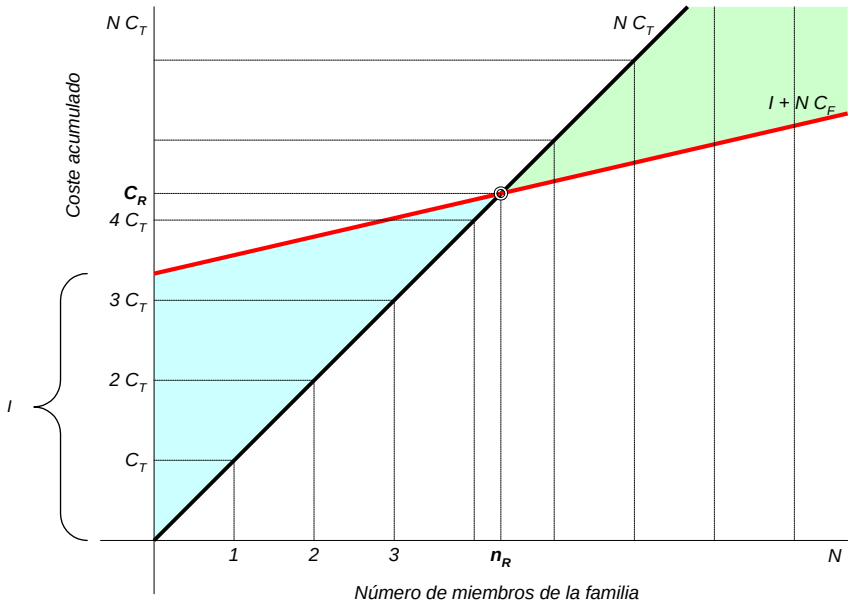


Figura 3.17: Explicación económica del método FAST.

Finalmente podemos concluir:

- Si el numero de aplicaciones a producir es superior al punto de amortización n_R, entonces será más barato emplear una técnica de tipo FAST.
- Si por el contrario n_R es demasiado grande para ser alcanzado, puede no merecer la pena invertir recursos en cambiar el modelo de producción hacia un modelo FAST.

Si comparamos los métodos tipo FAST con la producción en serie de una fábrica de galletas, podríamos decir que la producción en serie permite obtener *economía de escala* en la fabricación de galletas, mientras que un método de tipo FAST proporciona *economía de alcance*.

Veamos con más detalle la definición de ambos conceptos económicos:

- Economía de escala (*economies of scale*): «Es la condición donde pocas entradas, como esfuerzo y tiempo, son necesarias para producir grandes cantidades de una única salida.» [Withey96]

- Economía de alcance (*economies of scope*): «Es la condición donde pocas entradas, como esfuerzo y tiempo, son necesarias para producir una gran variedad de salidas. Se produce mayor valor añadido produciendo de manera conjunta diferentes salidas. Producir cada salida de modo aislado provoca un sobrecosto en las partes comunes. La economía de alcance se da cuando el coste de combinar dos o más productos en una sola línea de producción es menor que producirlos por separado.» [Withey96]

La diferencia estriba en que la fábrica produce miles de ejemplares continuamente del mismo bien (*la galleta*). Mientras que en el proceso de desarrollo de software es innecesario producir más de una vez el mismo bien (*un programa*). Todos los programas desarrollados son diferentes debido a que el coste de duplicación de un programa es prácticamente nulo.

Ventajas del método

El trabajo con una metodología de tipo FAST reporta los siguientes beneficios:

- El coste de producción se reduce considerablemente cuando los programas a producir son numerosos.
- El proceso está mejor documentado, es repetible, controlable, auditable: más cercano a las cotas de calidad conformes al modelo CMM.[20] Niveles altos de CMM son logrados en sectores industriales, sin embargo en desarrollo de aplicaciones y debido a la complejidad del proceso es muy infrecuente encontrar empresas certificadas con niveles altos de CMM.
- Eliminación de los errores de codificación.
- La fase de mantenimiento se desplaza desde el código a la especificación.

3.4.3. Generación de código basada en modelos orientados a objetos

El termino generación de código basada en modelos orientados a objetos, o más abreviadamente, OO–MBCG (*Object Oriented Model–Based Code Generation*) designa a las tecnologías que están siendo usadas para producir aplicaciones a partir de modelos de objetos.

Rodney Bell escribió un interesante artículo [Bell98] en la revista *Embedded Systems* en 1998. En dicho artículo, Bell da un repaso a la tecnología de modelos

[20] CMM: el *Capability Maturity Model* [Paulk93] esta reconocido como un modelo aceptado para medir la calidad de los procesos de producción de software.)

orientados a objetos y a las diferentes aproximaciones a la generación de código a partir de estos modelos.

Los métodos y herramientas orientados a objetos pueden mejorar la productividad, calidad y el reuso. La generación de código basada en modelos produce automáticamente código de aplicación a partir de modelos gráficos de objetos y de comportamiento. Estos métodos orientados a objetos con diagramas gráficos son útiles para analizar y documentar sistemas.

El punto débil de los métodos de análisis y diseño clásicos ha sido la transición hasta el código, la implementación. Sin generación de código, los beneficios del modelado orientado a objetos difícilmente cubren el ciclo de vida de un producto, debido a que los desarrolladores que realizan el mantenimiento, llevados por las prisas para sacar una nueva versión, obvian los modelos y sólo modifican el código. De este modo, los modelos se desfasan perdiendo, por tanto, su utilidad.

Disponer de una generación de código efectiva propicia que los modelos estén siempre actualizados y que la concordancia entre lo analizado y lo implementado se mantenga a lo largo del tiempo de vida del proyecto.

La generación de código basada en modelos sigue la tendencia de las herramientas de desarrollo. La abstracción del lenguaje se ha incrementado desde el ensamblador pasando por los lenguajes de alto nivel hasta los modelos gráficos. Las abstracciones empleadas se han movido del espacio de la solución al espacio del problema. Se necesita seguir con esta tendencia para incrementar la productividad, construir mayores aplicaciones y entender nuevos sistemas más complejos. Según Bell, todo esto sugiere que la adopción de generación de código basada en modelos será inevitable como ventaja competitiva en primer lugar y como herramienta cotidiana posteriormente.

La adopción de métodos de generación de código basados en modelos requiere considerar las siguientes facetas:

- la **adecuación** del lenguaje de modelado para representar el problema,
- la **suficiencia** de los constructores del modelo para generar el código,
- la **madurez de los traductores** para obtener código de calidad,
- las **herramientas** para tareas de desarrollo relacionadas con la generación de código, p.e. depuración,
- las **metodologías** para emplear la generación de código de un modo efectivo y
- la **selección** de herramientas y métodos apropiados a la aplicación.

Principios de funcionamiento de OO–MBCG

Existen tres tipos de aproximaciones a la generación de código basada en modelos orientados a objetos:

- **Estructural:** Generan bloques de código (como interfaces de clases) desde modelos estáticos y relaciones entre objetos.
- **De comportamiento:** Expresan mediante máquinas de estados y especificación de acciones el comportamiento de un sistema. El código generado contiene estática y dinámica.
- **Traductivo:** Desliga el modelo de la arquitectura de la aplicación. De este modo puede disponerse de modelos de aplicación independientes de plataforma y de modelos de arquitecturas independientes de aplicaciones. La traducción se realiza aplicando una arquitectura sobre un modelo de aplicación.

En los siguientes apartados se describirán las principales características de estas aproximaciones.

Aproximación Estructural

Genera bloques de código (como IDL o interfaces de clases) desde modelos estáticos y relaciones entre objetos.

Los modelos de estructura de objetos varían según los métodos (por ejemplo: modelo de objetos de OMT respecto a los diagramas de asociación de clases de Booch). Las primitivas de trabajo en estos modelos son clases, atributos, tipos y asociaciones. Algunos vendedores construyen en sus generadores el código fuente que corresponde con constructores de objetos. Otras herramientas usan un motor de traducción y plantillas preexistentes (que pueden ser modificadas y adaptadas) para especificar correspondencias con un código fuente en particular. Escritas en un lenguaje de *script*, las plantillas guían la traducción de los modelos en estructuras de código, como cabeceras de clases o declaraciones de funciones. Los lenguajes de *script* permiten a los diseñadores seguir estándares de codificación, personalizar la arquitectura del código generado y crear plantillas nuevas para lenguajes no soportados. Los traductores suelen ofrecer opciones para permitir indicar qué objetos y lenguaje usar en el proceso de generación. Frecuentemente, los diseñadores deben adaptar el modelo de arquitectura implícito en estas plantillas para soportar multitarea u otros diseños empotrados.

La generación de código estructural es apropiada a lo que se ha venido llamando *metodologías elaborativas*, donde los modelos de análisis se elaboran

gradualmente en fase de diseño e implementación. Las abstracciones se detallan y los subsistemas se diseñan y codifican para alcanzar los objetivos del proyecto. Una vez codificados, los subsistemas pueden ser probados y revisados. Debido a que los generadores pueden trabajar sobre modelos parciales, la traducción de modelos a código puede llevarse a cabo de manera incremental. Con herramientas para sincronizar y proteger el código, el desarrollo puede hacerse de manera iterativa. Sin tales herramientas, la generación de código será muy rápida para construir una versión inicial, pero deberá ser revisada a nivel de código después.

La generación de código estructural es incompleta pero ahorra esfuerzo de codificación manual y proporciona un marco de trabajo inicial consistente con los modelos. Las plantillas de traducción aportan un modesto grado de reuso.

Muchos fabricantes (incluyendo Rational, Aonix, Cayenne, Select Software, Iconix, Verilog, Mark V) soportan la generación de código estructural. La mayoría de las aplicaciones, excepto aquellas donde el coste de la memoria tenga mayor peso que la sobrecarga de clases, son adecuadas para esta aproximación. Como ejemplos de esta aproximación podemos citar herramientas que implementan UML o el modelo de Entidad Relación.

Aproximación de comportamiento

Generan código completo a partir de modelos de máquinas de estados y especificación de acciones en un lenguaje de alto nivel.

La especificación del comportamiento se basa en máquinas de estados aumentadas con especificación de acciones. Algunos métodos que modelan comportamiento con máquinas de estados añaden código (como C++ o un lenguaje propietario) para representar las acciones que ocurren durante la transición de estado. Junto con modelos de estructuras de objetos y mecanismos de comunicación, está técnica permite a las herramientas generar código para el modelo completo de la aplicación. Un beneficio de esta técnica es la capacidad para simular y verificar el comportamiento del sistema basado en modelos antes de que el código sea generado.

Los métodos que soportan la generación de código de comportamiento incluyen SDL (*Telecommunications Standard Specification and Description Language*) [SDL99], diagramas de estados jerárquicos de Harel [Harel87], ROOM (*Real-time Object-Oriented Method*) [Selic94] y más recientemente UML. Para modelar el comportamiento de un sistema completo, las máquinas de estados clásicas son extendidas: en paralelo, con máquinas de estados que se comunican (como en SDL) o con diagramas jerárquicos de estados (como en Harel y ROOM).

En los diagramas de estados los usuarios especifican el código explícito para manejar transiciones de eventos. Los lenguajes empleados incluyen C++,

C e incluso ensamblador. Los traductores usan una máquina virtual preexistente para su versión de máquina de estados que es implementada bien como librería de rutinas o bien es construida por el traductor. Esta máquina virtual implementa estados, transiciones y la comunicación con otras máquinas de estados. Los traductores integran código para el manejo de eventos (como acciones tras una transición) con la máquina de estados virtual.

Los métodos que usan esta aproximación también varían en los modos en los que se modela la estructura de los objetos y la comunicación. SDL incluye diagramas de bloques, diagramas de proceso, señales, definición de tipos de datos (con herencia) y mensajes de diagramas de secuencia para la comunicación entre objetos. El modelado estándar en SDL puede ser restringido para soportar la generación de código (ROOM, Harel). Las correspondencias de traducción típicas incluyen clases C++ (por ejemplo: para actores en ROOM) y clases miembro (datos o funciones, como en los componentes estructurales y de comportamiento en ROOM).

El contraste con la aproximación estructural, la codificación del comportamiento se ha hecho durante el modelado de objetos. En la aproximación de comportamiento, la programación se reduce a especificar manejadores de eventos (como las acciones de una máquina de estados) y objetos que son codificados a mano (por ejemplo, optimizados por motivos de rendimiento o eficiencia). Los traductores ofrecen opciones como omitir librerías de funciones para cumplir con restricciones de memoria. Las herramientas de traducción ofrecen poco control al usuario sobre la arquitectura.

Como en la aproximación estructural, la aproximación de comportamiento es usada para elaborar el análisis y los modelos de diseño. Los detalles pueden ser añadidos a la estructura de los objetos (por ejemplo, subclasificando actores en ROOM) y al comportamiento (añadiendo nuevas sentencias). Los desarrolladores pueden, progresivamente, desarrollar un modelo de implementación trabajando sobre el diseño, cambiando las simulaciones por ejecuciones de prototipos sobre la plataforma destino. Las herramientas no usan ingeniería inversa, debido a que todo el código proviene de los modelos. Esta aproximación fomenta un uso continuado de los modelos a lo largo de todo el ciclo de vida del sistema.

Los desarrolladores deben adoptar una visión de máquina de estados de la funcionalidad del sistema además de una visión de la estructura objetual del sistema. Una especificación completa de comportamiento es ejecutable y por tanto permite ser probada y depurada. La generación de código puede ser relativamente completa, con manejadores de eventos que constituyen como poco del 5 % al 10 % del código final.

La calidad del código generado debería ser comparativamente buena debido a que los traductores han madurado en respuesta a la retroalimentación de los clientes. Los proveedores de herramientas que ofrecen generación de código de comportamiento son i-Logix [i-L] (Harel), Telelogic [Tel02] y Verilog [Ver02](ambos SDL), y ObjecTime [ObjectTime] (ROOM). Las aplicaciones típicas de la industria de las telecomunicaciones además de muchos otros tipos de sistemas pueden ser modelados con máquinas de estados.

Aproximación traductiva

Las aproximaciones traductivas usan una arquitectura independiente de modelo para proporcionar a los usuarios el control total sobre la traducción de modelos completos a código.

La aproximaciones traductiva se basa en que los modelos de aplicación y arquitectura son independientes uno del otro. Un modelo de aplicación completo de estructura de objetos, comportamiento y comunicaciones es creado usando el método de Análisis Orientado a Objetos (OOA) [Coad91], el primer método OO desarrollado por Coad y Yourdon.

Como en el caso de la aproximación de comportamiento, los desarrolladores pueden simular el comportamiento del sistema antes de generar el código. Un modelo de arquitectura (un conjunto de patrones llamados plantillas o arquetipos) es desarrollada con una herramienta que soporte esta aproximación. Entonces, un motor de traducción genera el código para la aplicación de acuerdo a las reglas de correspondencia en la arquitectura. La aproximaciones traductivas ofrece un reuso significativo debido a que la aplicación y el modelo de arquitectura son independientes.

A la hora de especificar un sistema, el sistema es particionado en dominios. Un dominio es modelado por completo para permitir su simulación y generación de código. Un domino es modelado por un modelo de información de objetos, un modelo de estados para cada objeto y especificación de acciones para cada estado. La especificación de acciones se realiza en un lenguaje propietario de cada herramienta y no en un lenguaje de alto nivel. La simulación es llevada a cabo interpretando las especificación de las acciones. El modelo de aplicación es verificado por este mecanismo de simulación antes de la generación del código.

Un modelo de arquitectura es un conjunto completo de reglas de traducción que establecen una correspondencia de constructores OOA con un código fuente y mecanismos de implementación (*Run-Time*). La correspondencia debe ser completa, esto es, todo constructor OOA usado en el modelo de aplicación es traducido. Normalmente las correspondencias gestionan concurrencia (por ejemplo: múltiples hijos de ejecución, multi-tarea, mono-tarea), manejo de eventos (colas, comunicación entre procesos o flujos de entrada/salida) y

datos (estructuras, mecanismos de almacenamiento y persistencia). Cualquier lenguaje destino puede ser soportado con la aproximación traductiva: varios proyectos han usado C, C++, ADAm 4GL, SQL y ensamblador.

Construir un modelo de arquitectura es un proceso de desarrollo por sí mismo. Las correspondencias de traducción son escritas es lenguajes de *scripts* propietarios. Estas correspondencias tienen los constructores típicos: salida de cadenas literales, sustitución de objetos de modelo específicos por variables en las reglas de traducción, e iteración o selección de las reglas de traducción. Aunque construir una arquitectura requiere un esfuerzo similar a construir un compilador dirigido por tablas, afortunadamente, hay algo de ayuda. Los fabricantes de herramientas proporcionan arquitecturas genéricas que los desarrolladores pueden modificar. Existen arquitecturas desarrolladas por terceros. Los desarrolladores pueden reusar una arquitectura de otros productos en la misma plataforma para amortizar su esfuerzo. Los fabricantes están mejorando las herramientas de construcción de arquitecturas, las cuales pueden incluir librerías de plantillas, herencia de arquitecturas y ayudas para la composición.

Dado un modelo de aplicación y una arquitectura, el modelo de traducción extrae los objetos identificados del repositorio del modelo, realiza sustituciones y genera código de acuerdo a los *scripts* de reglas de correspondencias. La traducción puede incluir código de una librería *RunTime* y puede variar el código generado de acuerdo a las opciones (como omitir ciertos objetos) o anotaciones al modelo acerca de las propiedades del diseño. El desarrollador controla totalmente la generación del código en la aproximación traductiva y el código es potencialmente completo.

Los desarrolladores pueden elegir realizar ajustes manuales para objetos de rendimiento critico en ensamblador en lugar de emplear reglas para ellos. Los informes de proyectos indican que se genera hasta el 95 % del los sistemas de tamaño mediano.

La aproximación traductiva produce una aplicación, en lugar de elaborar un modelo de aplicación gradualmente en el diseño y la implementación. Esta aproximación presupone una independencia del desarrollo del modelo de aplicación.

El modelo de aplicación es independiente de la implementación (puede ser implementado en diversas arquitecturas) y el modelo de arquitectura es independiente de la aplicación (y por tanto puede ser reusado). Cuando ambos son puestos juntos, la traducción produce una aplicación. Mientras el modelo de aplicación puede ser verificado mediante simulación, el modelo de arquitectura es verificado principalmente por traducciones con éxito. Su independencia permite que los modelos de aplicación puedan ser reusados con diferentes arquitecturas y las arquitecturas para traducir otros modelos de aplicación, como

una variación en una línea de productos.

Con la aproximación traductiva, el modelo de aplicación se convierte en el *principal artefacto software* desplazando al código fuente. La combinación de arquitecturas y motor de traducción es análogo a un conjunto de compiladores para un lenguaje de alto nivel.

Como en la aproximación de comportamiento, sólo hay un sentido de generación, no hay ingeniería inversa. Para cambiar el sistema, los desarrolladores modifican la aplicación y/o el modelo de arquitectura según necesiten y regeneran el código. Aunque se puede generar código completo, la calidad del código depende por completo de los desarrolladores que han de responsabilizarse de la reprogramación el traductor. Esta variabilidad contrasta con las aproximaciones estructurales y de comportamiento, donde los vendedores garantizan una calidad en la traducción. La renuncia a está garantía de calidad se ve compensada con la flexibilidad en las arquitecturas destino, que las aproximaciones previas no pueden proporcionar. En una aproximación translativa, los desarrolladores pueden incluso cambiar de lenguaje de alto nivel (HLL, *high level language*) cambiando las correspondencias de arquitectura.

Una vez esta aproximación es completamente adoptada, una organización podría construir modelos de arquitectura para cada tecnología destino. Este desarrollo podría justificar modelos de arquitectura y grupos de aplicaciones separados, con apropiada especialización de sus respectivos expertos.

Algunos fabricantes que ofrecen herramientas para la aproximación traductiva son Project Technology [Pro] basado en el método de Shaler y Mellor [Shaler97] y Kennedy-Carter [Kennedy-Carter00].

La clave determinante de la aproximación traductiva es el potencial de reuso de las arquitecturas.

Para finalizar la descripción de las aproximaciones, la Tabla 3.2 proporciona un resumen y comparativa de las mismas.

Justificación de OO–MBCG

Bell [Bell98] justifica la adopción de OO–MBCG en términos de un análisis de coste/beneficio.

Beneficios:

Por una parte, un beneficio general de todas las aproximaciones es que refuerzan las ventajas de los modelos orientados a objetos. Estas ventajas incluyen la habilidad para construir sistemas mayores, más complejos y mejoran el mantenimiento con modelos que son mas fácilmente entendibles que el código. Con la capacidad de sincronizar cambios en los modelos y en el código

Aproximación:	*Estructural*	*De comportamiento*	*Traductiva*
Modelos soportados:	objetos	objetos/estados/ acciones	objetos/estados/ acciones/arquitectura
Metodología:	OMT, Booch, ...	Harel, SDL, ROOM, UML	Shlaer–Mellor
Lenguajes destino:	C++, Ada, ...	C++, C, ...	cualquiera (definido por el usuario)
Código generado:	marco de trabajo (esqueletos)	completa	completa, cualquier arquitectura
Control del código y de la arquitectura:	plantillas	código de usuario, librerías, opciones de generación	completa, separada, modelo de arquitectura
Soporte de herramientas:	ingeniera inversa, código protegido	simulación y depuración	simulación y depuración, construcción de la arquitectura
Coste:	pequeño	aprendizaje de las herramientas	construcción de la arquitectura
Ventaja:	sincronización entre código y modelo	verificación rápida, reuso de arquitectura	reuso de modelos, reuso de arquitecturas

Tabla 3.2: Comparación de tres aproximaciones a la OO–MBCG.

surge la oportunidad de iterar el desarrollo de la aplicación para soportar, de este modo, una metodología iterativa. El incremento de la productividad que supone la generación de código se extiende a lo largo de todo el ciclo de vida.

Las herramientas que generan código de comportamiento ofrecen capacidades para simular el comportamiento especificado. Esta facilidad proporciona la oportunidad de verificar el comportamiento del sistema durante el análisis, antes de que el diseño y el código sean desarrollados. Las herramientas para soportar una traducción traductiva mejoran el tiempo de puesta en el mercado (*time to market*), calidad y costes, permitiendo también el reuso de arquitecturas. Ambas aproximaciones pueden reducir tiempo del ciclo de desarrollo para alcanzar el producto. La aproximación traductiva puede reducir el tiempo de puesta en mercado al migrar una aplicación a una nueva plataforma (reusando el mismo modelo de aplicación).

Costes:

Las inversiones en generación de código comprenden la adopción y cambio de procesos además de costes adicionales en el proyecto inicial. El coste de cambio de los procesos incluye el aprendizaje de modelos de aplicaciones y la construcción de traductores junto con la compra y aprendizaje de las herramientas. La automatización, el reuso y la detección de errores en fases tempranas, reduce los costes de los proyectos subsiguientes y de su mantenimiento. Los costes pueden cambiar su tendencia pronto en el proyecto y afectar

a todo el ciclo de vida de la línea de producto. El mayor impacto positivo en términos de coste/beneficio proviene del incremento de competitividad (como el tiempo de puesta en mercado). Por supuesto, el coste/beneficio podría verse negativamente afectado si la adopción y uso de generación de código basada en modelos es mal gestionada.

En esta sección se ha presentado el cuarto y último pilar de esta tesis: las técnicas y métodos de generación de código.

FAST es se ha revelado como un método bien definido para abordar el desarrollo de aplicaciones en serie a partir de la identificación de una familia de aplicaciones. La serie de términos y conceptos introducidos serán de gran utilidad para explicar posteriores procesos detallados en esta tesis. Por otro lado, la explicación económica constituye un modo claro para evaluar la conveniencia o no de este tipo de aproximaciones.

La revisión a las aproximaciones de generación de código basada en modelos orientados a objetos (*OO-MBCG*) nos ha permitido revisar el estado actual de las herramientas comerciales con soporte para la producción semi-automatizada de aplicaciones.

3.5. Conclusiones del capítulo

En este capítulo se han revisado gran parte de fundamentos sobre los que se asienta el resto de capítulos de esta tesis.

El recorrido realizado ha cubierto los siguientes cuatro pilares:

1. *los modelos de especificación conceptual:* como sustrato de los modelos a construir,
2. *las interfaces de usuario:* como el dominio de trabajo,
3. *los patrones:* como mecanismos clave para describir los elementos en el espacio del problema y
4. *las tecnologías de generación de código:* con el afán de traducir de modo automático los patrones y conceptos a código directamente ejecutable.

Todo ello ha pretendido dotar al lector del transfondo necesario para abordar con seguridad y facilitar la comprensión durante la lectura del resto de los capítulos.

Capítulo 4

Especificación de interfaz de usuario: una aproximación basada en patrones

«Caminar sobre las aguas y desarrollar programas a partir de las especificaciones es fácil, si ambas están congeladas.»

— Howard V. Berard

En este capítulo se presenta el Modelo de Presentación como un modelo basado en el el lenguaje de patrones Just-UI para la especificación de interfaces de usuario. Los patrones que componen el lenguaje son presentados con detalle.

4.1. Conceptos del Modelo de Presentación

El Modelo de Presentación es un modelo para la especificación abstracta de interfaces de usuario. Se apoya en el método orientado a objetos para dar cuenta de las entidades y sus relaciones en el espacio del dominio. La información recogida de este modo, será empleada para la construcción de interfaces de usuario orientadas a objetos (*UI-OO, Object Oriented User Interfaces*) que pueden ser obtenidas traduciendo la especificación a diversos lenguajes de implementación.

4.1.1. Requisitos de construcción

A la hora de construir el Modelo de Presentación se ha intentado satisfacer desde un principio la siguiente serie de requisitos de construcción.

- **Expresividad:** El modelo debe ser rico para expresar la diversidad de conceptos que surgen en el análisis de interfaces de usuario.

- **Conjunto de constructores mínimo:** La inclusión de un nueva primitiva de construcción en el modelo deben estar debidamente justificada. A toda costa, se intentará proporcionar un conjunto canónico de constructores, evitando proporcionar mecanismos redundantes que produzcan duplicidades. Estos solapes sólo conducen a ambigüedades en la especificación o a sobre-especificaciones innecesarias.

- **Especificación mínima:** El analista debe poder expresar todo lo que sea posible buscando siempre el mínimo esfuerzo de especificación. El objetivo es maximizar la productividad del analista, ahorrando tiempo de especificación en tareas repetitivas y favoreciendo los casos frecuentes en detrimento de las excepciones si fuese necesario.

- **Alto reuso de los conceptos:** El reuso de conceptos ya definidos por el analista hace más compactas y homogéneas las especificaciones (y por ende, la interfaz final implementada[1]). El reuso y aplicación de un mismo concepto en contextos diferentes permite disponer de un numero reducido de primitivas que pueden combinarse siguiendo ciertas reglas para construir modelos válidos.

- **Prototipación rápida:** En el prototipado de aplicaciones se necesita disponer rápidamente de los prototipos para mostrar al cliente, bien para validar los requisitos en primer lugar, bien para proporcionar versiones en una fase posterior. En los estadios iniciales, la especificación del sistema está incompleta y la especificación de interfaz de usuario puede que aun no haya sido ni siquiera abordada. En estas condiciones, sería deseable poder seguir generando un prototipo marco que permita probar y validar la funcionalidad del modelo. En esta fase, el mecanismo de inferencia (como veremos en el siguiente capítulo) se revelará como una técnica esencial para la consecución de este requisito.

[1] Teniendo en cuenta que el reuso detectado en fases tempranas como en la fase de análisis se preserva en las fases sucesivas hasta llegar a la implementación.

4.1.2. Camino explorado

A la hora de construir modelos para la especificación de interfaz de usuario, optamos por el uso de patrones conceptuales de interfaz de usuario. Aunque la comunidad de investigadores en modelado y especificación de interfaz de usuario no había explorado esta vía, experiencias preliminares exitosas [Molina98] nos hicieron pensar que este camino era viable.

El uso de patrones no sólo soluciona algunos de las carencias de otros métodos clásicos, sino que además logra ocultar al analista gran parte de la complejidad subyacente. Este último logro, es el que ha permitido aplicar el modelo al desarrollo de aplicaciones industriales.

Durante los últimos cinco años hemos analizado, desarrollado, validado y evaluado diversas interfaces de usuario con el fin de localizar patrones útiles para la especificación de dichas interfaces de usuario.

En la primera versión, se disponía de una colección reducida y aislada de patrones (seis patrones descritos en [Molina98]). Según hemos ido descubriendo nuevos patrones, la colección ha tomado forma, con reglas de aplicación y composición de patrones muy claras. Por tanto, podemos decir que la colección de patrones ha evolucionado hasta constituirse en un *lenguaje de patrones*.

4.2. Lenguaje de patrones propuesto

Just-UI [Molina02d] es el nombre del lenguaje de patrones propuesto. Consta de dieciséis patrones (uno de nivel 1, cuatro de nivel 2 y once de nivel 3). Este lenguaje de patrones constituye el **Modelo de Presentación** como extensión de OO-Method para la especificación de interfaces de usuario. El modelo se estructura en tres niveles o capas de especificación, desde lo general a lo particular (*véase Figura 4.1*):

- **Nivel 1. Árbol de jerarquía de acciones:** Siguiendo el *Principio de Aproximación Gradual* expresa como la funcionalidad será presentada al usuario que accede al sistema.
- **Nivel 2. Unidades de interacción:** Modela las unidades de interacción que el usuario deberá emplear para llevar a cabo las tareas. Existen cuatro subtipos:
    1. UI de servicio
    2. UI de instancia
    3. UI de población

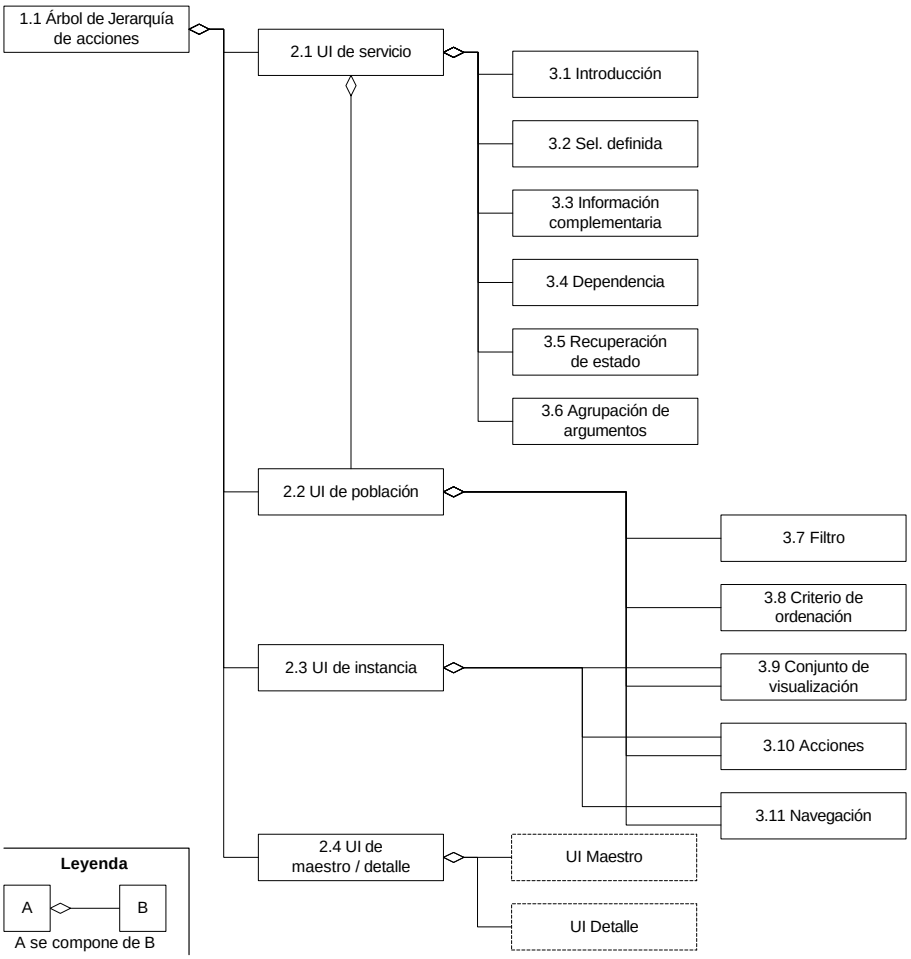


Figura 4.1: Lenguaje de patrones propuesto.

4. UI de maestro / detalle

- **Nivel 3. Patrones elementales:** Permiten restringir y precisar el comportamiento de las diferentes unidades de interacción. Son los siguientes:

  1. Introducción
  2. Selección definida
  3. Información complementaria
  4. Dependencia
  5. Recuperación de estado
  6. Agrupación de argumentos
  7. Filtro

8. Criterio de ordenación
9. Conjunto de visualización
10. Acciones
11. Navegación

4.2.1. Plantilla de descripción de los patrones

Es habitual al describir un lenguaje [Alexander77] o colección de patrones [Gamma95, Welie00a] emplear un formato de descripción lo más homogéneo posible para describir las diferentes facetas de los patrones.

El uso de una plantilla dota a las descripciones de una línea de homogeneidad para el lector y ayuda al escritor de patrones a comprobar si ha dejado algún aspecto sin describir.

Diversos autores como Gamma et al. [Gamma95] o van Welie [Welie00a] han propuesto diferentes plantillas de especificación más o menos adaptadas al contexto donde se proponen los patrones. Otros autores, en cambio, como Tidwell [Tidwell99] prefieren usar un formato narrativo, cercano al original empleado por Alexander [Alexander77].

John Vlissides argumenta en el artículo «*Seven Habits of Successful Pattern Writers*» [Vlissides96] diversos consejos para los escritores de patrones.[2] Vlissides propone el empleo de una estructura para la descripción de patrones. Esto facilita la comparación de patrones de la misma colección y entre colecciones si el formato empleado por ambas colecciones es similar.

Por otro lado, Vlissides también defiende que la estructura a emplear es dependiente del dominio a tratar. Por tanto, no es cierto el dicho de «*one size fits all*».[3]

Dadas las características de los patrones que van a ser descritos (patrones de análisis y de interfaz de usuario a la vez) se propone una plantilla de descripción adaptada a este contexto: el análisis de interfaces de usuario en un marco de modelado conceptual.

A continuación se detalla la plantilla que se ha empleado para describir los patrones de un modo estructurado:

Nombre

Describe el nombre del patrón en español y en inglés (este último en cursiva).

---

[2] En la elaboración de este capítulo se ha procurado seguir tales consejos.
[3] Un sólo tamaño sirve para todo.

Descripción

Presenta una breve descripción del patrón.

Definido en

Indica la relación de definición o alcance respecto a otros elementos del modelo.

Aplicado a

Expresa qué conceptos pueden recibir la aplicación del patrón.

Especificación

Relación de propiedades, estructura y formato empleado para definir completamente el patrón.

Meta-modelo

Extracto del meta-modelo asociado al patrón como extensión al meta-modelo de OO-Method [Pelechano98]. El meta-modelo ayuda a explicar la relación del patrón con otros elementos del modelo conceptual y constituye la base para construir herramientas de soporte al modelo como editores, representaciones del modelo y traductores del modelo.

Fundamento

Razonamiento de utilidad del patrón. Similar al apartado *Rationale* empleado en [Gamma95].

Guía de aplicación

Sugerencia de escenarios donde el patrón puede ser aplicado.

Ejemplo de problema

Ejemplo de requisito de usuario (*problema*) que ha de ser satisfecho dentro del dominio.

Captura en una herramienta CASE de modelado

Descripción de la especificación en una herramienta CASE de como resolver el problema planteado aplicando el patrón. Para los ejemplos se empleará la herramienta CASE *Oliva Nova Modeler*® desarrollada por CARE Technologies y que implementa una herramienta para el modelado conceptual incluyendo todos los patrones de interfaz de usuario aquí descritos.

Ejemplo de implementación

Ejemplos de traducción automática del patrón a diferentes plataformas empleando los traductores de interfaz de usuario desarrollados por CARE Technologies. Las traducciones se llevaron a cabo a partir de especificaciones construidas con Oliva Nova Modeler®.

A continuación se presentan uno a uno los dieciséis patrones siguiendo la estructuración basada en niveles y usando como estructura interna a cada patrón la plantilla recién descrita.

4.2.2. Nivel 1: Acceso a la aplicación

El nivel 1 esta constituido por un sólo patrón: el *Árbol de jerarquía de acciones* (AJA). Este patrón se emplea para definir el acceso a la funcionalidad de las aplicación por parte del usuario. Dependiendo del tipo de usuario, los permisos establecidos, las tareas que deba llevar a cabo o el dispositivo a emplear, el acceso a presentar puede ser radicalmente distinto. Por este motivo, se sugiere disponer de análisis previo de tareas [Paternò00] o árboles de descomposición funcionales (ARF) [Insfrán01] como ayuda al diseño de AJA efectivos.

Cada patrón comenzará en nueva página para facilitar su localización y lectura.

1.1 Árbol de jerarquía de acciones

Nombre

Árbol de jerarquía de acciones (AJA) / *Hierarchical Action Tree (HAT)*

Descripción

El Árbol de jerarquía de acciones es una abstracción útil para construir un árbol que exponga al usuario final la funcionalidad del sistema basandose en el *paradigma verbo-nombre* (*véase 3.2.2, página 68*).

Siguiendo el *principio de aproximación gradual* (*véase 29, página 77*), esta estructuración evita al usuario verse desbordado por la gran cantidad de funcionalidad que un sistema de tamaño medio podría presentar al usuario.

El analista tiene la responsabilidad de crear un árbol de acceso bien estructurado a la funcionalidad del sistema. Sin embargo, no está sólo: dispone de herramientas como la descomposición funcional del Árbol de Refinamiento de Funciones del Modelo de Requisitos[Insfrán01] o un Análisis de Tareas [Paternò00] que le pueden ayudar en la construcción del AJA.

Definido en

Vistas del modelo conceptual.[4]

Aplicado a

Vistas del modelo conceptual.

Especificación

Usando como estructura de datos un árbol, el analista puede construir los mecanismos de acceso a la aplicación. Con más detalle, el Árbol de jerarquía de acciones contiene nodos donde cada nodo cumple que:

- Los nodos intermedios (nodos con hijos, no hoja) están etiquetados con una cadena de texto a la que llamaremos *alias*. Su función es puramente agrupadora. La etiqueta se empleará como nombre visible para el usuario final.
- Los nodos hoja están también etiquetados con un alias, pero además referencian a una unidad de interacción que será alcanzada tras la selección de esta hoja por parte del usuario.

De este modo, el analista puede construir un menú abstracto donde puede agrupar por criterios arbitrarios (funcionalidad, alfabéticamente, frecuencia de uso, por pertenencia a clases, etc.) la funcionalidad del sistema que desea ofertar a los usuarios finales.

[4] Véase definición de vista en el apartado 5.4.2, *página 214*.

La implementación de este patrón sobre una herramienta CASE que dé soporte al analista en la definición, podría ayudar realizando comprobaciones de balanceado del árbol (profundidades semejantes en cada rama), ergonómicas (como es la regla del 5±2, es decir, cada nivel esta compuesto por entre tres y siete elementos, cantidades adecuadas para una buena retentiva en la memoria humana basada en estudios psicológicos) o bien proporcionando asistentes que creen árboles por defecto a partir de las clases y los métodos de éstas en un esquema conceptual. Los asistentes de éste tipo pueden encargarse de construir árboles a partir de los criterios anteriormente citados: funcionalidad, alfabéticamente, frecuencia de uso, por pertenencia a clases, etc.

El Árbol de Jerarquía de Acciones racionaliza el acceso del usuario al sistema con la finalidad de facilitar el acceso a la funcionalidad.

El analista puede decidir construir desde cero el árbol. Sin embargo no está obligado. En particular, puede apoyarse en el trabajo llevado a cabo en la toma de requisitos. Si existe trabajo previo de Ingeniería de Requisitos [Insfrán99, Insfrán00], el Árbol de Refinamiento de Funciones (*ARF*) puede ser traducido automáticamente [Insfrán01] en un un primer Árbol de Jerarquía de Acciones.

Lo que se persigue con este tipo de técnicas es favorecer un prototipado de aplicaciones muy rápido desde la toma de requisitos.

Meta-modelo

La Figura 4.2 muestra el meta-modelo para representar el árbol de jerar-

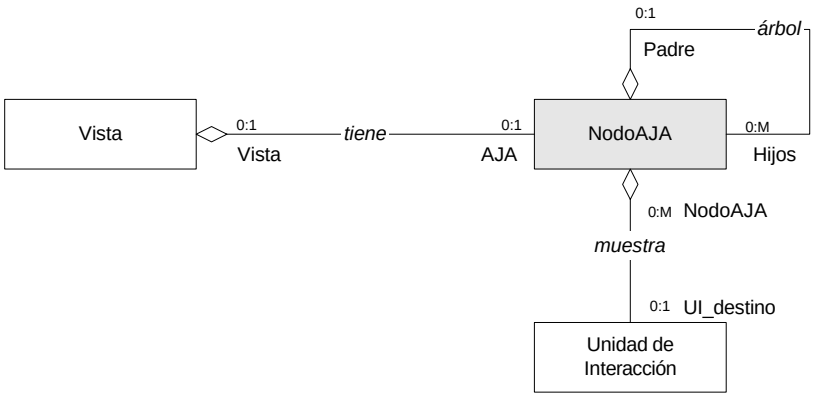


Figura 4.2: Meta-modelo del árbol de jerarquía de acciones.

quía de acciones. Un AJA está siempre definido sobre una *vista*. El árbol es una sucesión de nodos etiquetados formando un árbol hasta alcanzar las hojas del árbol compuestas por unidades de interacción destino.

Semántica asociada

Cada nodo intermedio del árbol sirve como elemento de agrupación. Las hojas contienen unidades de interacción para proporcionar acceso a la funcionalidad. El Árbol de Jerarquía de Acciones organiza el modo en el que los servicios van a ser ofertados al usuario. La organización en ramas del árbol sigue el *Principio de Aproximación Gradual* que ayuda a reducir la carga mental del usuario.

Fundamento

El patrón AJA es útil para estructurar de un modo lógico la funcionalidad de la aplicación al usuario. Siguiendo el *Principio de Aproximación Gradual* cuando el número de elementos comienza a superar la decena es mejor estructurarlos jerárquicamente para facilitar su localización a los usuarios.

Guía de aplicación

El patrón AJA puede emplearse para definir un modo racional para acceder a la funcionalidad de la aplicación. Un análisis previo de tareas [Paternò00] o un estudio de Ingeniería de Requisitos pueden guiar de modo efectivo la creación de un árbol adecuado aplicando las técnicas descritas en [Insfrán01].

Ejemplo de problema

Tenemos especificado un sistema para la gestión de una biblioteca. Necesitamos expresar como el usuario, en este caso el bibliotecario, accederá a la funcionalidad del sistema. Para este ejemplo, los analistas han determinado un organización lógica en función de las preferencias y frecuencia de uso de los servicios del sistema. Las agrupaciones finales son:

- Biblioteca
  - Autores
  - Estanterías
  - Libros
  - Materias
  - Proveedores
  - Pedidos

Dentro de cada agrupación se presentará la funcionalidad asociada a cada clase.

Captura en una herramienta CASE de modelado

La Figura 4.3 muestra como se ha especificado en la herramienta Oliva Nova Modeler® el árbol de jerarquía de acciones que soluciona el problema planteado en la sección previa. La parte derecha de la ventana muestra un

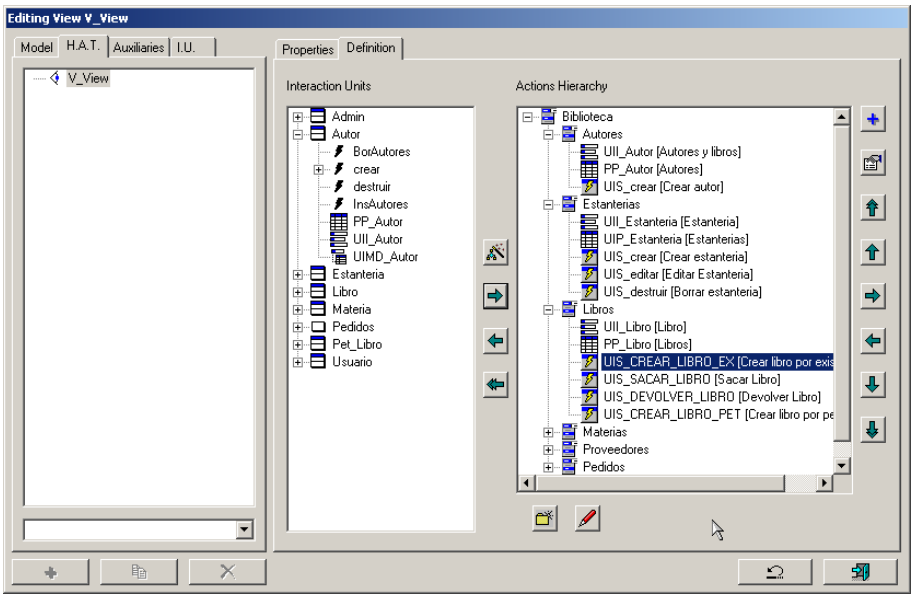


Figura 4.3: Ejemplo de especificación del Árbol de jerarquía de acciones.

árbol donde los nodos intermedios son agrupadores creados ex-proceso para organizar el acceso. Mientas que los nodos hoja, son unidades de interacción definidos en el modelo.

Ejemplo de implementación

La Figura 4.4 muestra una implementación del Árbol de jerarquía de acciones generada para la plataforma Windows. En ella, el AJA ha sido traducido a un menú clásico de las aplicaciones Windows. Nótese también como entradas estándar como Fichero o Ayuda han sido también producidas aunque no estaban en la especificación original. Esto es debido a que, para la plataforma empleada, la Guía de estilo para aplicaciones Windows (*véase apéndice E*) recomienda que todos los menús de aplicación incorporen estas características por defecto con el comportamiento estándar.

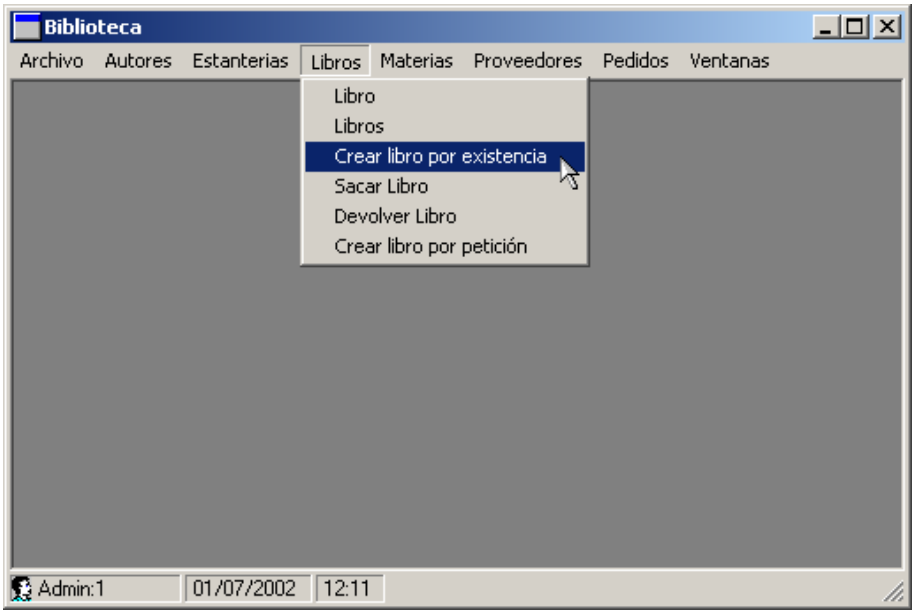


Figura 4.4: Ejemplo de implementación del Árbol de jerarquía de acciones.

4.2.3. Nivel 2: Unidades de interacción

Las Unidades de Presentación [Bodart96] (PU, *Presentation Units*) son AIO [Vanderdonckt93] que abstraen la presentación de contextos o escenarios de interacción con el usuario. Una ventana en Macintosh o en X11, un formulario en Windows o una página Web sobre un navegador pueden ser vistas como ejemplos de unidades de presentación que contienen conjuntos de AIO.

Sin embargo, para la especificación de interfaces de usuario, no sólo estamos interesados en los aspectos de presentación (estática), sino en el comportamiento (dinámico) de dicha interfaz de usuario. Por tal motivo, definimos las **Unidades de Interacción** [Molina02d] (IU, *Interaction Units*) como Unidades de Presentación donde no sólo se abstrae la presentación sino también el comportamiento de la unidad por medio de tipificación según patrones. En el proceso de abstracción se desechan los detalles de implementación y decoración y se abstrae el comportamiento esencial en términos de interacción hombre–máquina.

Hasta la fecha, hemos detectado y caracterizado las siguientes unidades de interacción como patrones de interacción abstractos:

1. Unidad de Interacción de servicio
2. Unidad de Interacción de instancia
3. Unidad de Interacción de población
4. Unidad de Interacción de maestro/detalle

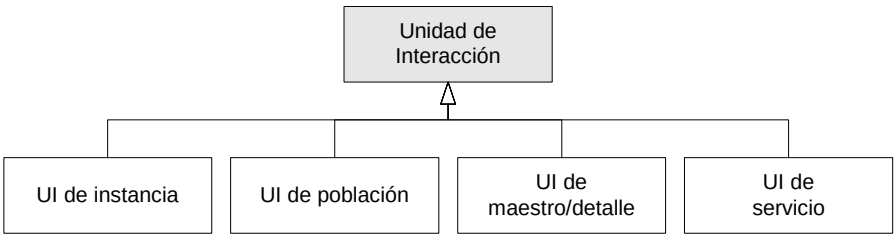


Figura 4.5: Meta-modelo de las unidades de interacción.

La Figura 4.5 muestra el meta-modelo asociado a las unidades de interacción. En dicha figura, la Unidad de Interacción es una clase *abstracta*, es decir, no tiene población directa, sino que todas sus instancias lo son a su vez, de una clase especializada de ésta.

2.1 Unidad de interacción de servicio

Nombre

Unidad de interacción de servicio / *Service Interaction Unit*

Descripción

Modela la presentación de un diálogo encaminado a que un usuario lance un servicio. La funcionalidad subyacente a este patrón consiste en proporcionar al usuario final:

- la posibilidad de proporcionar los argumentos necesarios para el lanzamiento del servicio,
- la validación de dichos argumentos,
- una acción para proceder al lanzamiento del servicio,
- una acción para la cancelación del servicio y
- que informe al usuario de errores los producidos, o bien, confirmación de ejecución satisfactoria.

Definido en

Servicios.

Aplicado a

Jerarquía de acciones, Acciones.

Especificación

Nombre de la unidad de interacción. El propio servicio tiene toda la información necesaria para obtener la unidad de interacción.

Los patrones elementales que son aplicables dentro del ámbito de las unidades de interacción de servicio son:

- Introducción
- Selección definida
- Agrupación de argumentos
- Selección de objetos (mediante UI de población para argumentos objeto-valuados)
- Información complementaria
- Recuperación de estado

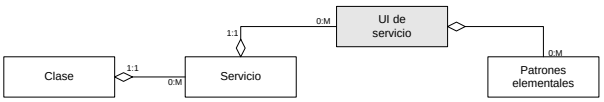


Figura 4.6: Meta-modelo de la unidad de interacción de servicio.

Meta-modelo

La Figura 4.6 muestra el meta-modelo asociado a las unidades de interacción de servicio. Un servicio puede tener definidas muchas UI de servicio, mientras que la UI de servicio lo es de un único servicio. A su vez, otros patrones elementales (que describiremos más adelante) pueden ser aplicados dentro del ámbito de una UI de servicio.

Semántica asociada

El patrón de presentación de servicio encapsula el diálogo que el usuario ha de establecer con el sistema para llevar a cabo el lanzamiento de un servicio:

- Los argumentos del servicio se mostrarán al usuario para que pueda dar valor a cada uno de ellos.
- Los argumentos serán validados de acuerdo a su tipo, grado de obligatoriedad, etc. No se podrá lanzar el servicio si los valores proporcionados por el usuario no son válidos, estableciendo de este modo un primer nivel de validación de datos en el nivel de interfaz de usuario que será completado con un segundo nivel de validación más férreo a nivel de lógica de negocio.
- Se proporcionarán acciones para proceder al lanzamiento del servicio y para cancelar su ejecución.
- Podría proporcionarse soporte a la operación deshacer (*Undo*) cuando ésta esté disponible.
- Tras la ejecución del servicio se informará al usuario de los errores que pudieran haberse producido, o bien confirmación de ejecución satisfactoria.

Fundamento

En los sistemas de información es muy frecuente la presentación de formularios que han de ser completados por el usuario. Una vez completados, los datos son validados y desencadenan un cambio en el estado del sistema. La unidad de interacción de servicio es adecuada para modelar este tipo de situaciones.

Guía de aplicación

Cuando el analista detecte un escenario donde deba ser invocados servicios definidos en el sistema.

Ejemplo de problema

Existe un servicio destinado a formalizar la compra de un vehículo por parte de un cliente. Dicho servicio debe ser ofertado al usuario final para proporcionar los datos correspondientes.

Captura en una herramienta CASE de modelado

La Figura 4.7 muestra la pantalla de definición en Oliva Nova Modeler® donde la UI de Servicio denominada UIS_COMPRA asociada al servicio COMPRA de la clase Coche. Al mismo tiempo se fija *"Compra"* como el alias del servicio.

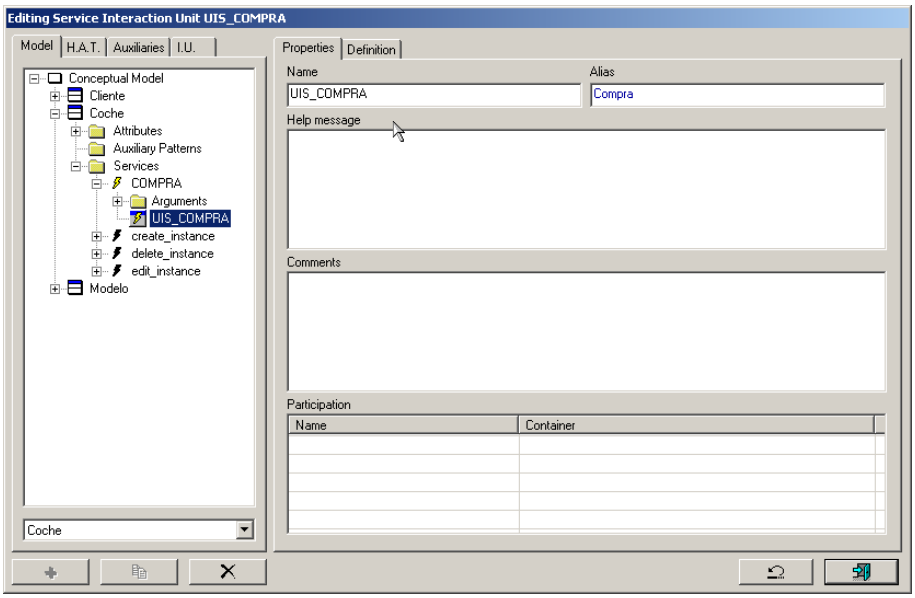


Figura 4.7: Especificación de una unidad de interacción de servicio 1/2.

La segunda pestaña (*véase Figura 4.8*) muestra los aspectos de definición del patrón donde puede apreciarse los argumentos y sus alias. Desde esta pantalla es posible aplicar el patrón de *Agrupación de argumentos* que se describirá en la página 169.

Ejemplo de implementación

El ejemplo de implementación (*véase Figura 4.9*) muestra como la unidad de interacción de servicio ha sido traducida a una ventana Windows donde cada argumento aparece representado con un CIO dependiendo del tipo de

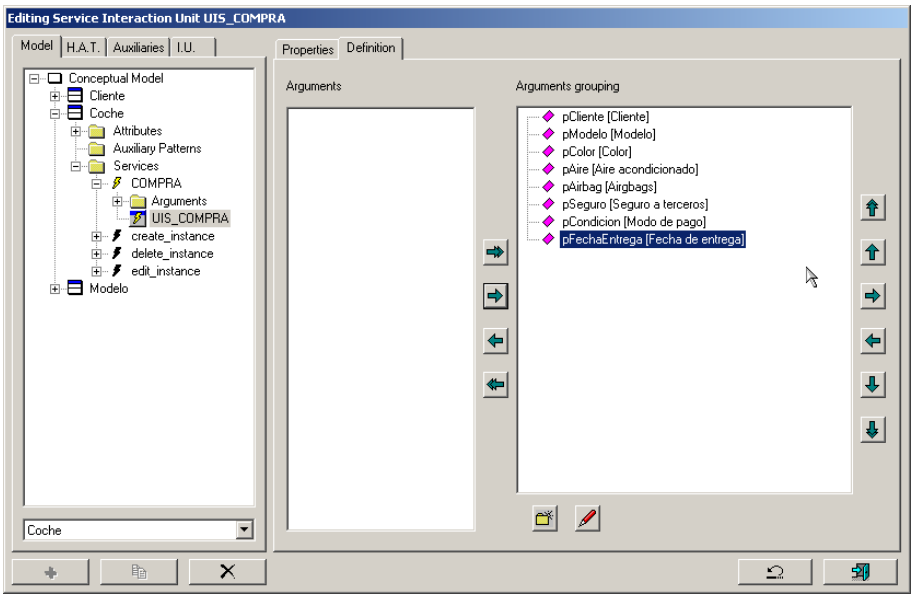


Figura 4.8: Especificación de una unidad de interacción de servicio 2/2.

cada argumento y de los patrones aplicados. Los botones Aceptar y Cancelar permiten el lanzamiento del servicio o su cancelación, respectivamente.

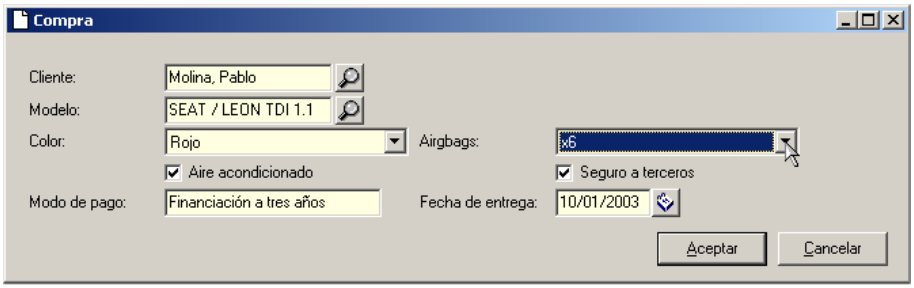


Figura 4.9: Ejemplo de implementación de una unidad de interacción de servicio.

2.2 Unidad de interacción de instancia

Nombre

Unidad de interacción de instancia / *Instance Interaction Unit*

Descripción

Modela la presentación de los datos o estado de una instancia. El usuario puede observar detalles de la instancia.

Definido en

Clase.

Aplicado a

Jerarquía de acciones, Acciones, Navegación, Maestro/Detalle y Clase (UI de instancia por defecto).

Especificación

Una unidad de instancia se define sobre una clase. Se le proporciona un nombre y las siguientes propiedades:

- *Conjunto de visualización:* Atributos pertenecientes a la clase actual (o alcanzables desde esta) que se mostrarán.
- *Acciones:* Acciones que pueden ser lanzadas sobre el objeto que está siendo presentado.
- *Navegación:* Conjunto de roles, superclases y subclases. Permite alcanzar instancias relacionadas por agregación o mostrar facetas de herencia del objeto que está siendo presentado.

Meta-modelo

La Figura 4.10 muestra el meta-modelo asociado a las unidad de interacción de instancia. Una UI de Instancia se define en una clase y se compone de un conjunto de visualización, acciones (opcional) y navegación (opcional).

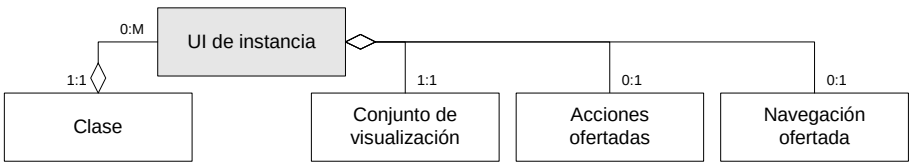


Figura 4.10: Meta-modelo de la unidad de interacción de instancia.

Semántica asociada

La unidad de interacción de instancia indica qué información será presentada al usuario respecto a los objetos de una clase. La navegación y las acciones posibles dentro de este contexto son también precisadas.

Fundamento

Este patrón describe las propiedades básicas involucradas en la observación de objetos, describiendo:

- *qué información ha de ser mostrada*,
- *qué acciones deben ofertarse sobre el objeto* y
- *qué información adicional puede ser consultada.*

Guía de aplicación

Este patrón es útil allí donde se requiera mostrar información detallada sobre **un objeto**.

Ejemplo de problema

En un escenario dado, se desea mostrar la información detallada de cada cliente. Se necesita mostrar el nombre, el apellido, la dirección postal del cliente y los teléfonos de contacto. Se han detectado una serie de acciones o tareas que pueden ser iniciadas en este escenario como son: crear un nuevo cliente, borrarlo, modificar los datos si se observa que son erróneos, etc. Además de esta información básica, en ocasiones es necesario consultar más datos relativos al cliente como pueden ser sus últimos pedidos, pagos, facturas o la moneda que suele emplear este cliente.

Captura en una herramienta CASE de modelado

La captura de este patrón comienza con la definición de una nueva UI de instancia definida sobre la clase Cliente (*véase Figura 4.11*). El nombre empleado para la instancia del patrón es UII_Cliente.

A continuación, el siguiente paso consiste en fijar el *Conjunto de visualización*, *Acciones* y *Navegación* asociadas al patrón de instancia que estamos definiendo. Por simplicidad, en la Figura 4.12 aparecen ya creados y seleccionados estos tres elementos. Aunque obviamente, estas subpiezas deben definirse si no lo están[5].

El conjunto de visualización empleado (CV_Cliente), usa los siguientes atributos para ser mostrados al usuario:

- Nombre, Apellidos, Direccion, Provincia, Pais, Telefono, Fax

[5]Véanse los patrones *conjunto de visualización* (pág. 182), *acciones* (pág. 186) y *navegación* (pág. 190).

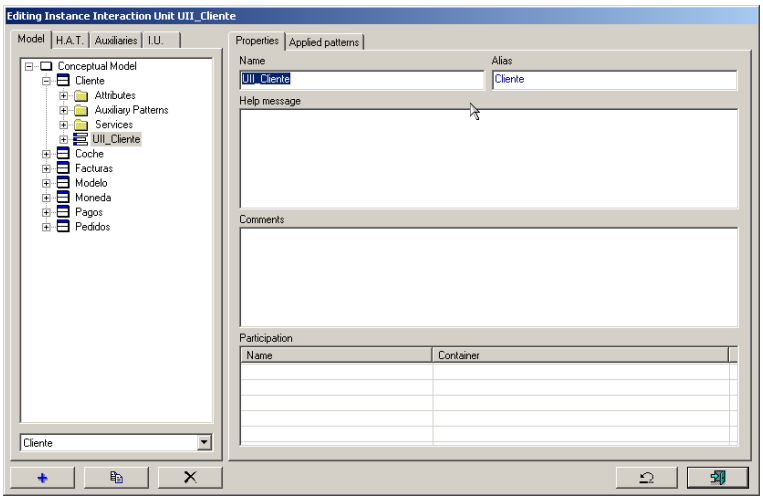


Figura 4.11: Ventana de definición de la UI de Instancia 1/2.

Las acciones (AO_Cliente), define las siguientes unidades de interacción como accesibles:

- UIS_create_instance, UIS_delete_instance, UIS_edit_instance, UIS_Einforme, UIS_Imp

Por último, la navegación (NO_Cliente), define las siguientes unidades de interacción accesibles para mostrar información relacionada:

- UIP_Pedidos, UIP_Factura, UIP_Pagos, UIP_Moneda

Ejemplo de implementación

La implementación del patrón para plataformas Windows (*véase Figura 4.13*) consiste en una ventana donde se muestra la información a mostrar. Dos barras de botones (una vertical y otra horizontal) dan cuenta de la funcionalidad ofertada por el patrón Acciones (*véase página 186*) y Navegación (*véase página 190*), respectivamente.

En la parte superior de la figura aparece un control selector de objetos que permite identificar un objeto por su función de identificación. El área de trabajo muestra el estado actual del objeto, mostrando los diferentes valores actuales para las propiedades.

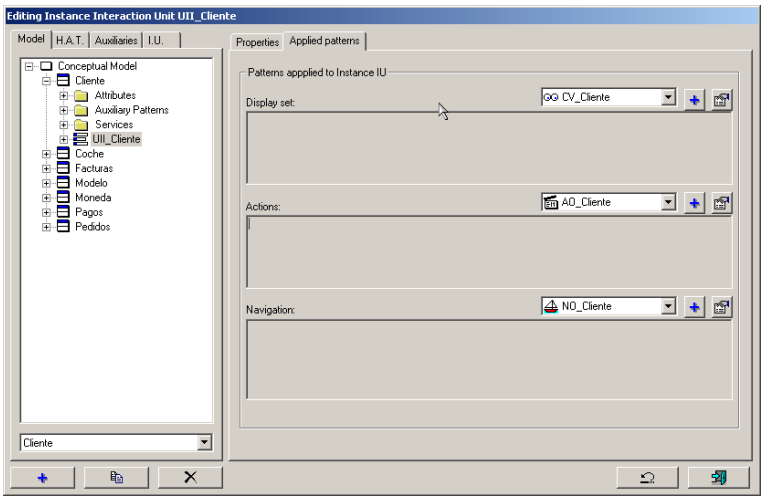


Figura 4.12: Ventana de definición de la UI de Instancia 2/2.

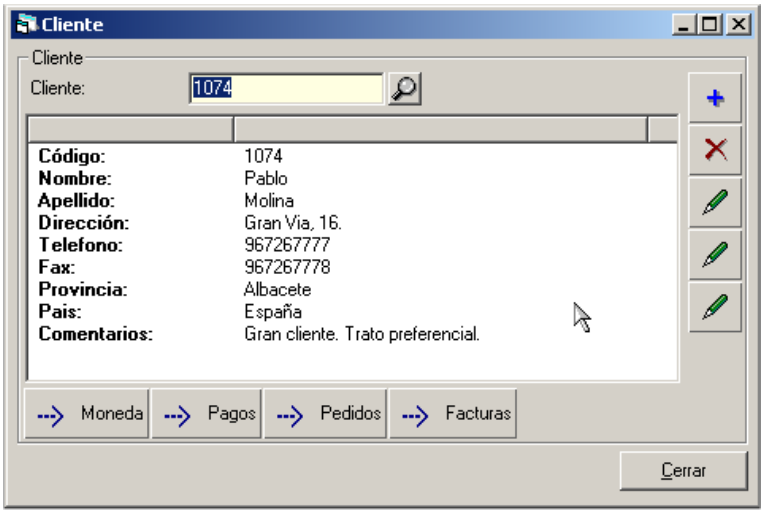


Figura 4.13: Ejemplo de implementación de la UI de instancia.

2.3 Unidad de Interacción de Población

Nombre

Unidad de interacción de población / *Population Interaction Unit*

Descripción

Permite modelar unidades de interacción encaminadas a mostrar conjuntos de instancias para una clase dada. Establece los mecanismos que facilitan la selección y consulta de objetos, proporcionando posibilidades de filtrado y ordenación.

Definido en

Clase.

Aplicado a

Árbol de jerarquía de acciones, Acciones, Navegación, Maestro/Detalle, Clase (UI Población por defecto), Argumentos objeto-valuados (Selección de objetos) y Variables de filtro objeto-valuadas (Selección de objetos).

Especificación

La unidad de interacción de población consta de uno o más conjuntos de visualización, criterios de ordenación (opcionales), filtros (opcionales, en su defecto se entenderá como filtro TRUE, toda la población de la clase), navegación (opcional) y acciones (opcional) (*véase Tabla 4.1*).

- El conjunto de visualización fija qué información relativa a un objeto será mostrada y en qué orden.
- El criterio de ordenación fija la ordenación en base a campos de ordenación e indicando ordenación ascendente o descendente.
- El filtro es una fórmula bien formada destinada a restringir la población en consultas sobre la población de la clase. El filtro puede contener variables que permitan parametrizar la búsqueda.
- Acciones: Acciones que pueden ser invocadas sobre un objeto que esté seleccionado. Una acción puede consistir, por ejemplo, en acceder a una unidad de interacción de servicio para la ejecución de un servicio o una unidad de presentación de instancia para detallar el estado del objeto actualmente seleccionado.
- Navegación: Conjunto de roles, clases padres e hijas. Permite alcanzar instancias relacionadas por agregación o mostrar facetas de herencia del objeto que está seleccionado.

Propiedad	Descripción
Conjunto de visualización	Es una lista ordenada de atributos que mostraran los valores asociados para cada objeto consultado por el usuario (*véase página 182*).
Criterio de ordenación	Lista ordenada de atributos que fija como serán ordenados los objetos para su presentación (*véase página 178*).
Filtro	Fórmula bien formada que restringe la población de objetos de una clase para permitir realizar búsquedas por aproximaciones sucesivas (*véase página 173*).
Acciones	Serie de acciones ordenadas que usuario puede realizar sobre el objeto seleccionado (*véase página 186*).
Navegación	Lista ordenada de roles y facetas que permiten navegar a información relacionada por agregación y herencia (*véase página 190*).

Tabla 4.1: Información para la especificación de la unidad de población.

Meta-modelo

La Figura 4.14 muestra el meta-modelo asociado a las unidad de interacción de población. Una UI de Población se define en una clase y se compone de:

- varios filtros (opcionales),
- varios criterios de ordenación (opcional),
- uno o varios conjuntos de visualización,
- acciones (opcional) y
- navegación (opcional).

Cada filtro puede, a su vez, establecer una preferencia con un criterio de ordenación (criterio de ordenación por defecto para ese filtro en esa unidad de

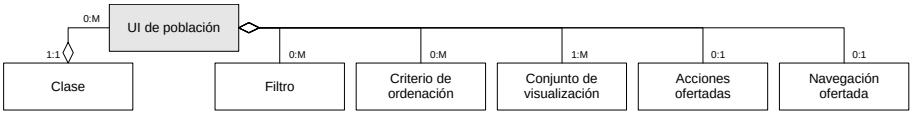


Figura 4.14: Meta-modelo de la unidad de interacción de población.

interacción).

Semántica asociada

Este patrón encapsula la funcionalidad requerida para buscar, seleccionar y manipular instancias de una clase dada. Se proporcionan capacidades de filtrado (qué instancias mostrar), ordenación (cómo ordenarlas) y de presentación (qué atributos mostrar).

Fundamento

Este patrón proporciona la funcionalidad para trabajar con conjuntos de objetos definiendo:

- *qué información debe ser mostrada* al usuario,
- *qué mecanismos de búsqueda* se disponen,
- *qué mecanismos de ordenación* están disponibles,
- *qué acciones* pueden ser lanzadas sobre los objetos y
- *qué navegación* adicional se facilita para consultar objetos relacionados.

Guía de aplicación

Este patrón se aplica cuando el usuario necesita trabajar con conjuntos de objetos.

Ejemplo de problema

En nuestro sistema de gestión de bibliotecas se hace necesario que los usuarios puedan consultar la población de libros y realizar búsquedas usando diversos criterios para localizar los libros.

Captura en una herramienta CASE de modelado

Para mostrar los libros al bibliotecario, podemos definir una unidad de interacción de población (PP_Libro) como muestra la Figura 4.15.

La segunda y tercera pestañas (mostradas en las Figuras 4.16 y 4.17) muestra los patrones elementales que participan en la definición de este patrón: filtros, criterios de ordenación, conjuntos de visualización, navegación y acciones.

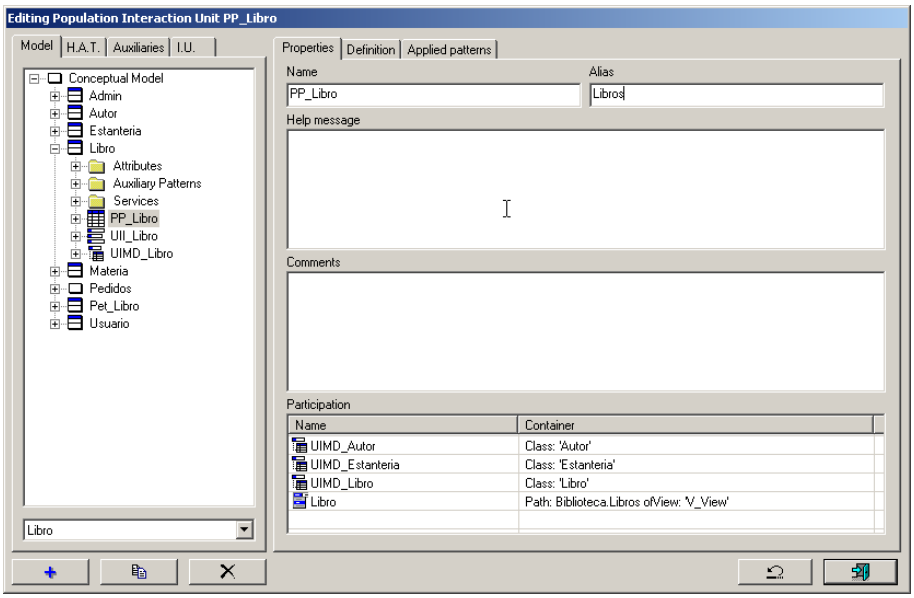


Figura 4.15: Ventana de definición de la UI de Población 1/3.

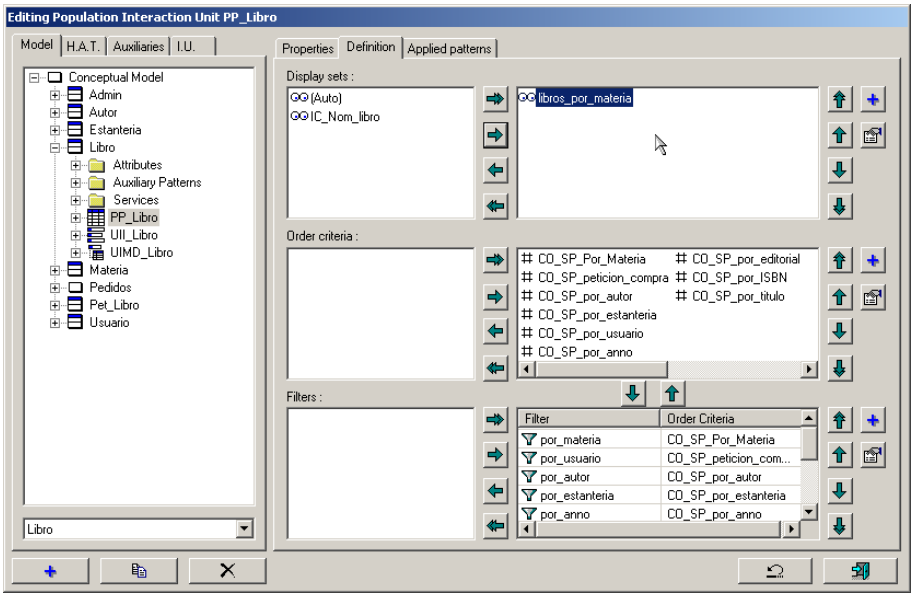


Figura 4.16: Ventana de definición de la UI de Población 2/3.

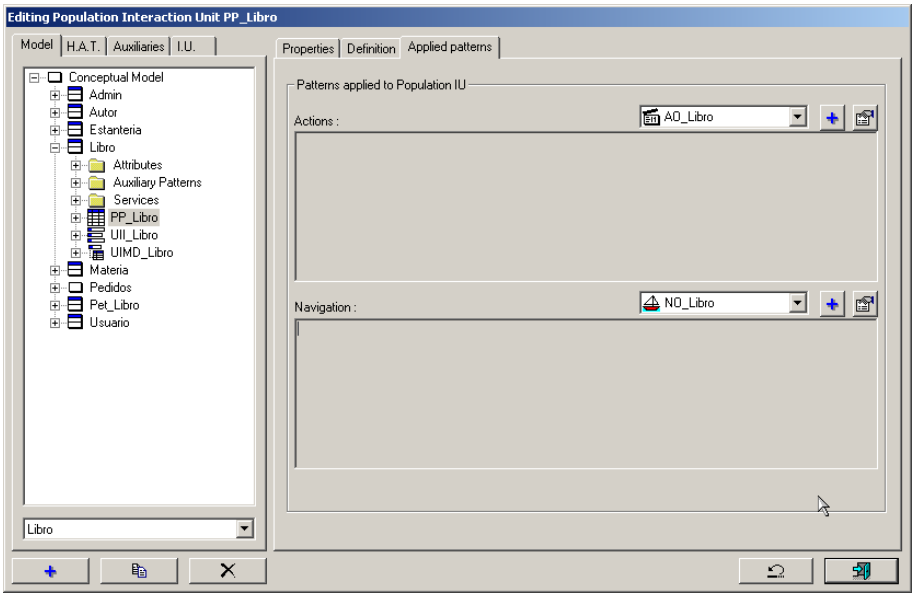


Figura 4.17: Ventana de definición de la UI de Población 3/3.

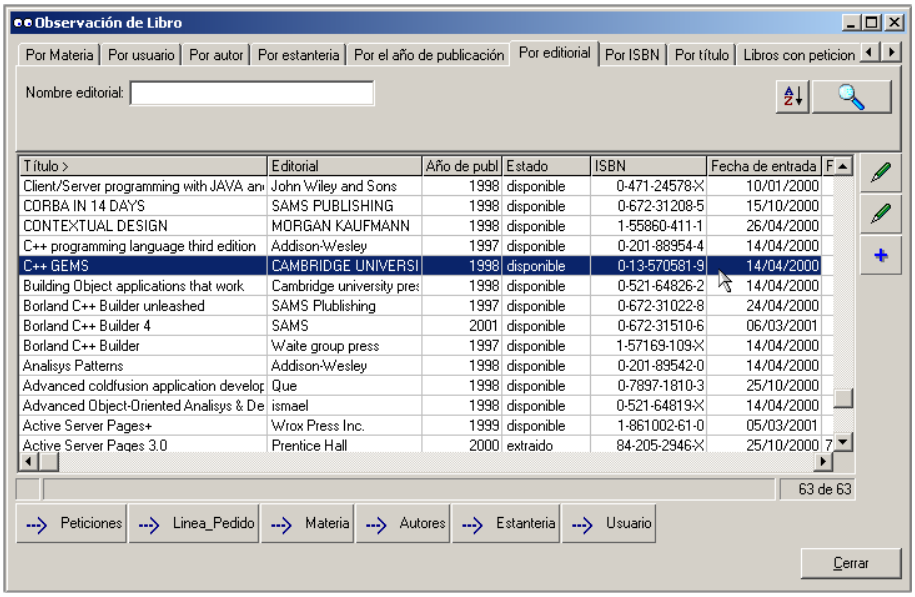


Figura 4.18: Ejemplo de implementación de UI de Población en Windows.

Ejemplo de implementación

Como ejemplo de implementación, la Figura 4.18 muestra la apariencia de la unidad de interacción bajo Windows. Una rejilla muestra los datos. Las columnas siguen la definición impuesta por el conjunto de visualización. Los filtros se organizan en pestañas en la parte superior de la ventana. Las acciones y navegaciones están soportadas por las barras de botones con orientación vertical y horizontal, respectivamente.

Por otro lado, la Figura 4.19 muestra la implementación producida para una aplicación Web. La implementación usa una tabla HTML para mostrar los datos. Un control desplegable permite seleccionar el filtro a aplicar. En esta implementación los enlaces sirven para proporcionar las acciones y la navegación.

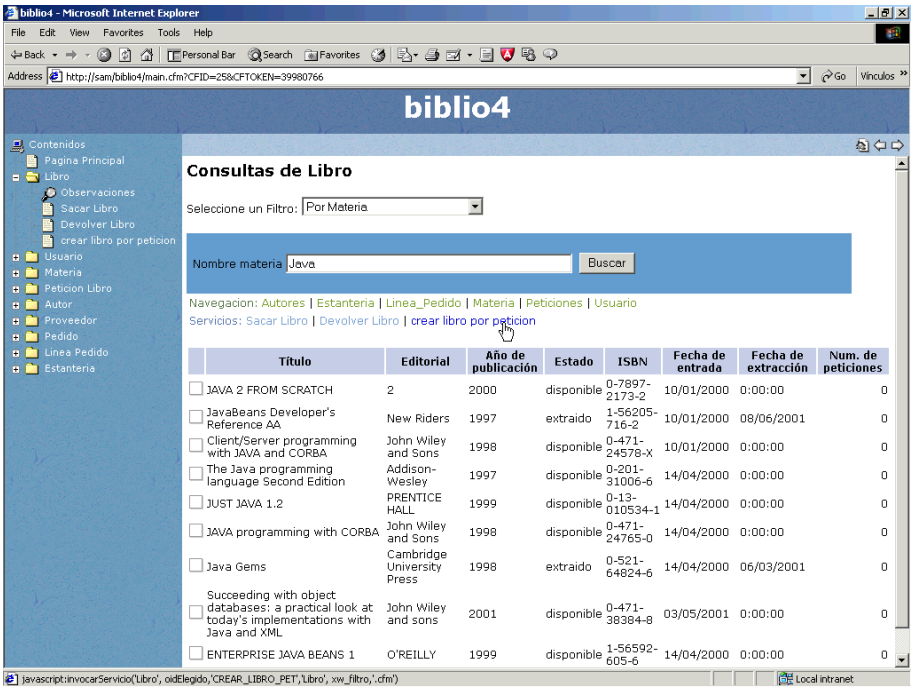


Figura 4.19: Ejemplo de implementación de UI de Población en Web.

2.4 Unidad de interacción de maestro / detalle

Nombre

Unidad de interacción de maestro/detalle / *Master/Detail Interaction Unit*

Descripción

Modela unidades de interacción más complejas de presentación cabecera-despliegue que serán construidas de manera incremental a partir de unidades de interacción más sencillas. Dichas unidades de interacción están divididas en dos componentes lógicos, un componente maestro (que actua como objeto director) y otro de detalle (mostrando información vinculada al objeto maestro).

Cada vez que la instancia del componente maestro cambia, cambian a su vez, el detalle. De modo que: siempre se cumple que los objetos mostrados como detalle pertenecen (están relacionados) con la instancia maestra presentada.

Definido en

Clase.

Aplicado a

Árbol de jerarquía de acciones, Acciones, Navegación, Detalle.

Especificación

Una UI de población tiene las siguientes propiedades:

- *Nombre.* Identificador de la unidad de interacción.
- *Alias.* Etiqueta visible para el usuario representando a la UI.
- Maestro
  - *Unidad de interacción maestro.* Es una unidad de interacción definida en la clase que actúa como director. Tiene como principal característica la de ser capaz de fijar un objeto.
- Detalles
  - *Unidades de interacción de detalle.* Una o más unidades de interacción que actúan como detalles (deben ser capaces de mostrar objetos).
  - *Fórmula* bien formada que indica la población alcanzada en el detalle. Para cada detalle, es preciso proporcionar los roles atravesados en la clase maestra hasta alcanzar el detalle (si los hubiere). Si se emplea un camino que involucra varios roles multivaluados, *p.e.* sobre la clase Oferta la fórmula Secciones.Lineas, se entenderá con la siguiente semántica: se mostraran como detalle todas las líneas de todas las secciones de una oferta.

- *Alias.* Etiqueta visible para el usuario representando al detalle.

Meta-modelo

La Figura 4.20 muestra el meta-modelo asociado a las unidad de interac-

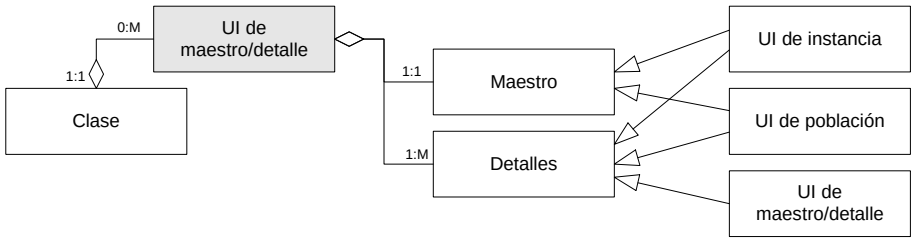


Figura 4.20: Meta-modelo de la unidad de interacción de maestro/detalle.

ción de maestro/detalle. Una UI de maestro/detalle se define en una clase y se compone de:

- un componente *maestro* (UI de instancia o de población),
- uno o más *detalles*:
  - una unidad de interacción de detalle (UI de instancia, población o maestro/detalle)
  - una fórmula de navegación que relaciona el componente maestro con el detalle.

Semántica asociada

El patrón de presentación de maestro/detalle representa una composición de unidades de interacción encapsulando al mismo tiempo comportamiento de sincronización entre el componente maestro y los detalles. Siempre que cambian los datos del componente maestro, el detalle cambia automáticamente para sincronizar sus datos respecto al maestro.

Fundamento

La unidad interacción de maestro/detalle responde a la necesidad de los usuarios de disponer de información relacionada en un mismo escenario. La sincronización de la información entre los componentes maestro y detalle, garantiza al usuario que la información presentada en los detalles corresponde al objeto maestro seleccionado.

La aplicación del patrón Unidad de Interacción de Maestro/Detalle está fuertemente vinculada a la existencia de relaciones de agregación entre los objetos maestro y detalles. La existencia de relaciones de agregación con cardinalidad genérica *«1 a muchos»* como ocurre en los pares: *Factura/Lineas* o *Autor/Libros* pueden ser firmes candidatos a ser presentados de un modo natural

como unidades de maestro/detalle. Ya que, de este modo, la presentación de la información vinculada resulta muy sencilla de interpretar por parte de los usuarios de la aplicación.

Guía de aplicación

El patrón puede ser aplicado en los contextos donde el analista detecte que el usuario necesita mostrar *información relacionada* respecto un objeto *central* y que dicha información debe estar situada dentro de un mismo escenario. En estas condiciones dicho objeto *central* actuara como maestro y la *información relacionada* puede disponerse como detalles.

Ejemplo de problema

En el sistema de gestión de bibliotecas existe un requisito por el cual se necesita consultar los libros disponibles de un autor dado. Este requisito puede ser cubierto fácilmente aplicando un patrón maestro/detalle donde una unidad de interacción definida para la clase Autor es empleada como *maestro* y una unidad de interacción definida para la clase Libro es empleada como *detalle*.

Captura en una herramienta CASE de modelado

Para definir una nueva unidad de interacción de maestro/detalle usaremos la ventana habitual de declaración para dar un nombre (UIMD_Autor) y un alias (*"Autor"*) al patrón (*véase Figura 4.21*). La segunda pestaña de la ven-

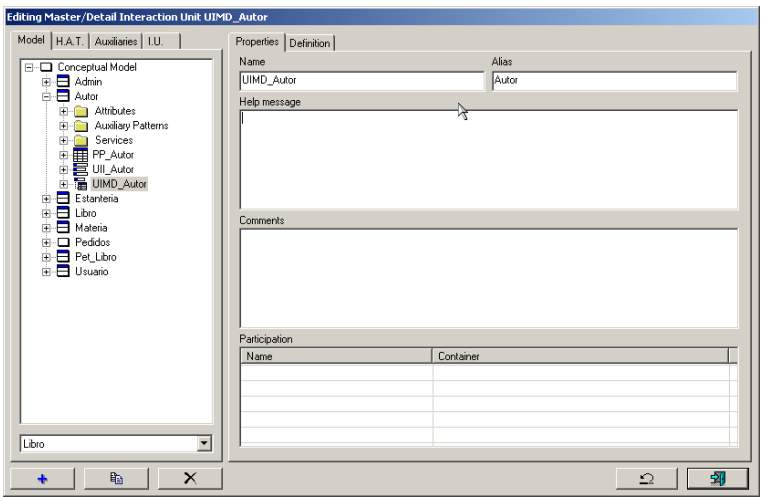


Figura 4.21: Ejemplo de especificación de UI de Maestro/Detalle 1/2.

tana de definición (*véase Figura 4.22*) contiene la parte importante de la especificación. Es aquí donde se indica:

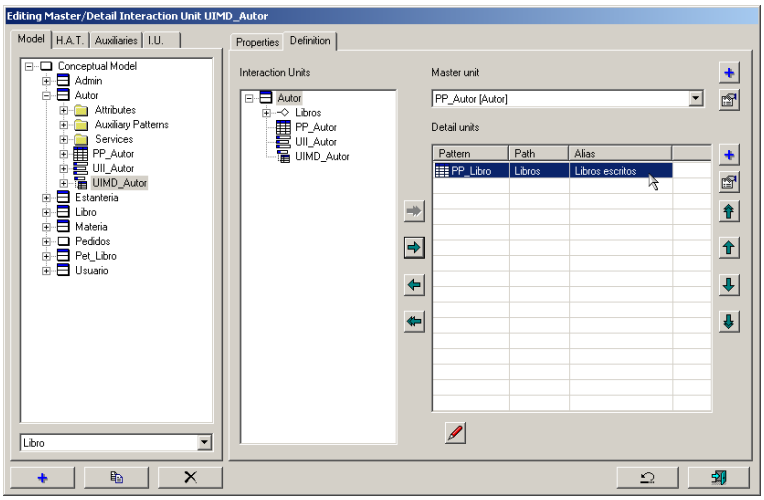


Figura 4.22: Ejemplo de especificación de UI de Maestro/Detalle 2/2.

- qué unidad de interacción actuará como *maestro* (PP_Autor en el ejemplo) y
- qué unidades de interacción actuarán como *detalles* (en el ejemplo hay sólo una definida: PP_Libro que es alcanzada tras cruzar el rol Libros desde la clase Autor).

Ejemplo de implementación

A nivel conceptual, la unidad de interacción de maestro/detalle es una composición de dos o más unidades de interacción más sencillas. A nivel de diseño e implementación la unidad de interacción de maestro/detalle puede representarse, del mismo modo, como un componente compuesto a partir de componentes reusables más sencillos.

La Figura 4.23 muestra la implementación propuesta para el ejemplo considerado en el apartado anterior sobre plataforma Windows.

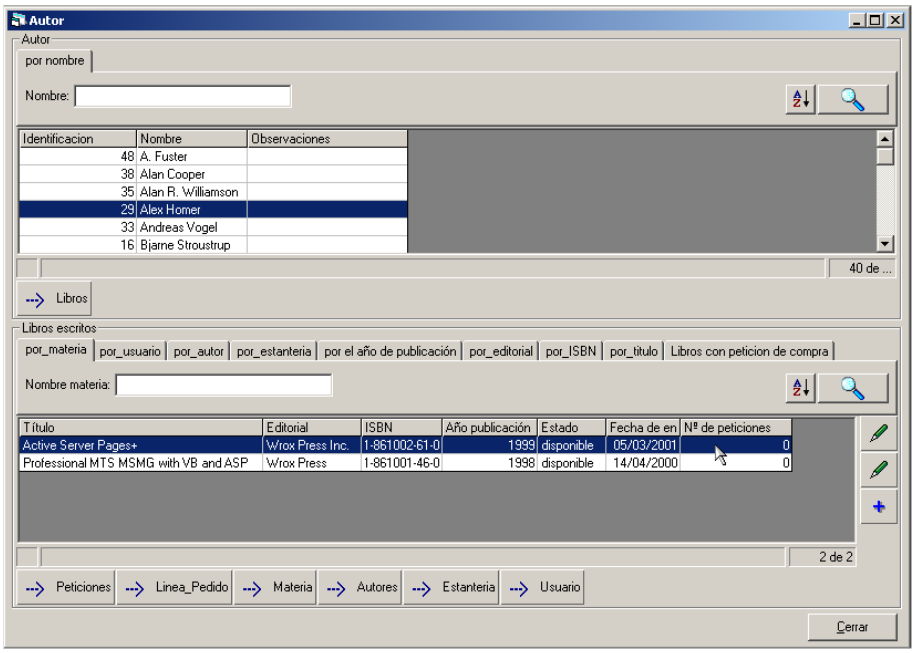


Figura 4.23: Ejemplo de implementación de UI de Maestro/Detalle.

4.2.4. Nivel 3: Patrones elementales

Los patrones elementales se emplean como los *ladrillos primitivos* (piezas base) para componer las unidades de interacción.

En la presente sección se describen los siguientes once patrones:

1. Introducción
2. Selección definida
3. Información complementaria
4. Dependencia
5. Recuperación de estado
6. Agrupación de argumentos
7. Filtro
8. Criterio de ordenación
9. Conjunto de visualización
10. Acciones
11. Navegación

El ámbito de aplicación de estos patrones es el siguiente:

- Los patrones de introducción, selección definida e información complementaria pueden ser aplicados a argumentos de servicio o variables de filtro.
- Los patrones de dependencia, recuperación de estado y agrupación de argumentos se aplican en el marco de una unidad de interacción de servicio.
- Filtros y criterios de ordenación se aplican sobre unidades de interacción de población.
- Por último, conjuntos de visualización, acciones y navegación son aplicables tanto a unidades de interacción de instancia como de población.

3.1 Introducción

Nombre

Introducción / *Introduction*

Descripción

El patrón de introducción captura los aspectos relevantes a los tipos de datos que van a ser introducidos por el usuario. El sistema puede facilitar la entrada de datos dirigiéndola (con mensajes de ayuda y ejemplos), o restringiéndola (con mensajes de error, máscaras de edición o rangos de valores validos). De este modo, disponemos de un primer filtrado de los datos conforme el usuario los va introduciendo, lo cual redunda en un menor número de errores en los datos de entrada.

Definido en

Modelo.

Aplicado a

Argumentos dato-valuados, atributos y variables de filtro objeto-valuadas.

Propiedad:	**Descripción**
Alias	Nombre visible para el usuario.
Máscara de edición	La máscara de edición guiará la entrada, impidiendo la introducción de caracteres no validos.
Rango de valores	Si el conjunto de valores de entrada está acotado dentro de un rango de valores puede indicarse los valores máximo y mínimos que el valor puede tomar. Sólo tiene sentido para datos numéricos.
Valor por defecto	Valor por defecto que tomará el argumento.
Tamaño máximo	Longitud máxima en caracteres que puede llegar a ocupar el campo de entrada.
Acepta nulos	Sí/No. Indica si se permite que el campo quede sin valor asignado.
Mensaje de error	Texto que se mostrará al usuario cuando el valor del argumento no sea valido y se le inste a modificarlo.
Mensaje de ayuda	Mensaje de ayuda Texto que contiene instrucciones para indicar al usuario que dato se le está requiriendo en este momento.

Tabla 4.2: Información para la especificación del patrón de introducción.

Especificación

La información a recoger es la siguiente: el nombre de la instancia del patrón[6], máscara de edición, rango de valores validos, valor por defecto, mensaje de ayuda y mensaje de validación. La tabla 4.2 describe cada uno de los elementos.

Símbolo:	**Semántica**
x	Se acepta cualquier carácter.
X	Se acepta cualquier carácter en mayúsculas.
9	Se acepta sólo un dígito.
#	Se acepta cualquier carácter numérico junto con los caracteres '+' y '-'.
a	Se acepta cualquier carácter alfanumérico.
A	Se acepta cualquier carácter alfanumérico en mayúsculas.
\	Carácter de escape. El carácter que sigue a '\' se considerará como un literal.
: /	Resto de caracteres, se asumen como literales que se mostrarán tal cual, para definir delimitadores.

Tabla 4.3: Convenciones para definición de máscaras.

Máscara:	**Campo vacío**	**Campo válido**
99/99/9999	__/__/____	15/01/2003
#999,99 EUR	____,__ EUR	+123,45 EUR
(999) 999-999	(___) ___-___	(902) 405-504
AA.9999	__.____	CD.1412
Aaa\#xX	___#__	Ac8#pK

Tabla 4.4: Ejemplos de definición de máscaras.

Propiedad Máscara

Las máscaras de edición facilitan la introducción de datos al usuario. Para especificar máscaras de edición en Just-UI se hace necesario definir la sintaxis del lenguaje de definición de máscaras a emplear.

Tradicionalmente, las máscaras de edición se expresan mediante una cadena de caracteres que representa restricciones sobre los caracteres de entrada. Se propone un subconjunto básico de lexemas usados frecuente en diversos lenguajes de especificación de máscaras. Se pretende de este modo, facilitar el aprendizaje al analista y de otra parte simplificar la traducción a diversas plataformas.

[6]Este nombre sirve de identificador para el analista. De este modo, se facilita el reuso de este patrón en la especificación, pudiendo referir a la instancia del patrón por este nombre.

Los códigos propuestos para definición de máscaras aparecen en la Tabla 4.3.

Junto con la máscara de edición, suele definirse un carácter especial, *el carácter de blanco*, que se empleará para visualizar en pantalla los huecos pendientes de ser completados por el usuario. Normalmente se emplea el carácter de espacio ' ' o el carácter de subrayado '_'. Como es muy poco frecuente el uso de otro carácter de blanco distinto del subrayado, es corriente tomar un valor por defecto para el carácter de blanco en lugar de forzar a declararlo en cada máscara.

Por último, la Tabla 4.4 muestra algunos ejemplos de especificación de máscaras. En la segunda columna (aspecto del campo vacío) puede observarse como el formato del campo sirve de retroalimentación al usuario para proceder a completarlo.

Semántica asociada

El patrón de introducción es útil allí donde el usuario ha de introducir datos al sistema. Los argumentos suelen incluir información como el tipo o si el argumento admite nulos o no. Sin embargo, asociar a un argumento un patrón de introducción permite detallar mucho más aspectos de interfaz que pueden ser aprovechados posteriormente para lograr una mayor calidad de la interfaz de usuario del sistema.

En principio podría pensarse que sólo los argumentos de los servicios podrán beneficiarse de la aplicación de este tipo de patrón. En última instancia esta afirmación es cierta, sin embargo, aplicar este tipo de patrón también a atributos tiene sus ventajas. Si un analista define un patrón de introducción destinado a modelar la captura de teléfonos y aplica el patrón construido a un atributo denominado `telefono` en una clase denominada `cliente`, resultará que podemos inferir[7] que todos los argumentos de servicios de esta clase `cliente` destinados a cambiar el teléfono se verán afectados por este mismo patrón.

Esto es posible, gracias a que el Modelo Funcional de OO-Method establece correspondencias unívocas entre argumento y atributo cuando la caracterización del atributo es de estado como sucede en el caso planteado.

De este modo, el esfuerzo de especificación por parte del analista se reduce considerablemente, pasando de tener que aplicar el patrón a cada argumento que modifique el teléfono a tener sólo que hacerlo para los atributos de la clase gracias al proceso de inferencia reseñado.

Finalmente comentaremos que si un argumento dispone de:

[7] La inferencia es un proceso destinado a completar modelos con información faltante para proceder a la generación de código en ambientes de prototipado rápido. Para una explicación detallada del proceso de inferencia consulte el capítulo 6.

- un patrón inferido a partir de su atributo relacionado y
- un patrón aplicado directamente,

el que tiene prioridad es este último, ya que, es el criterio del analista el que tiene preferencia por ser más especifico.

Fundamento

El usuario, como humano que es, comete errores cuando tiene que facilitar información al sistema. Si el analista es capaz de anticiparse a los errores previniendo su aparición obtendremos sistemas más usables. El patrón de introducción al restringir los valores permitidos de entrada, ayuda conseguir este fin.

Guía de aplicación

Este patrón es aplicable cuando el usuario deba proporcionar información al sistema, introduciéndola a través de un teclado, mediante lenguaje natural, o cualquier otro medio de comunicación.

Meta-modelo

La Figura 4.24 muestra el meta-modelo para el patrón de introducción.

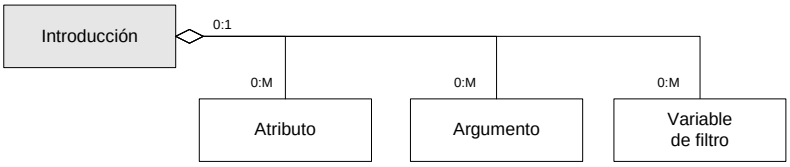


Figura 4.24: Meta-modelo del patrón de introducción.

Un patrón de introducción se crea instanciando la clase Introducción. Las propiedades máscara de edición, rango, etc. son representados mediante atributos en la clase. Una vez se ha definido el patrón, este puede ser aplicado a atributos de clases, argumentos dato-valuados o variables de filtro dato-valuadas.

Información recogida:

Nombre del patrón, máscara de edición, rango de valores validos, valor por defecto, mensaje de ayuda y validación.

Ejemplo de problema

Una compañía de seguros dispone de un servicio de asistencia telefónica. Se desea evitar, en la medida de lo posible, errores en la introducción de

datos del cliente. En particular, se ha determinado como de vital importancia que el teléfono del cliente sea correcto, puesto que es lo que permite ponerse en contacto con él. Los teléfonos válidos tienen nueve dígitos y siguen el formato: 000-000-000. Para satisfacer este requisito, podemos emplear el patrón de introducción para limitar los valores validos para aquellos argumentos destinados a recoger números de teléfonos.

Captura en una herramienta CASE de modelado

La especificación se lleva a cabo instanciando un nuevo patrón de introducción. En la Figura 4.25 se muestra la primera parte del proceso donde se proporciona un nombre (I_Telefono) y el mensaje de ayuda. La Figura 4.26 muestra la definición de la máscara empleando ###-###-### como formato para los números de teléfono y un mensaje de validación que será mostrado al usuario cuando la entrada sea incorrecta.

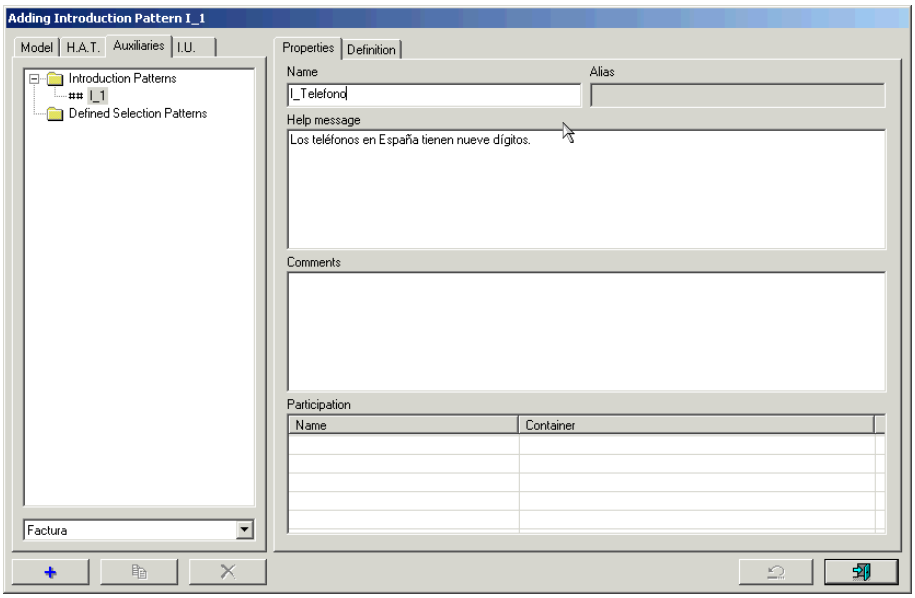


Figura 4.25: Especificación de un patrón de introducción 1/2.

Ejemplo de implementación

La Figura 4.27 muestra una implementación de una unidad de interacción de servicio con diversos argumentos. En particular, los argumentos teléfono y fax implementan el patrón de introducción I_Telefono (previamente definido en la sección anterior). Dichos campos sólo pueden contener dígitos y cada teléfono ha de contener exactamente nueve dígitos. De este modo, la interfaz de usuario ayuda al usuario a reducir los errores en la introducción de datos. Como ejemplo de ayuda contextual, véase como en la figura aparece como

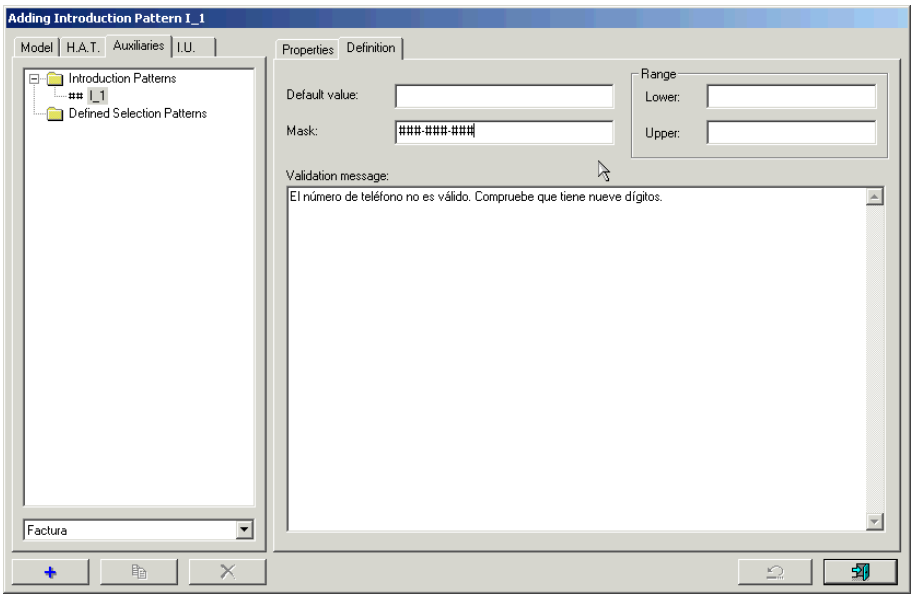


Figura 4.26: Especificación de un patrón de introducción 2/2.

mensaje emergente (*Tooltip*) el mensaje de ayuda especificado para este patrón.

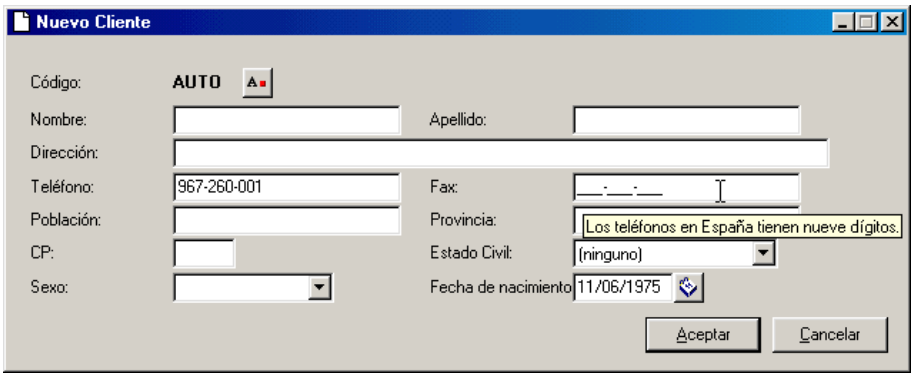


Figura 4.27: Ejemplo de implementación del patrón de introducción.

3.2 Selección definida

Nombre

Selección definida / *Defined Selection*

Descripción

Permite al analista proporcionar por enumeración los elementos válidos para un argumento. El conjunto definido de este modo, se comporta como un tipo enumerado declarado por el analista que restringe el valor de los argumentos que lo emplean a los valores especificados.

Definido en

Modelo.

Aplicado a

Argumentos dato-valuados de servicios, atributos y variables de filtro dato-valuadas.

Especificación

Nombre del patrón, y conjunto de valores. Cada valor consta de una etiqueta y un código. Uno de los elementos del conjunto puede marcarse como valor por defecto. La tabla 4.5 describe cada uno de las propiedades consideradas.

Propiedad:	**Descripción**
Alias	Nombre visible para el usuario.
Definición del conjunto	Enumeración de los elementos del conjunto.
Valor por defecto	Valor por defecto que tomará el argumento.
Selección simple / múltiple	▪ Mínimo a seleccionar (normalmente: 0 o 1) ▪ Máximo a seleccionar (normalmente: 1) Acepta nulos: SI $\longrightarrow$ Mínimo=0 / NO $\longrightarrow$ Mínimo>0
Mensaje de error	Texto que se mostrará al usuario cuando la selección realizada no sea válida, bien por que no se cumple la cardinalidad mínima o máxima.
Mensaje de ayuda	Texto que contiene instrucciones para indicar al usuario que dato se le está requiriendo en este momento.

Tabla 4.5: Información para la especificación del patrón de introducción.

Semántica asociada

La enumeración de un conjunto de valores fuerza a que el argumento sólo pueda tomar uno de los valores indicados. De este modo, siguiendo uno de los principios básicos en interfaces de usuario, se consigue evitar errores cometidos por el usuario al introducir datos. Al igual que el patrón de introducción, se permite la aplicación del patrón sobre atributos apoyado sobre el proceso de inferencia con los mismos motivos: mínimo esfuerzo de especificación por parte del analista.

Fundamento

La motivación principal para usar este patrón es la de prevenir los errores por parte del usuario en la introducción de datos. El patrón de selección definida proporciona una lista cerrada de valores, por tanto, el usuario no puede indicar otros valores diferentes a los esperados.

Guía de aplicación

Cuando el analista detecte que un determinado argumento tiene un conjunto cerrado y conocido de valores, puede aplicar este patrón y definir dichos valores por medio de enumeración.

Meta-modelo

La Figura 4.28 muestra el meta-modelo para el patrón de selección defini-

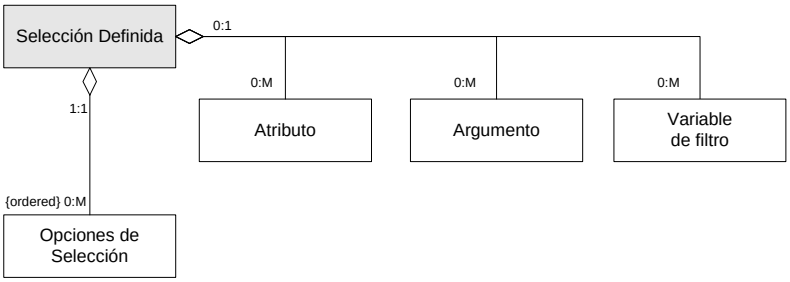


Figura 4.28: Meta-modelo del patrón de selección definida.

da. Un patrón de selección definida se crea instanciando la clase Selección Definida. Las opciones de selección están representadas por la clase Opciones de selección. Una vez se ha definido el patrón, este puede ser aplicado a atributos de clases, argumentos dato-valuados o variables de filtro dato-valuadas.

Ejemplo de problema

Para realizar pólizas de seguros, la compañía de seguros requiere información personal de los clientes tales como el estado civil o el sexo. El estado civil es un atributo cuyos valores son enumerados y bien conocidos (a saber:

soltero, casado, viudo, separado, divorciado). Por lo tanto, una medida para evitar errores del personal de la compañía al recoger datos de clientes es obligar a tomar uno de estos valores predefinidos y prohibir cualquier otro valor.

Captura en una herramienta CASE de modelado

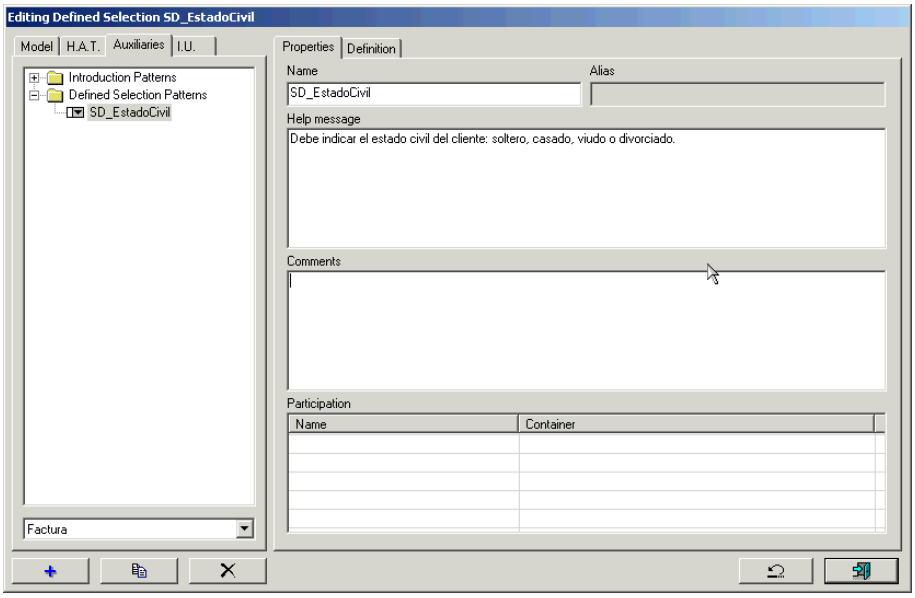


Figura 4.29: Especificación de un patrón de selección definida 1/2.

Podemos definir un nuevo patrón de selección definida para gestionar los valores del atributo estado civil. El patrón SD_EstadoCivil ha sido definido tal y como muestra la Figura 4.29. Como valores de selección se necesitan los siguientes:

- soltero (valor por defecto),
- casado,
- viudo y
- divorciado.

La Figura 4.30 muestra una pantalla de la herramienta CASE donde se ha definido el conjunto de valores permitidos.

Ejemplo de implementación

La Figura 4.31 muestra una pantalla generada para una unidad de interacción de servicio para la plataforma Windows. En ella, se aprecia que el patrón

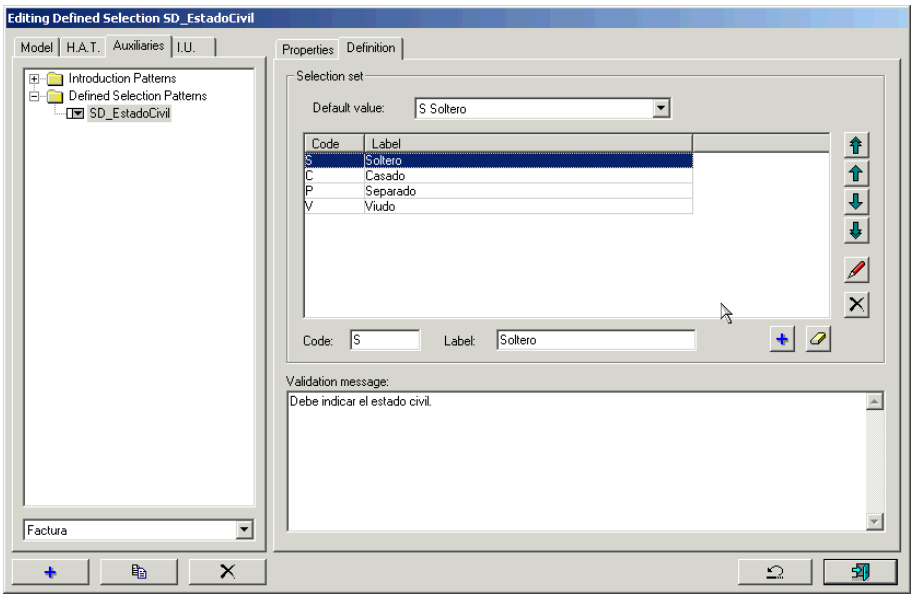


Figura 4.30: Especificación de un patrón de selección definida 2/2.

de selección definida asociado al argumento estado civil ha sido implementado usando un control ComboBox.

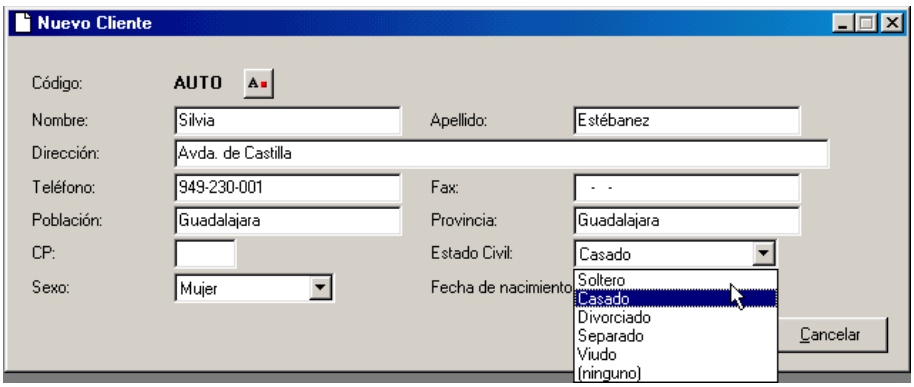


Figura 4.31: Ejemplo de implementación del patrón de selección definida.

3.3 Información complementaria

Nombre

Información complementaria / *Supplementary Information*

Descripción

Captura información adicional para ser mostrada al usuario para complementar al *OID*[8] de un objeto. Esta información de retroalimentación (*feedback*) ayuda al usuario a confirmar que la selección realizada fue la correcta.

Definido en

Argumentos objeto-valuados de servicio y variables de filtro objeto-valuadas.

Aplicado a

Argumentos de servicio de tipo objeto-valuado, variables de filtro de tipo objeto-valuado y clases (como información por defecto para los argumentos de tipo objeto-valuado cuyo dominio sea la clase considerada).

Especificación

La información complementaria no es más que la aplicación a un caso particular del patrón conjunto de visualización.

Meta-modelo

El patrón de información complementaria es una aplicación particular del patrón conjunto de visualización. Para una explicación detallada véase el meta-modelo asociado al patrón conjunto de visualización en la página 182.

Semántica asociada

La rápida retroalimentación es un principio básico de la teoría de interfaces de usuario. Este principio motiva la aparición de este patrón. La información adicional proporcionada junto al identificador de un objeto (OID) permite al usuario final asegurarse o retractarse de una selección o de la introducción de un código previamente memorizado que puede ser correcto o no. La rápida retroalimentación asegura al usuario en sus acciones y le permite trabajar de manera más rápida.

Fundamento

El objetivo es proporcionar retroalimentación inmediata al usuario [IBM92].

Guía de aplicación

Cuando el mecanismo de identificación de objetos empleado no sea sufi-

[8] OID: *Object identifier.* Secuencia de atributos que designan de modo unívoco cada objeto de la clase considerada.

ciente para los usuarios, debemos completarlo aplicando este patrón, añadiendo la información suficiente para que el usuario pueda reconocer el objeto.

Ejemplo de problema

En una compañía de seguros, los usuarios se identifican mediante un código de cliente. Este código es difícil de recordar y propenso a cometer errores en su introducción. Cada vez que se introduce un código debería aparecer como respuesta inmediata el nombre del cliente asociado a ese código. De este modo, el operador del sistema puede estar seguro de haber introducido el código adecuado.

Captura en una herramienta CASE de modelado

Un patrón de información complementaria consiste en la aplicación de un conjunto de visualización a un argumento objeto-valuado. Por tanto comenzaremos definiendo un conjunto de visualización (*véase Figura 4.32*) denominado DS_ClienteInfo.

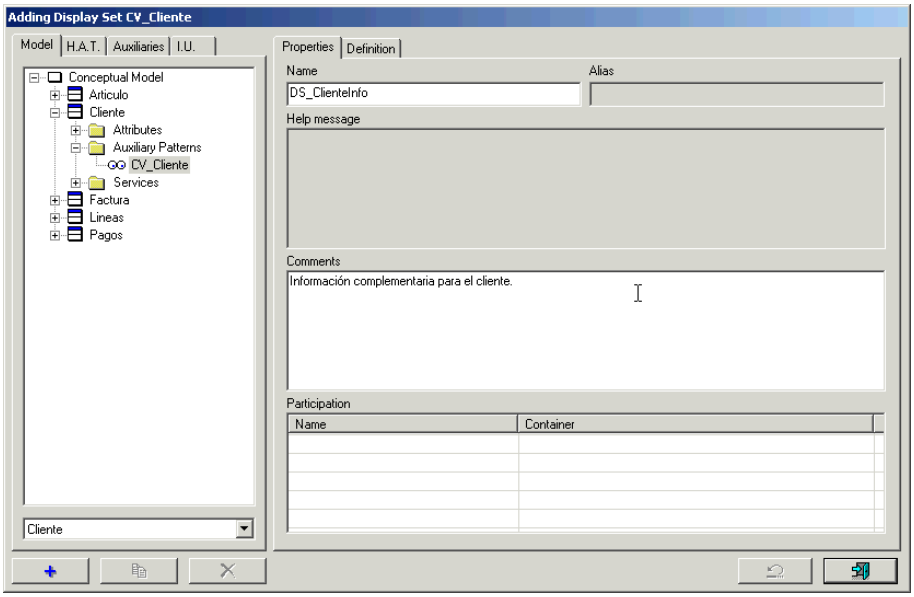


Figura 4.32: Especificación de un patrón de información complementaria 1/3.

Como información complementaria, estamos interesados en emplear el nombre y el apellido del cliente seleccionado. La Figura 4.33 muestra como definir los elementos dentro del conjunto de visualización.

Una vez completada la definición del conjunto de visualización, sólo resta aplicar el patrón de información complementaria al ejemplo considerado. La Figura 4.34 muestra como aplicar el patrón de información complementaria

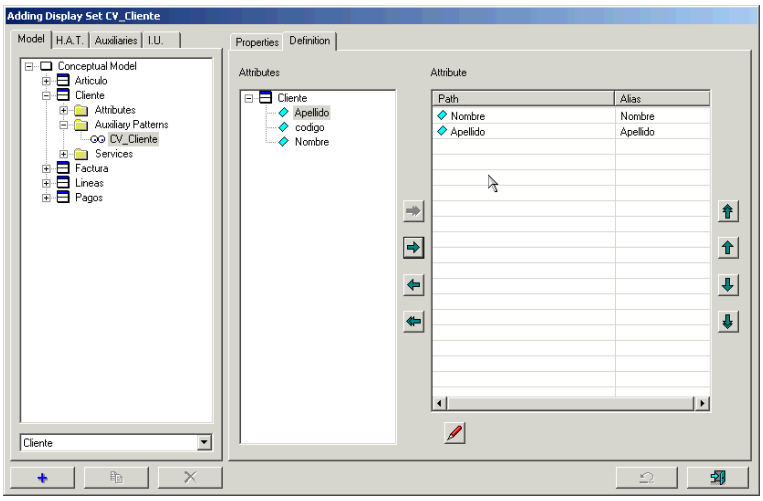


Figura 4.33: Especificación de un patrón de información complementaria 2/3.

sobre un argumento de tipo cliente del servicio Delete_Instance de la clase Cliente.

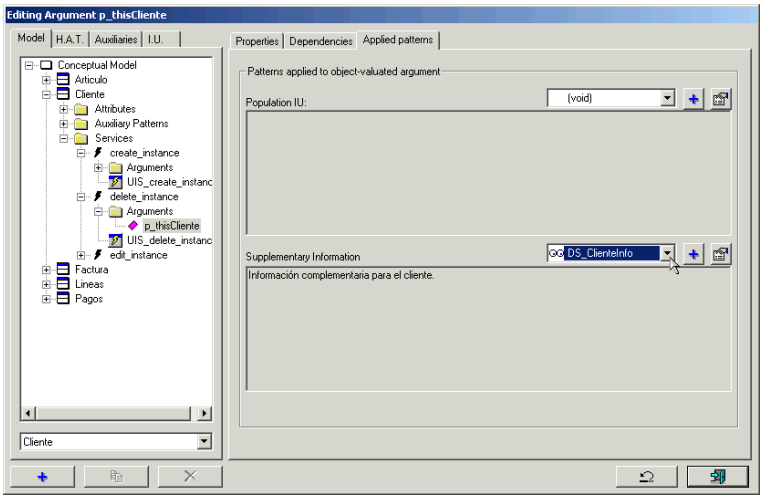


Figura 4.34: Especificación de un patrón de información complementaria 3/3.

Ejemplo de implementación

Una implementación sobre plataforma Windows se ilustra en la Figura 4.35. En dicha figura se muestra la implementación para la introducción de una referencia a un objeto de tipo Cliente.

Figura 4.35: Ejemplo de implementación del patrón de información complementaria.

Cuando el usuario proporciona un cliente, bien introduciendo un código de cliente, o bien mediante el mecanismo de selección implementado por el botón conteniendo una *lupa*, a la derecha aparecerán el nombre y el apellido del cliente seleccionado (*véase Figura 4.36*). Esto proporciona una retroalimentación (*feedback*) clara al usuario de la selección realizada.

Figura 4.36: Ejemplo de implementación del patrón de información complementaria 2/3.

Por contra, si el cliente suministrado no existe, el usuario recibirá un aviso en forma de mensaje de alerta (*véase Figura 4.37*).

Figura 4.37: Ejemplo de implementación del patrón de información complementaria 3/3.

3.4 Dependencia

Nombre

Dependencia / *Dependency*

Descripción

Los datos que el usuario debe suministrar al sistema no siempre son independientes uno de otro, en especial cuando los datos están relacionados entre sí. Puede ocurrir que los valores de unos queden determinados por los valores tomados por otros. Si el sistema puede modelar dichas restricciones puede impedir al usuario introducir combinaciones de información inválida. El patrón de dependencia surge cuando la determinación del valor de uno o varios argumentos de entrada tienen como efecto la variación del estado o las restricciones sobre otros datos solicitados por el sistema. Se dice entonces que, el valor de un dato depende o está en función de otros.

El patrón de dependencia describe las relaciones de dependencia entre argumentos de un servicio. Permite especificar la respuesta del sistema al usuario durante el transcurso de un diálogo de introducción de datos.

Definido en

UI de servicio y los respectivos argumentos del servicio asociado.

Aplicado a

UI de servicio y los respectivos argumentos del servicio asociado.

Especificación

La especificación se lleva a cabo empleado reglas ECA (Evento-Condición-Acción) que describen las relaciones de dependencia entre argumentos del servicio. La especificación completa puede consultarse en el *Anexo D. Especificación de reglas ECA* en la pág. 325.

Meta-modelo

La Figura 4.38 muestra el meta-modelo para el patrón de dependencia. Un patrón de dependencia está compuesto por una serie de reglas evento-condición-acción (ECA). La clase ECA representa cada una de estas reglas. Las reglas ECA son definidas siguiendo un orden de evaluación sobre UI de Servicio y aplicadas sobre las instancias de Argumento. Las reglas tienen efecto sobre los argumentos en una UI de Servicio dada.

Semántica asociada

Las reglas ECA asociadas al patrón de dependencia son evaluadas en función de los eventos disparados en la interfaz de usuario. El orden de evaluación de la reglas es importante (diferentes órdenes pueden producir resultados di-

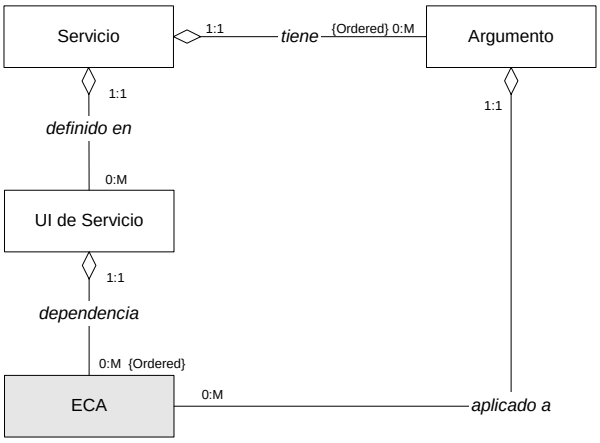


Figura 4.38: Meta-modelo del patrón de dependencia.

ferentes). El *Anexo D. Especificación de reglas ECA* en la pág. 325 describe con detalle esta semántica.

Fundamento

El patrón de dependencia modela el comportamiento dinámico de los AIO (representantes de los argumentos de servicio) durante la interacción con el usuario. Las reglas de dependencia precisan como la interacción sobre un AIO modifica el estado de otros AIO.

Guía de aplicación

Este patrón puede ser aplicado en el marco de un servicio, cuando el analista detecte interdependencias entre argumentos del servicio.

Ejemplo de problema

Al realizar una pequeña encuesta, los clientes son preguntados si tienen un trabajo remunerado. Si la respuesta es afirmativa se procede a preguntar por los ingresos brutos. Si la respuesta es negativa, se asumirá que no se perciben ingresos. La signatura del servicio se muestra en la Tabla 4.6

Captura en una herramienta CASE de modelado

Cada argumento definido en un servicio tiene asociado un y sólo un AIO asociado a él. Este AIO (a nivel conceptual) representa al argumento frente al usuario. Cuando el sistema es implementado, los AIO son traducidos a CIO o controles (los controles con los que realmente interacciona el usuario).

Este nivel de abstración proporcionado por los AIO permite especificar reglas sobre los AIO asociados a cada argumento. Los AIO tienen atributos como Active (indica si el usuario puede modificar su valor o no), Value (contiene el

```
class Cliente : service encuesta alias 'Encuesta'(
      pCliente as Cliente alias 'Cliente',
      pTrabaja as bool alias 'Trabaja',
      pSalario as real alias 'Sueldo' )
```

Tabla 4.6: Signatura del servicio encuesta.

valor actualmente asignado al argumento) y eventos de interfaz como SetValue (este evento se lanza cada vez que el usuario modifica su valor) o SetActive (que se lanza cada vez que el AIO es activado o desactivado).

El problema planteado en el apartado anterior puede expresarse con el siguiente par de reglas (*véase Tabla 4.7*). Cada vez que ocurre un cambio en el

```
On Event pTrabaja.SetValue()
      If pTrabaja.value=true Then
           pSalario.value=5000; pSalario.active=true
      If pTrabaja.value=false Then
           pSalario.value=0; pSalario.active=false
```

Tabla 4.7: Reglas de dependencia para el servicio encuesta.

valor asociado al AIO `pTrabaja` (es decir, cambia su valor), las reglas son evaluadas para comprobar su condición. Si la condición es satisfecha, la regla es ejecutada.

Las reglas de dependencia pueden ser declaradas en Oliva Nova Modeler® usando la ventana mostrada en la Figura 4.39. Desde está ventana es posible crear, destruir y reordenar las reglas de dependencia. Tras hacer doble click sobre una regla de dependencia, la ventana de edición de reglas aparecerá (*véase Figura 4.40*). Esta segunda ventana permite la definición de las fórmulas de la regla en sus tres vertientes: evento, condición y acciones asociadas.

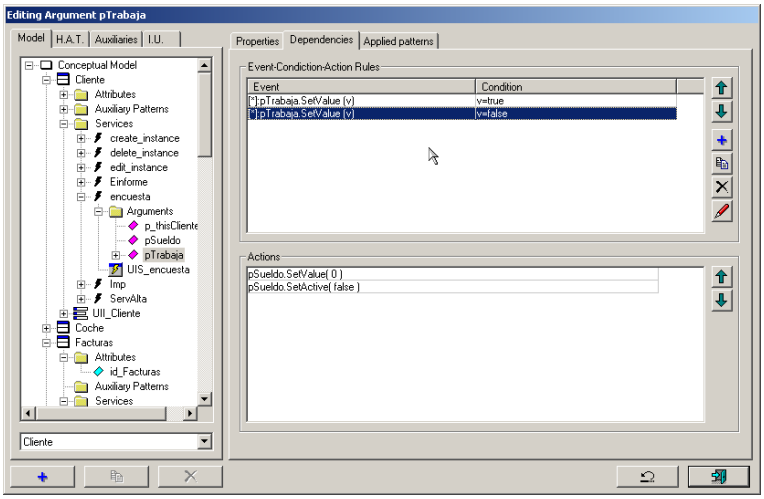


Figura 4.39: Especificación de una regla de dependencia 1/2.

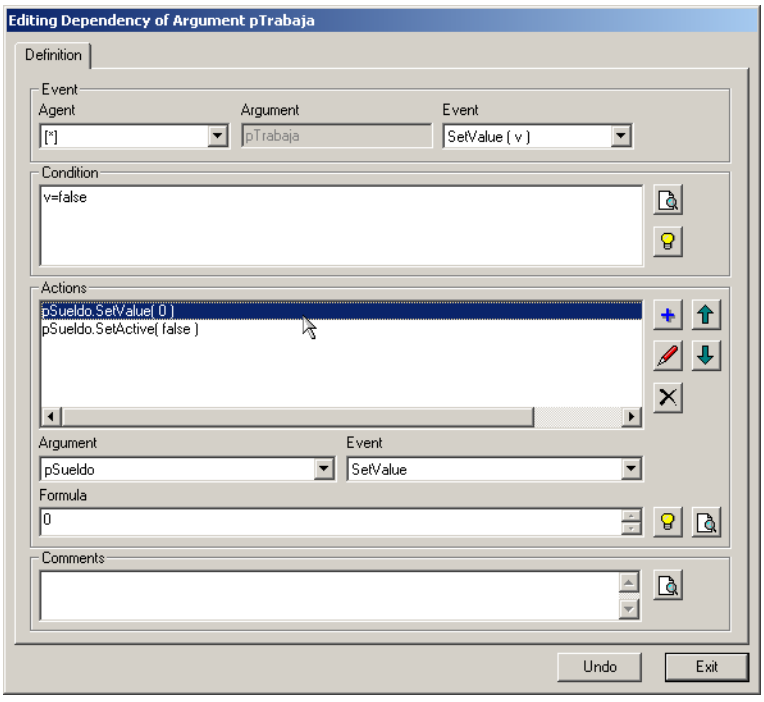


Figura 4.40: Especificación de una regla de dependencia 2/2.

Ejemplo de implementación

La habitual unidad de interacción de servicio (*véase Figura 4.41*) ha sido traducida para dar cuenta de la implementación del servicio Encuesta.

Sin embargo cuando el usuario marca a cierto el campo *Trabaja* (*véase Figura 4.42*) las reglas de dependencia son evaluadas provocando la ejecución de una de ellas y dando como resultado que el campo *Sueldo* adquiera el valor *"5000"*.

El patrón de dependencia puede ser traducido como código que forma parte de la lógica de la interfaz de usuario. Su ejecución es siempre disparada por eventos provocados en la interfaz de usuario.

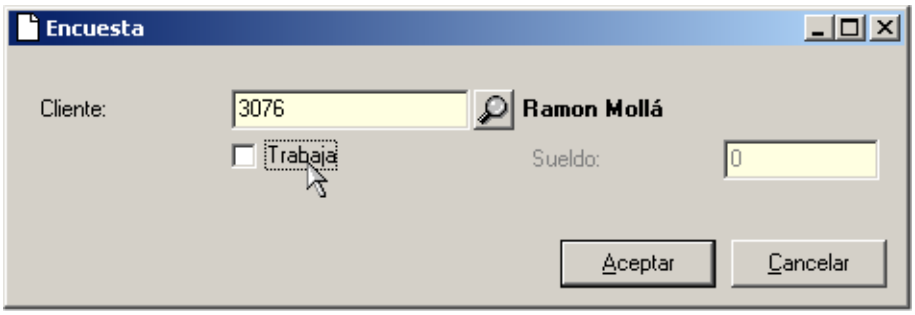


Figura 4.41: Aspecto antes del disparo de la regla de dependencia.

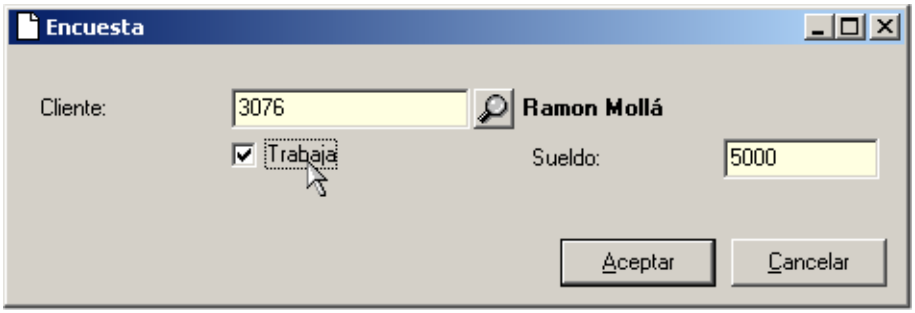


Figura 4.42: Aspecto después del disparo de la regla de dependencia.

3.5 Recuperación de estado

Nombre

Recuperación de estado / *Status Recovery*

Descripción

Permite inicializar argumentos de un servicio, en base al conocimiento del objeto que va a sufrir el servicio y las relaciones entre los argumentos del servicio y los atributos de la clase implicada. Por ejemplo, al modificar los datos de un cliente, una vez que se han introducido la identificación de dicho cliente, puede recuperarse como valor del argumento teléfono el valor actual del atributo teléfono para el objeto seleccionado. Una vez modificado y lanzado el servicio, el nuevo teléfono suplantará al anterior.

Este patrón puede ser visto como un conjunto implícito de reglas de dependencia de la forma ilustrada en la Tabla 4.8.

```
On Event <This>.SetValue()
      If true Then
           <Arg1>.value = <Atrib1>;
           <Arg2>.value = <Atrib2>;
                     ⋮
           <Argn>.value = <Atribn>
```

Tabla 4.8: Plantilla de reglas de dependencia para especificar recuperación de estado.

Donde tenemos que:

- `<This>` representa el nombre el parámetro que actúa como objeto receptor del mensaje.
- Cada `<Arg`$_i$`>` representa al argumento i-ésimo del servicio.
- Cada `<Atrib`$_i$`>` representa el atributo i-ésimo del objeto referenciado por `This`.
- Para poder aplicar el patrón de recuperación de estado ha de cumplirse que para cada `<Arg`$_i$`>` existe una evaluación de estado en el modelo funcional de la forma `<Atrib`$_i$`>=<pArg`$_i$`>` que permita recuperar el valor de un argumento a partir del valor de un atributo.

Definido en

No procede.

Aplicado a

UI de servicio.

Especificación

No procede. El propio modelo OO-Method en base a la definición de servicios y sus consecuencias definidas en el Modelo Funcional basta para deducir qué argumentos están relacionados con qué atributos (todas las evaluaciones de la forma: $< atributo = argumento >$) para recuperar su estado al fijar el argumento que representa al objeto servidor.

Meta-modelo

No necesita especificación adicional sobre el modelo. Las relaciones existentes en el Modelo Funcional entre servicio, argumento, y atributo recogen toda la semántica necesaria.

Semántica asociada

Al proporcionar el servicio y el objeto servidor, todos los argumentos del servicio que tengan evaluaciones definidas de la forma: $< atributo = argumento >$ se inicializan con el valor de los atributos asociados según el valor actual del objeto servidor.

Fundamento

Los servicios de modificación permiten cambiar las propiedades de los objetos. Lo normal es que un usuario ante un cambio decida alterar sólo una parte de las propiedades. El hecho de inicializar todos los argumentos con los valores previos tiene tres ventajas claras:

1. muestra el estado del objeto al presentar los valores actuales para las propiedades,
2. permite al usuario cerciorarse de que el objeto que pretende cambiar es realmente el que ha seleccionado y
3. permite al usuario modificar sólo aquellas propiedades en las que realmente está interesado.

Guía de aplicación

No procede. Como se ha dicho previamente, su detección y aplicación es llevada a cabo de modo automático por los traductores de código.

Ejemplo de problema

Cuando un cliente cambia de domicilio o especialmente de teléfono, comunica a la compañía tal situación para que la compañía pueda seguir en contacto

con el cliente. Ante una solicitud de cambio de datos de un cliente, el usuario necesita contrastar los datos antiguos con los que le indica el cliente antes de proceder a cambiarlos. De este modo se evitan cambios innecesarios a información que ya es correcta.

Captura en una herramienta CASE de modelado

Debido al carácter totalmente implícito del patrón, no se requiere ningún esfuerzo adicional del analista para especificar este patrón.

Ejemplo de implementación

El ejemplo ilustrado (*véase Figura 4.43*) muestra una implementación de una unidad de servicio para cambiar los datos de un cliente. El efecto del patrón de recuperación de estado se produce cuando el usuario identifica el objeto que se desea modificar. Nótese que habitualmente este argumento siempre es dispuesto en primer lugar. En ese instante, la aplicación utiliza la referencia al objeto para recuperar el valor de sus atributos y los muestra en los campos de edición (*véase Figura 4.44*). A partir de este momento el usuario puede consultar la información que se le presenta y puede modificar cualquier dato, tal y como acostumbra en cualquier otra Unidad de Interacción de Servicio.

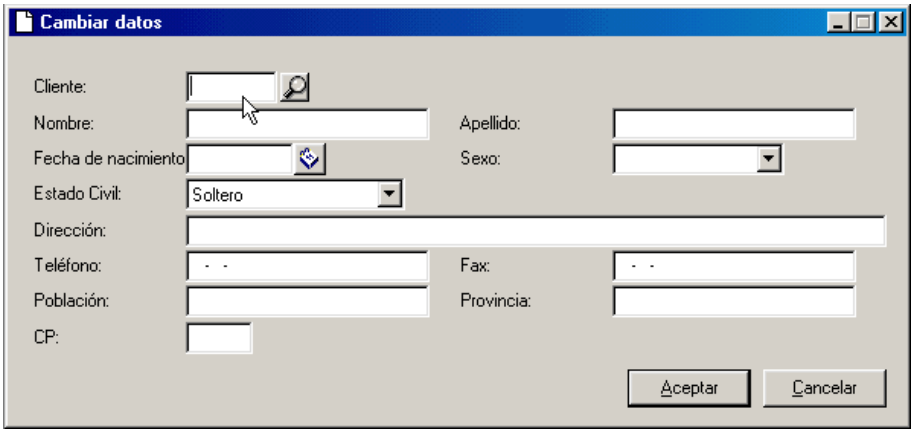


Figura 4.43: Aspecto previo a la recuperación de estado.

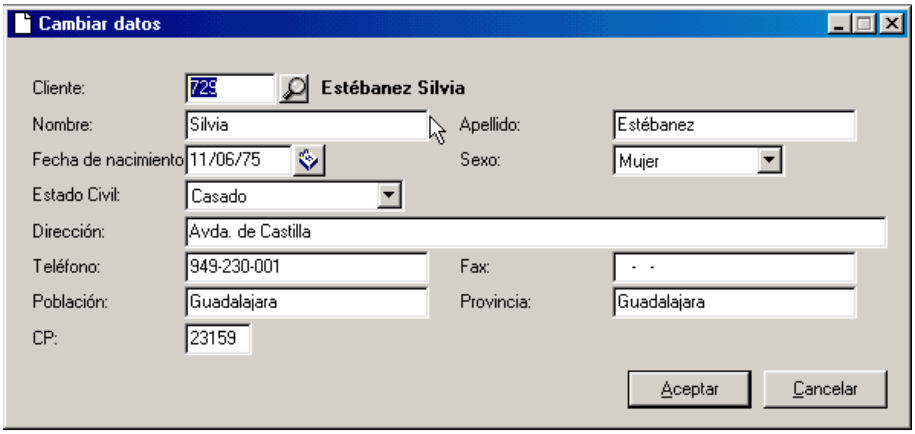


Figura 4.44: Aspecto tras la recuperación de estado.

3.6 Agrupación de argumentos

Nombre

Agrupación de argumentos / *Argument Grouping*

Descripción

Captura la agrupación de argumentos de un servicio. Cuando un servicio contiene un número elevado de argumentos se hace necesario ordenar de modo lógico los argumentos a solicitar. El *Principio de Aproximación Gradual* guía de nuevo la esencia de este patrón.

Definido en

UI de servicio y los respectivos argumentos del servicio asociado.

Aplicado a

UI de servicio y los respectivos argumentos del servicio asociado.

Especificación

Se lleva a cabo mediante la agrupación y relación de continencia de grupos y argumentos dentro de grupos. Dada una Unidad de Interacción de Servicio, podrán definirse agrupaciones caracterizadas por:

- Un *alias* o etiqueta del grupo. Esta etiqueta será mostrada al usuario final.
- Una serie de elementos ordenados que conforman los elementos contenidos dentro del grupo. Estos elementos son: *argumentos* de servicio, o bien, de modo recursivo, *agrupaciones*.

De este modo se consigue establecer una clasificación arborescente a los argumentos del servicio. Dicho acceso arborescente, usado adecuadamente por un analista, permite aplicar el *Principio de Aproximación Gradual* cuando el número de argumentos es elevado.

Meta-modelo

La Figura 4.45 muestra el meta-modelo para el patrón de agrupación de argumentos. El patrón de agrupación de argumentos consiste en una serie de Elementos de agrupación definidos sobre una UI de Servicio dada. Cada Elemento de agrupación puede tomar el rol de:

- *elemento agrupador*: contiene una etiqueta (alias) y recursivamente puede contener otros Elementos de agrupación, o bien,
- *argumento*: en las hojas de este árbol de agrupación aparecen las referencias a los Argumentos.

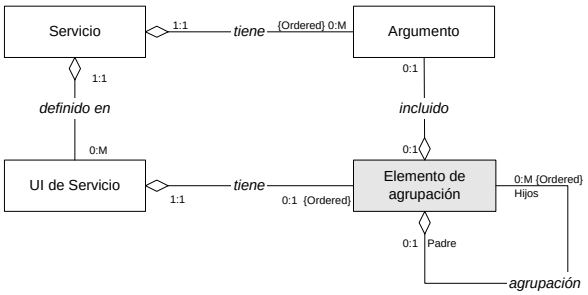


Figura 4.45: Meta-modelo del patrón de agrupación de argumentos.

Semántica asociada

Las agrupaciones definidas empleando este patrón permiten ordenar de un modo lógico y físicamente los argumentos del servicio. Tales agrupaciones condicionan o restringen la disposición de presentación (*presentation layout*) de los CIO en la implementación final.

Fundamento

Cuando el número de argumentos es elevado, es conveniente aplicar técnicas del tipo *«divide y venceras»* tal y como propone el *Principio de Aproximación Gradual*. Una organización en grupos permite al analista organizar los argumentos, simplificando de este modo el aprendizaje de la interfaz por parte del usuario.

Guía de aplicación

Este patrón es apropiado cuando el número de argumentos crece por encima de nueve o diez. Las agrupaciones deben llevarse a cabo buscando un criterio lógico y sencillo para el usuario.

Por contra, no debe abusarse de los niveles de anidación de los grupos, así como evitar grupos con muy pocos elementos (uno o dos) o con demasiados (más de nueve).

Ejemplo de problema

Al crear un nuevo cliente para una compañía de seguros se necesitan multitud de datos del cliente tales como datos personales, dirección postal, dirección de residencia, cuenta de domiciliación, datos de los descendientes, etc. Ante tal volumen de información, sería más cómodo poder agrupar los datos a solicitar en base grupos lógicos de información en vez de solicitarlos todos de una manera lineal y sin agruparlos.

Captura en una herramienta CASE de modelado

Para capturar esta información, crearemos agrupaciones y las dispon-

dremos en una estructura arborescente. Los argumentos del servicio serán añadidos como hojas a este árbol. Para el ejemplo propuesto, el analista crearía las siguientes agrupaciones donde redistribuir los argumentos del servicio:

- Datos Personales
  - Titular
  - Cónyuge
- Dirección
  - Dirección habitual
  - Segundo domicilio
- Cuenta bancaria
  - Dirección de la sucursal
- Estado de salud

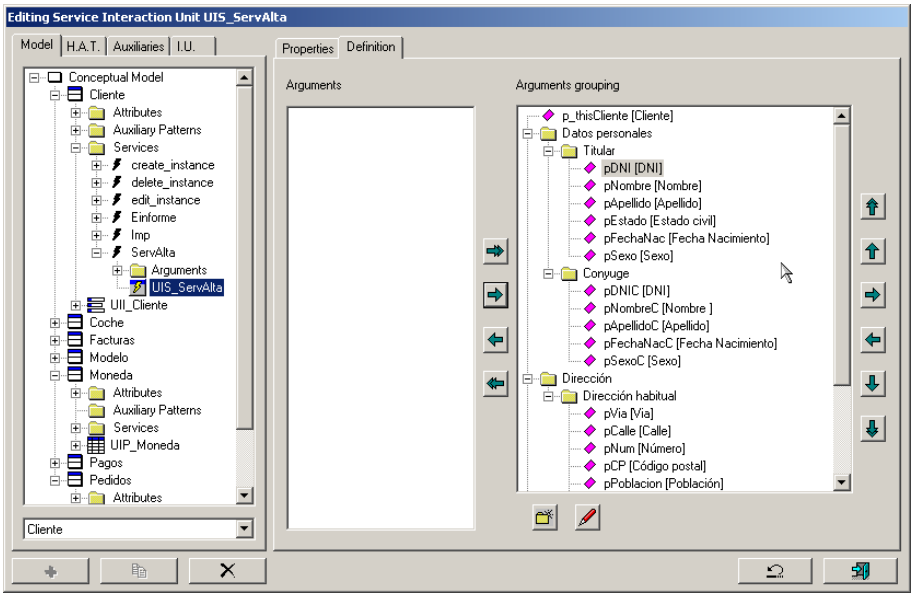


Figura 4.46: Especificación de agrupación de argumentos.

La pantalla de definición de las agrupaciones de argumentos está situada en la ventana de unidad de interacción de servicio. La Figura 4.46 muestra la especificación de las agrupaciones propuestas para agrupar los argumentos del servicio ServAlta de la clase Cliente.

Ejemplo de implementación

Para la plataforma Windows, se ha seguido la siguiente estrategia de traducción (*véase Figura 4.47*):

- Emplear un control de pestañas (*tab control*) para las agrupaciones a primer nivel.
- Emplear marcos (*frame*) para agrupaciones a sucesivos niveles.

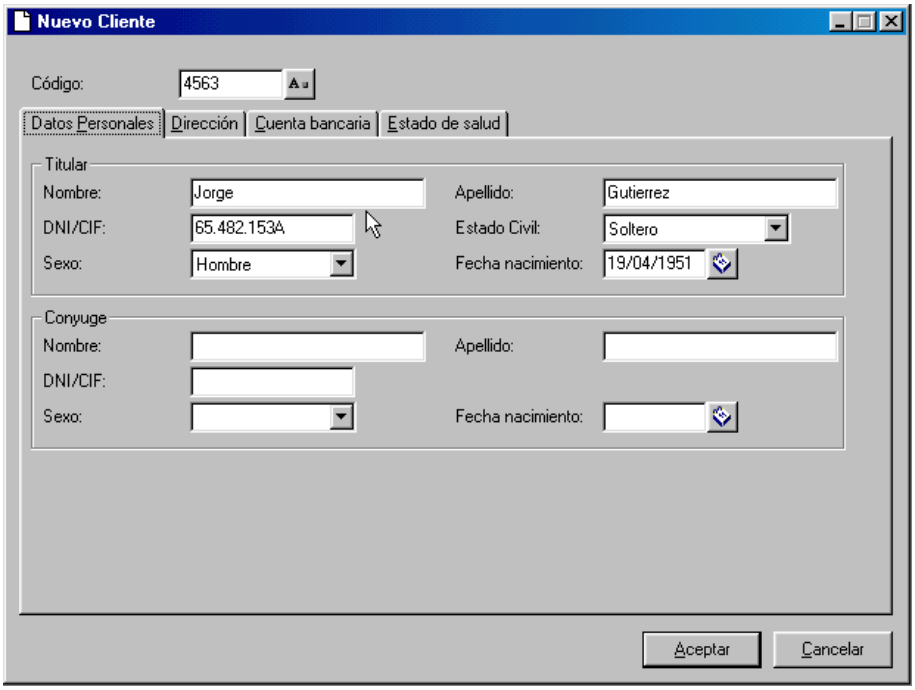


Figura 4.47: Ejemplo de implementación del patrón de agrupación de argumentos.

3.7 Filtro

Nombre

Filtro / *Filter*

Descripción

Un filtro, o criterio de ordenación, expresa una condición de búsqueda que el usuario emplea para localizar información que necesita sobre una población de objetos pertenecientes a una determinada clase.

Definido en

Clase.

Aplicado a

Unidad de Interacción de Población.

Especificación

Un filtro se define como una fórmula bien formada en el lenguaje *OASIS* [Pastor95] en forma de expresión lógica abierta[9]. El apéndice C define la reglas de formación de las fórmulas los filtros.

En ejecución, el usuario debe dar valor a dichas variables antes de invocar un filtro. Esta asignación de valor permite alcanzar una fórmula cerrada que es evaluable a un valor lógico. La fórmula cerrada de filtro es evaluada entonces para cada objeto perteneciente a la población de la clase. Los objetos que satisfacen la condición de filtro constituyen el subconjunto de objetos que serán presentados al usuario.

Meta-modelo

La Figura 4.48 muestra el meta-modelo para el filtro. Un Filtro se define sobre una Clase. Cada filtro tiene una lista ordenada de Variables de filtro que participan en la fórmula del filtro. A su vez, cada Variable de filtro puede tener relación Patrón Auxiliar para aplicar otros patrones de introducción, selección definida o información complementaria. Los filtros enriquecen los componentes UI de Población proporcionando los mecanismos de búsqueda.

Semántica asociada

La semántica asociada a un filtro está detallada con profusión en el apéndice C, pág. 319.

Fundamento

Se prefiere el uso de filtros específicos frente a técnicas de tipo *Query By Example* (QBE) [Zloof77] debido a que, en general, el usuario no necesita herramientas genéricas de búsqueda. Al contrario, en una unidad de interacción con

[9] Abierta: puede contener variables libres.

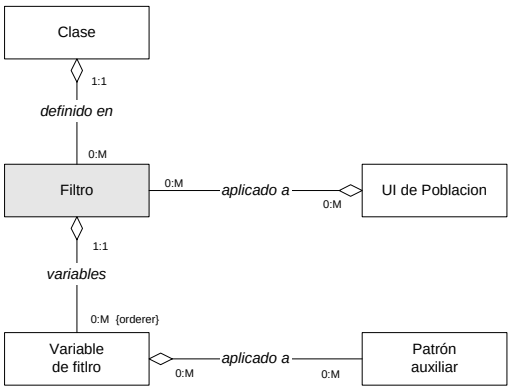


Figura 4.48: Meta-modelo del patrón filtro.

unas tareas bien definidas, el analista determina los criterios más frecuentes de búsqueda para las tareas a realizar y ofertar sólo estos. Este modo de proceder redunda en una mayor usabilidad. Un mecanismo de consulta de tipo QBE requiere de un aprendizaje por parte del usuario y sólo los usuarios avanzados sacarán realmente partido de este tipo de característica.

Todo esto no impide que los mecanismos basados en QBE pueden añadirse como características avanzadas para los usuarios expertos.

Otra ventaja adicional de la identificación temprana de los filtros como parte de los requisitos consiste en que ya desde el diseño de la aplicación pueden sugerirse optimizaciones para mejorar la eficiencia de estos procesos de búsqueda.

Guía de aplicación

Es conveniente definir filtros para clases cuya población de instancias estimada sea mayor de veinte elementos. A partir de estos valores, el usuario no puede observar de un vistazo toda la población de clase. Y es entonces cuando los filtros sirven de ayuda para localizar instancias particulares de un modo rápido.

Ejemplo de problema

Un filtro puede responder a la siguiente necesidad del usuario: *"Se necesita poder localizar los jugadores de un torneo. Como el número de jugadores es elevado, necesitaremos poder buscarlos a partir del nombre y apellido."* El ejemplo expresado en lenguaje formal $\mathcal{OASIS}$ [Pastor95, Letelier98] tendría la fórmula de filtro para la clase Jugador que puede observarse en la Tabla 4.9.

Donde se asume que:

`Nombre like` v_{Nombre} $\wedge$ `Apellido like` $v_{Apellido}$

Tabla 4.9: Ejemplo de fórmula de filtro para localizar jugadores.

- `Nombre` es un atributo de la clase Jugador que almacena el nombre del jugador.
- `Apellido` es un atributo de la clase Jugador que almacena el apellido del jugador.
- v_{Nombre} es una variable de filtro que el usuario dará valor durante la interacción con el filtro en tiempo de ejecución. Aquí el usuario puede indicar todo o parte del nombre para realizar la búsqueda.
- $v_{Apellido}$ es una variable de filtro similar a la anterior para indicar todo o parte del apellido del jugador.

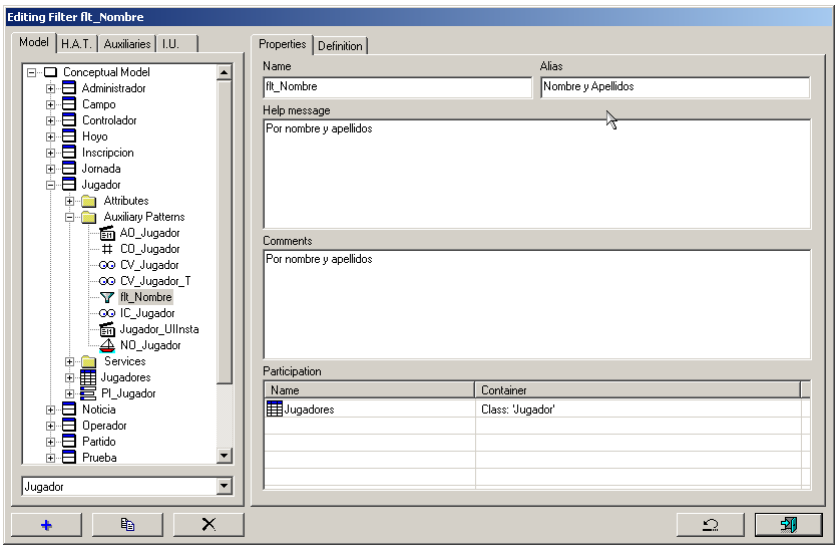


Figura 4.49: Ejemplo de especificación de un filtro 1/2.

Captura en una herramienta CASE de modelado

En la herramienta Oliva Nova Modeler® (*véase Figura 4.49*) podemos definir un nuevo filtro para una clase dada. En este caso definimos el filtro

flt_Nombre sobre la clase Jugador. Como siempre, en esta primera pestaña pueden proporcionarse además un alias, mensaje de ayuda, y comentarios.

La segunda pestaña (*véase Figura 4.50*) contiene la definición propiamente dicha del filtro. En esta pantalla se definen las variables de filtro (parte inferior) y la fórmula del filtro (parte superior).

Una vez completada la definición del filtro, este puede ser aplicado a unidades de interacción de población.

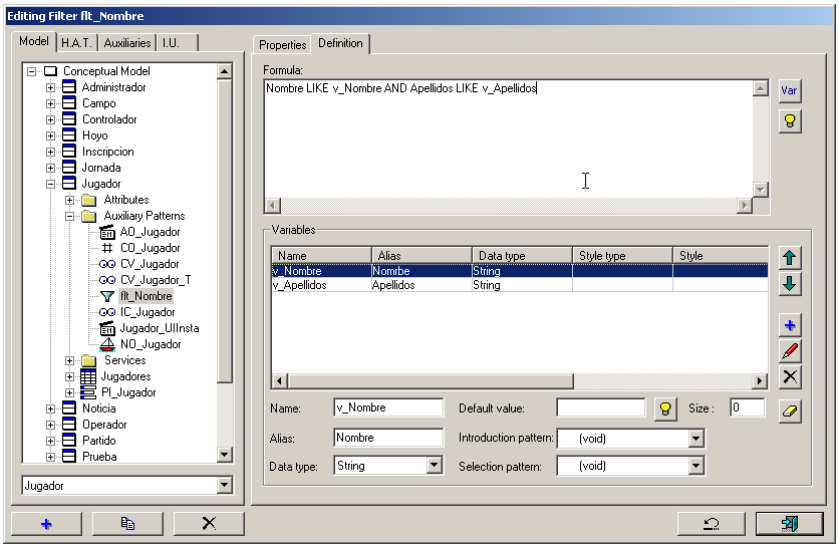


Figura 4.50: Ejemplo de especificación de un filtro 2/2.

Ejemplo de implementación

La Figura 4.51 muestra una unidad de interacción de población que proporciona un filtro (pestaña de parte superior) como facilidad de filtrado. En esta primera pantalla, el filtro no ha sido aplicado todavía y por tanto, la población mostrada es la población completa de la clase.

Por el contrario, en la Figura 4.52 se muestra el mismo escenario donde el usuario ha introducido *"Jav"* en la variable de filtro Nombre y *"Hern"* en la variable de filtro Apellido. Tras pulsar el botón de búsqueda (en el margen derecho de la pestaña conteniendo un icono que representa una lupa), la población mostrada se reduce ostensiblemente para mostrar sólo aquellas instancias que cumplen con la condición impuesta.

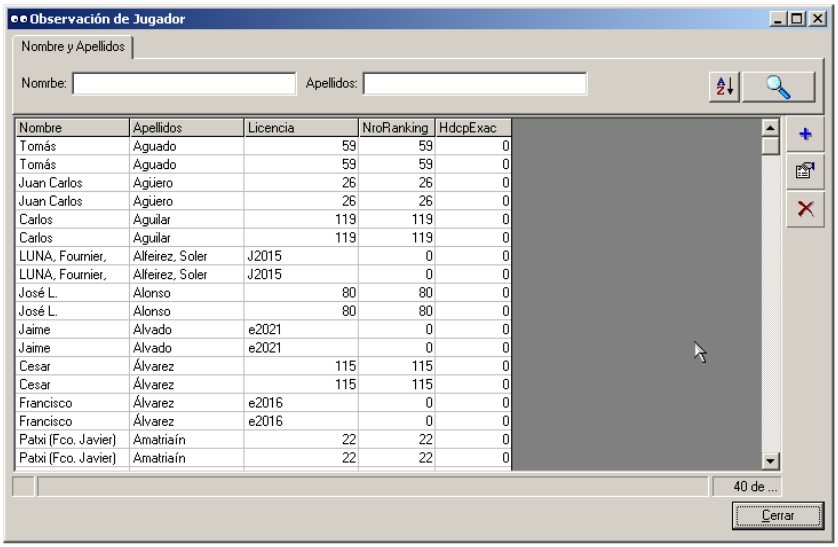


Figura 4.51: Población completa sin aplicar el filtro.

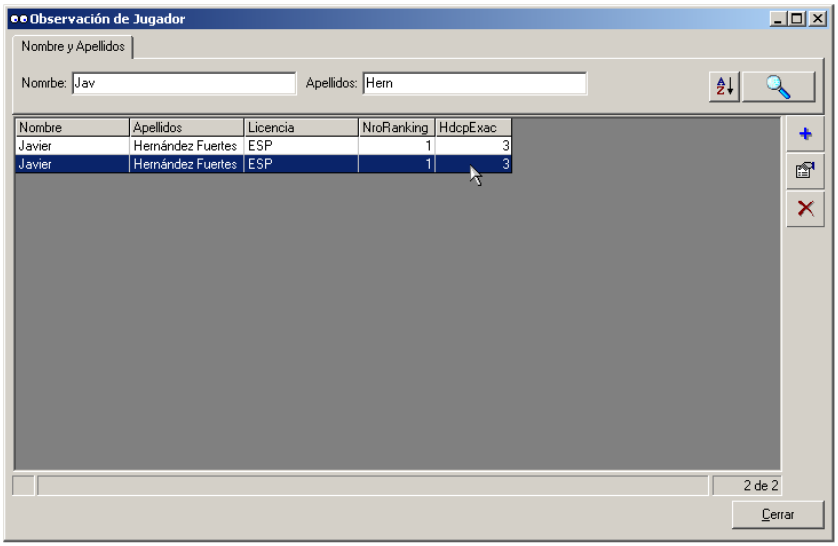


Figura 4.52: Ejemplo de ejecución de un filtro.

3.8 Criterio de ordenación

Nombre

Criterio de ordenación / *Order Criterium*

Descripción

Un criterio de ordenación proporciona un mecanismo de ordenación de objetos basado en las propiedades de éstos.

Definido en

Clase.

Aplicado a

Unidad de Interacción de Población.

Especificación

El criterio de ordenación consiste en una lista ordenada de pares `<expresión, sentido>`. Donde expresión es un atributo visible por la clase (propio o accesible por relaciones de agregación) y sentido indica `ASC` (ascendente) o `DES` (descendente). El orden de los elementos en la lista indica la prioridad de ordenación: el primer criterio es el más prioritario.

Meta-modelo

La Figura 4.53 muestra el meta-modelo para el criterio de ordenación. Un

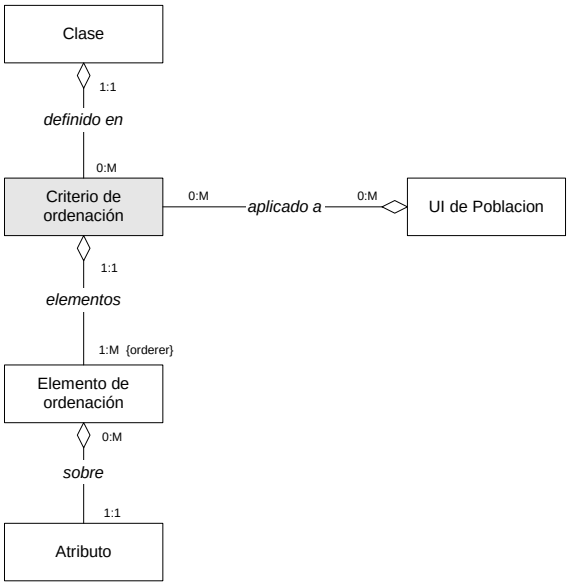


Figura 4.53: Meta-modelo del patrón criterio de ordenación.

Criterio de ordenación se define sobre una Clase. Cada Criterio de ordenación tiene una lista ordenada de Elementos de ordenación donde se indica la expresión de ordenación y su sentido (ascendente o descendente). A su vez, cada Elemento de ordenación tiene un enlace al Atributo alcanzado por la expresión. Los criterios de ordenación definen los mecanismos de ordenación en los los componentes UI de Población.

Semántica asociada

La semántica asociada a un criterio de ordenación C_o es la imposición de una relación de orden[10] $\prec$ sobre un conjunto de objetos $Set(A)$ obteniendo como resultado una lista ordenada o secuencia $seq(A)$.

Fundamento

En los sistemas de información donde se trabaja con grandes colecciones de datos, aparece muy frecuentemente la necesidad de presentar la información ordenada. Una relación de personas clasificadas por nombre y apellidos o una lista de pedidos de fabricación ordenados por fecha de entrega son ejemplos sencillos y cotidianos.

Guía de aplicación

Cuando el conjunto de objetos con el que se trabaja sobrepasa la docena, se hace recomendable emplear un criterio de ordenación para facilitar la búsqueda y exploración de los objetos. No existen criterios a priori para la aplicación de criterios de ordenación. Según el dominio y la naturaleza de las tareas a realizar, diferentes clases de objetos requerirán clasificaciones diferentes dependiendo del contexto de uso.

Ejemplo de problema

Se desea ordenar una serie de Pruebas de un torneo de Golf. El criterio a emplear es la fecha de inicio del torneo en orden ascendente. Lo expresaríamos del siguiente modo:

```
<FechaInicio, ASC>
```

Donde se asume que:

- FechaInicio es un atributo de la clase Prueba.

Captura en una herramienta CASE de modelado

El criterio de ordenación de ejemplo puede ser especificado de un modo muy sencillo. La Figura 4.54 muestra el proceso de creación donde se asigna un nombre al patrón (CO_Prueba, en este caso), un alias y un mensaje de ayuda.

[10] Que la relación de orden sea *parcial* ($\preceq$) o *total* ($\prec$), depende de si el criterio de ordenación considere subconjuntos de atributos únicos (*total*), o no (*parcial*).

En la siguiente pestaña (Figura 4.55) puede observarse como en la parte derecha de la pantalla queda definidos el criterio de ordenación: por fecha de inicio, sentido ascendente.

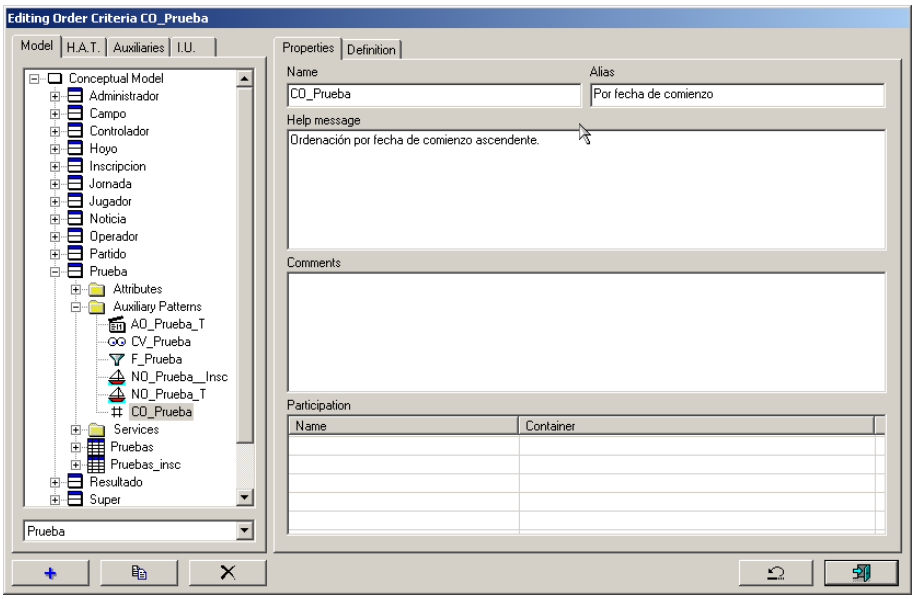


Figura 4.54: Especificación de un patrón de criterio de ordenación 1/2.

Ejemplo de implementación

La Figura 4.56 muestra una unidad de interacción de población para la clase Prueba. En ella podemos apreciar como los objetos mostrados aparecen ordenados por la fecha de inicio en orden ascendente. El símbolo «<» que aparece en la cabecera de la columna refuerza al usuario el criterio de ordenación ques está siendo empleado y su sentido («<» ascendente o «>» descendente).

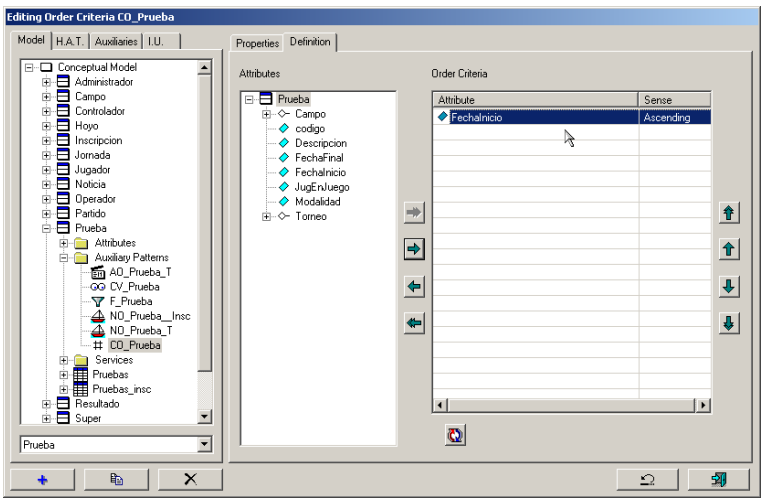


Figura 4.55: Especificación de un patrón de criterio de ordenación 2/2.

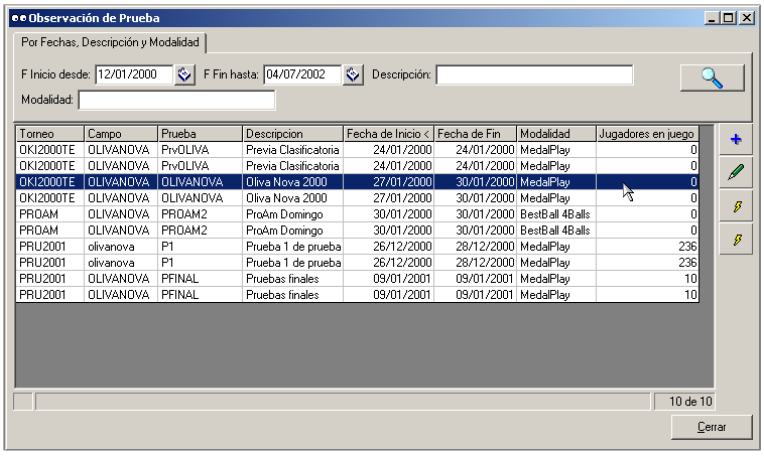


Figura 4.56: Ejemplo de implementación del criterio de ordenación.

3.9 Conjunto de visualización

Nombre

Conjunto de visualización / *Display Set*

Descripción

Un conjunto de visualización describe una lista ordenada de propiedades que son observables por los usuarios en un contexto dado. Es una relación de propiedades de una clase que el usuario necesita observar.

Definido en

Clase.

Aplicado a

Unidad de Interacción de Población, Unidad de Interacción de Instancia, Información complementaria.

Especificación

El conjunto de visualización se especifica como una lista ordenada de expresiones de propiedades (atributos de la propia clase o atributos alcanzables mediante relaciones de agregación alcanzables desde la clase).

Meta-modelo

La Figura 4.57 muestra el meta-modelo para el conjunto de visualiza-

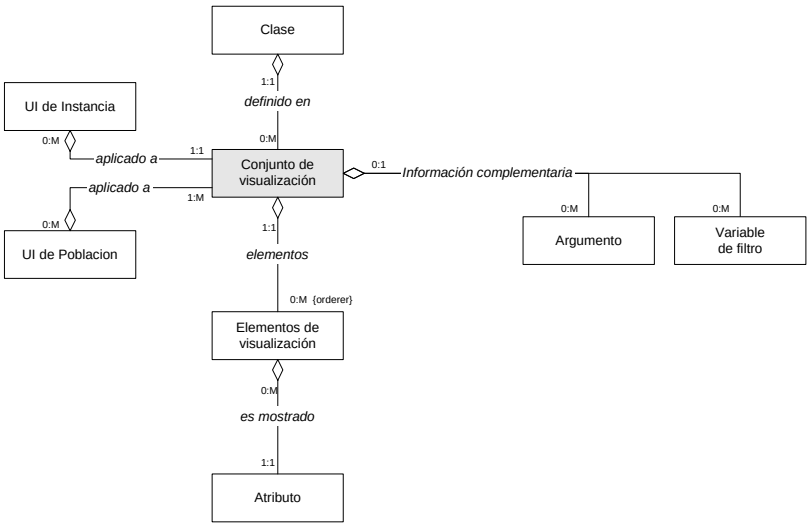


Figura 4.57: Meta-modelo del patrón conjunto de visualización.

ción. El Conjunto de visualización se define sobre una Clase. La definición de los elementos a mostrar es una *lista ordenada*[11] de expresiones sobre atributos almacenados en Elementos de visualización. Una vez creado, el conjunto de visualización puede jugar los roles de:

- *conjuntos de visualización* en unidades de interacción de instancia o de población, o bien
- *información complementaria* en argumentos objeto-valuados y variables de filtro objeto-valuadas.

Semántica asociada

Un conjunto de visualización es una secuencia de propiedades que pueden ser observadas por cualquier usuario (agente). Esta secuencia de propiedades ordenada contiene la información que el analista ha considerado relevante para un objeto y para un escenario determinado. Por sí mismo, el conjunto de visualización restringe el estado observable de los objetos. Sin embargo la relación de agencia (relación de visibilidad establecida por medio de interfaces) entre observador y observado especificada a través de *vistas* permite afinar todavía más la visibilidad de las propiedades en función de los agentes.[12]

Fundamento

La consulta de los objetos, su estado y sus propiedades relevantes son necesarias para que el usuario explore y conozca los objetos que manipula. En muchas ocasiones no es necesario mostrar al usuario todo el estado del objeto (es decir, todas sus propiedades), sino que, al contrario, un subconjunto pequeño puede ser más que suficiente para acceder a la información importante y no perder al usuario en infinidad de datos.

Guía de aplicación

Este patrón es básico para que el usuario pueda consultar objetos. Siempre que se necesite especificar qué información debe ser mostrada en un determinado escenario este patrón puede ser empleado.

Ejemplo de problema

En un sistema de gestión de torneos de golf, el análisis del domino ha identificado que el torneo se compone de pruebas, y a su vez, cada prueba se lleva a cabo en diferentes jornadas. Se desean mostrar una serie de datos correspondientes a una Jornada. La información relevante es la siguiente: El nombre del torneo, la prueba, una descripción de la jornada y la fecha de celebración.

[11] Orden explicitado mediante el estereotipo `Ordered`.

[12] La semántica de visibilidad en función de las vistas será descrita con detalle en la sección 5.4, *página 211*.

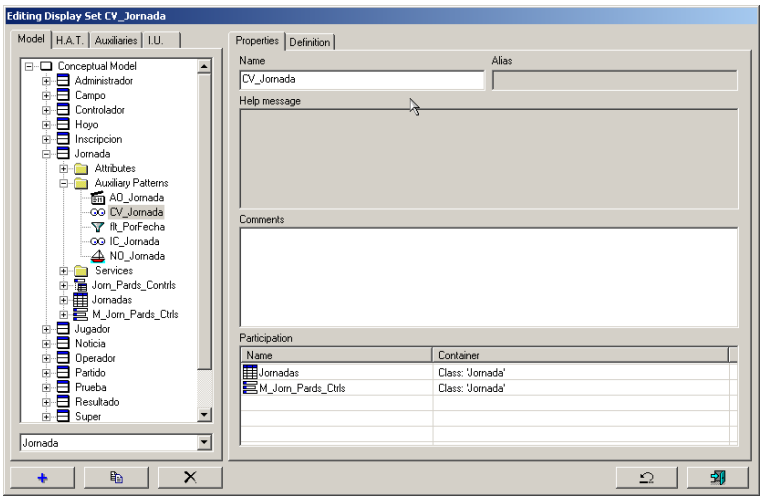


Figura 4.58: Especificación de un conjunto de visualización 1/2.

Captura en una herramienta CASE de modelado

El conjunto de visualización necesario para responder al requisito queda expresado del siguiente modo[13]:

```
Prueba.Torneo.Descripcion, Prueba.Descripcion,
Descripcion, Fecha
```

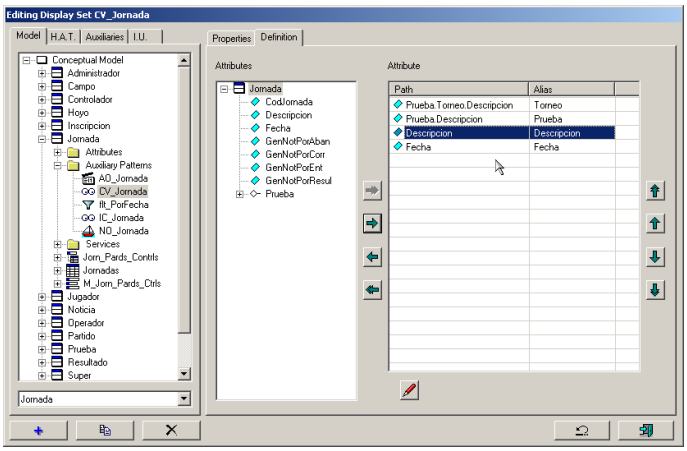


Figura 4.59: Especificación de un conjunto de visualización 2/2.

[13]La notación `a.b` es similar a la empleada en las fórmulas en OCL (*Object Constrain Language*). Donde `a` expresa el cruce de una asociación o agregación usando el rol `a`.

En la herramienta Oliva Nova Modeler® se define el conjunto de visualización por medio de la pantalla ilustrada en la Figura 4.58. Una vez proporcionado un nombre (CV_Jornada) para la nueva instancia del patrón, la segunda pestaña (*véase Figura 4.59*) permite definir la lista ordenada de atributos que deseamos mostrar en la interfaz de usuario final.

Una vez definido, el conjunto de visualización puede ser usado en diferentes contextos: unidades de interacción de instancia o de población y como información complementaria sobre argumentos y variables de filtro objeto valuadas.

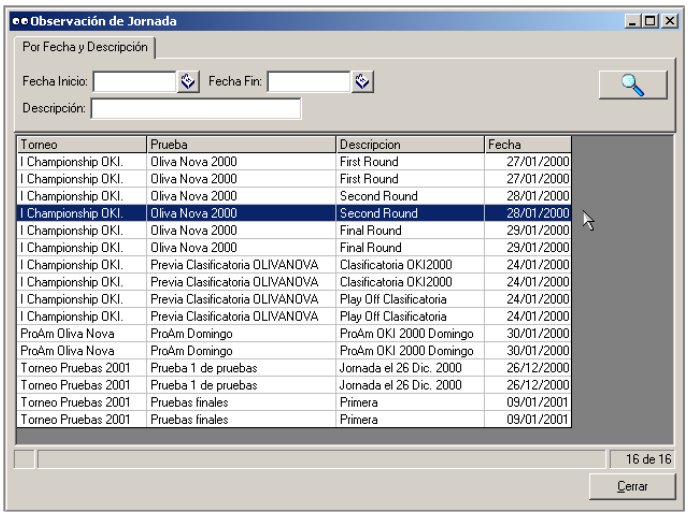


Figura 4.60: Ejemplo de implementación de conjunto de visualización.

Ejemplo de implementación

El ejemplo presentado para el conjunto de visualización muestra una unidad de interacción de población donde se muestra la información especificada para las Jornadas (veasé Figura 4.60). En este caso, el conjunto de visualización ha sido traducido a un componente rejilla donde cada fila representa un objeto y cada columna una propiedad del objeto a ser mostrada.

3.10 Acciones

Nombre

Acciones / *Actions*

Descripción

Determina las acciones que son ofertadas al usuario al seleccionar o fijar un objeto. Las acciones conducen a unidades de interacción que permiten lanzar servicios sobre el objeto seleccionado o editar objetos compuestos por otros más sencillos a través de agregación.

Definido en

Clase.

Aplicado a

Unidad de Interacción de Instancia, Unidad de Interacción de Población.

Especificación

Lista acciones que pueden ser lanzadas y unidades de interacción alcanzados tras la ocurrencia de cada acción. Un conjunto de acciones vendrá constituido por una lista de unidades de interacción ordenada.

Meta-modelo

La Figura 4.61 muestra el meta-modelo para las acciones. Las Acciones se

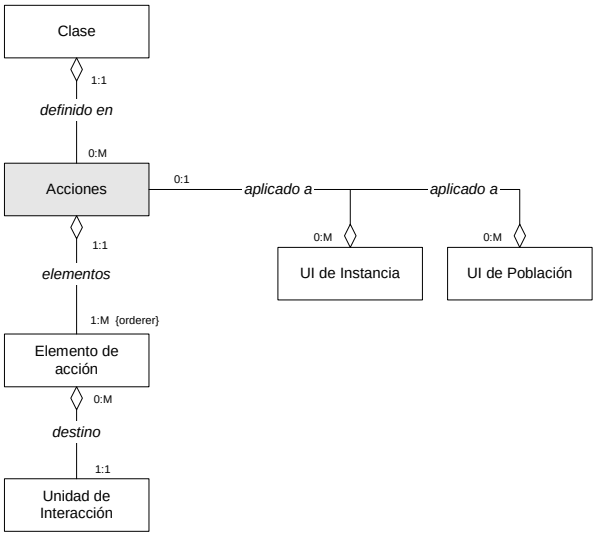


Figura 4.61: Meta-modelo del patrón acciones.

definen sobre una Clase. Las Acciones tienen una lista ordenada de Elementos

de acción donde se indica el alias y la Unidad de Interacción que actúa como destino. Las Acciones definen los mecanismos de exposición de servicios y navegación en los los componentes UI de Instancia y UI de Población.

Semántica asociada

Un conjunto de acciones expresa qué acciones pueden ser llevadas a cabo (por exclusión, ninguna otra acción será permitida) por un usuario en un contexto dado.

Fundamento

El patrón de Acciones describe un subconjunto de servicios o tareas que pueden ser llevadas a cabo sobre el objeto seleccionado. Las acciones constituyen *el verbo* del paradigma de interacción *nombre/verbo*. En decir, responden a la pregunta: *¿Que podemos hacer con el objeto?*

Guía de aplicación

Este patrón es útil para definir el conjunto de acciones aplicables en un contexto donde uno o varios objetos han sido fijados (o seleccionados).

Ejemplo de problema

En un sistema de gestión de torneos de golf, en un escenario dado se requiere dar soporte a las tareas relacionadas con el mantenimiento de pruebas. El usuario debe poder consultar las pruebas existentes y, llegado el caso, necesitará crear, eliminar o modificar las información relativa a las pruebas.

Captura en una herramienta CASE de modelado

Podemos especificar un patrón de acciones que proporcione de un modo muy compacto el conjunto de funcionalidad requerida para el escenario indicado. Comenzaremos definiendo un patrón de acciones (*véase Figura 4.62*) denominado (AO_Partido_T). Las acciones a ofertar son descritas en la segunda pestaña de la ventana (*véase Figura 4.63*). En dicha figura puede apreciarse que se han definido como acciones alcanzables las unidades e interacción de servicio necesarias para proporcionar el acceso a los servicios de creación de pruebas, edición y borrado de las pruebas.

Una vez completada la especificación del patrón de acciones, está listo para poder ser empleado en unidades de interacción de instancia o de población.

Ejemplo de implementación

La Figura 4.64 muestra una implementación de una unidad de población para la clase Prueba. Esta unidad de interacción carece de filtros, pero posee un patrón de acción. Dicho patrón ha sido implementado como una barra de botones dispuesta verticalmente alineada a la derecha de la ventana[14]. Alternativamente, un menú emergente (*popup menu*) presenta también las acciones

[14]Siguiendo el *Principio de Consistencia Visual Espacial*, funcionalidad semejante aparece dispuesta espacialmente en las mismas localizaciones para facilitar su reconocimiento pot el usuario.

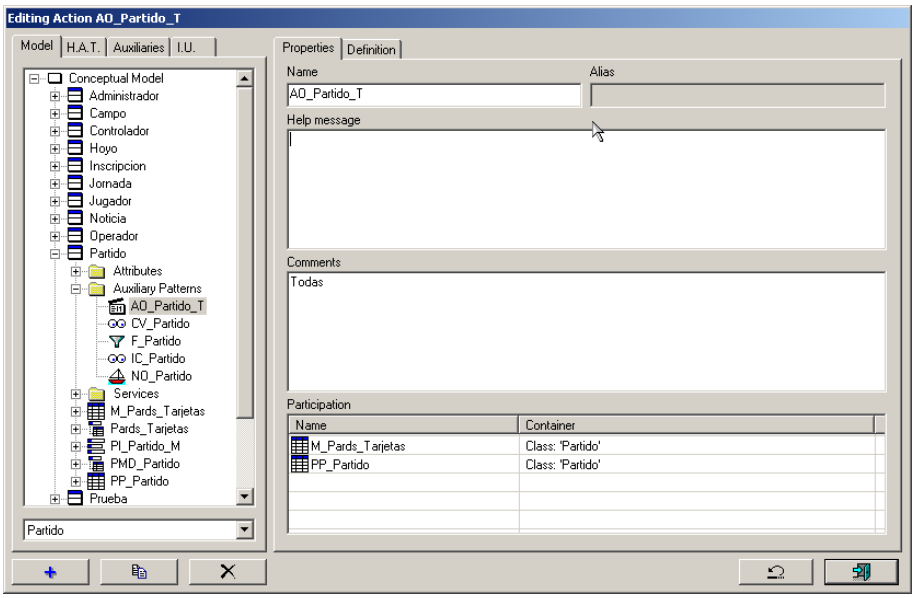


Figura 4.62: Especificación de un patrón de acciones 1/2.

disponibles. Otras opciones auxiliares como Actualizar (actualiza los datos repitiendo la consulta), Limpiar (borra los datos de la pantalla), Opciones (permite acceder a funciones de personalización y exportación) y Ayuda (acceso a la ayuda en linea de la aplicación).

Cuando el usuario selecciona una acción, es conducido a un nuevo escenario donde aparece la unidad de interaction definida como destino del salto de la acción.

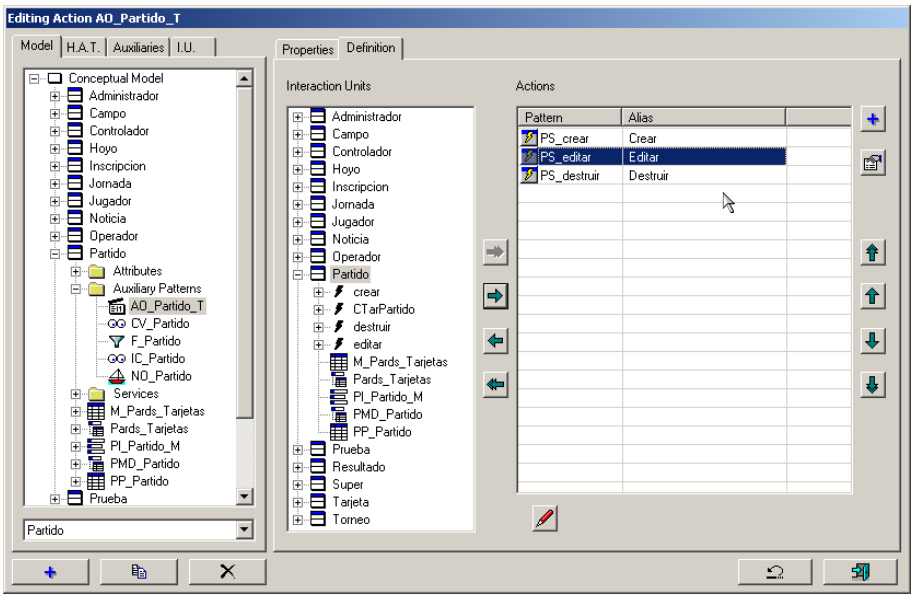


Figura 4.63: Especificación de un patrón de acciones 2/2.

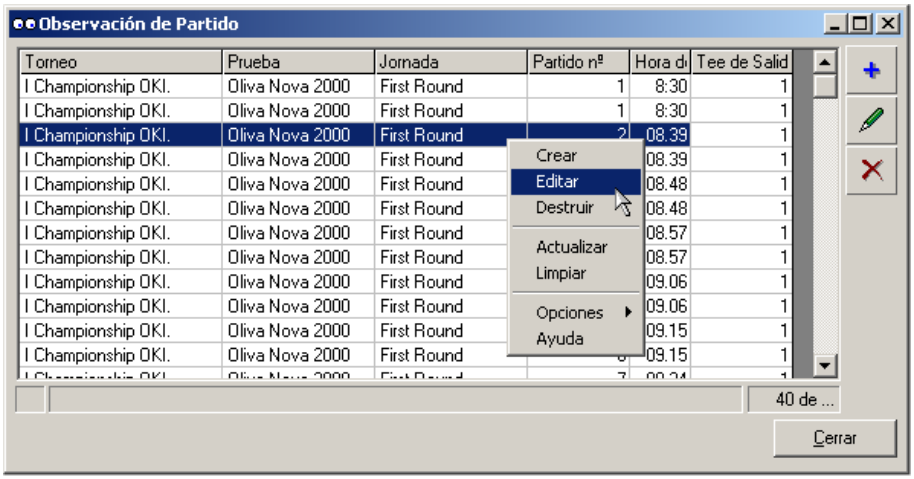


Figura 4.64: Ejemplo de implementación de acciones.

3.11 Navegación

Nombre

Navegación / *Navigation*

Descripción

Determina qué información relacionada con el objeto actual es alcanzable en un contexto dado. Por ejemplo, dada una factura, es interesante que el usuario pueda navegar para obtener más información como es: qué cliente es el destinatario de la factura o que líneas componen la factura.

Definido en

Clase.

Aplicado a

Unidad de Interacción de Instancia, Unidad de Interacción de Población.

Especificación

La navegación es un subconjunto ordenado de las relaciones (agregación y herencia) visibles desde una *clase origen* hasta aquellas *clases destino* alcanzables (son alcanzables precisamente aquellas con las que se tiene relación de visibilidad por agregación o herencia).

Meta-modelo

La Figura 4.65 muestra el meta-modelo para la navegación. La Navegación

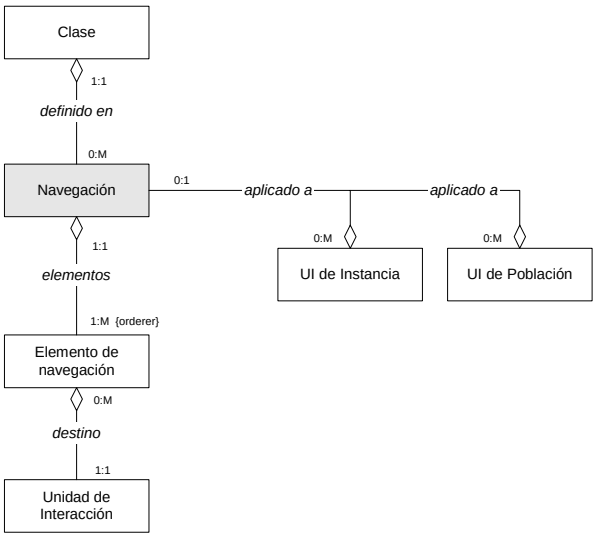


Figura 4.65: Meta-modelo del patrón navegación.

se definen sobre una Clase. La Navegación tienen una lista ordenada de Elementos de navegación donde se indica el alias, una expresión de navegación y la Unidad de Interacción que actúa como destino. La Navegación proporciona mecanismos de consulta de información relacionada en los los componentes UI de Instancia y UI de Población.

Semántica asociada

Dentro del marco de una unidad de interacción, una vez se ha fijado un objeto, se puede ofrecer al usuario la posibilidad de obtener información relacionada con el objeto actual. Un menú de navegación puede ser ofertado en estos casos a partir de la información proporcionada por el patrón de navegación.

Como el patrón está definido como un subconjunto de las relaciones de la clase, es posible restringir la navegación tanto como estime necesario el analista. La navegación, aunque proporciona gran potencia expresiva, ha de quedar limitada a la estrictamente necesaria, de lo contrario se incrementaría la carga mental del usuario innecesariamente.

Fundamento

Cuando los objetos están relacionados con otros, el usuario necesitará consultar la información relacionada. Dependiendo del contexto o escenario de aplicación, diferente información relacionada puede necesitar ser consultada. El analista puede crear diferentes patrones de navegación para proporcionar un comportamiento diferente en cada escenario.

Guía de aplicación

El patrón es aplicable cuando la clase a mostrar contiene información relacionada que puede ser importante para los usuarios.

Ejemplo de problema

En el sistema de gestión de torneos de golf, se necesita mostrar información adicional a las pruebas. Dada una prueba, en ocasiones es necesario recabar información sobre el torneo asociado, las inscripciones, las jornadas así como los campos y hoyos de juego.

Captura en una herramienta CASE de modelado

La captura de este patrón en Oliva Nova Modeler® se lleva a cabo siguiendo el mismo proceso realizado para los patrones precedentes. Una primera pestaña (*véase Figura 4.66*) permite dar nombre al patrón (p.e. NO_Prueba_T) y añadir un mensaje de ayuda y comentarios.

La segunda pestaña (*véase Figura 4.67*) permite construir los elementos de navegación indicando la unidad de interacción destino alcanzada, la expresión de camino cruzada, y el alias a la navegación.

Una vez definido el patrón, este puede ser aplicado a unidades de interacción de instancia o de población.

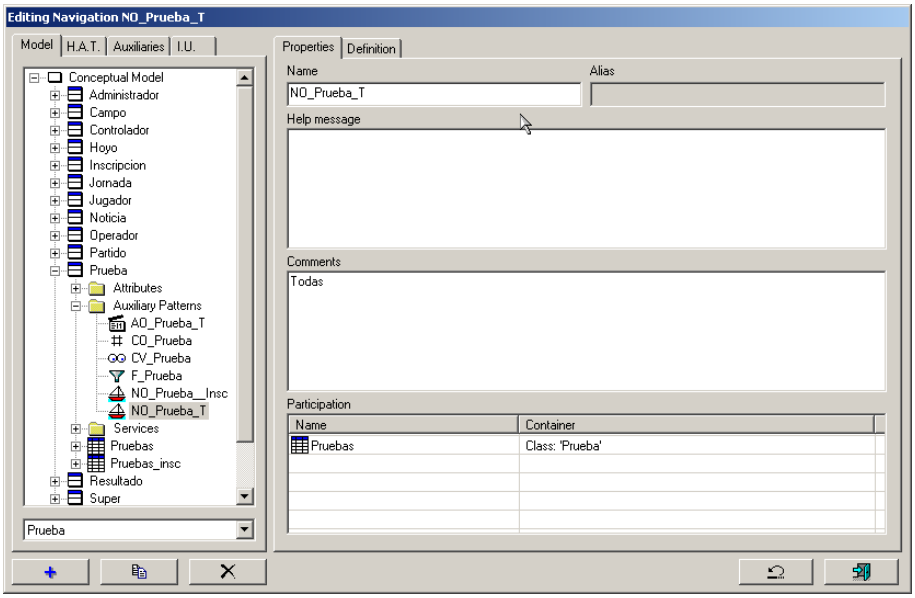


Figura 4.66: Especificación de un patrón de navegación 1/2.

Ejemplo de implementación

De modo análogo al patrón de acciones, la navegación es implementada sobre la plataforma Windows como una banda de botones horizontal (*véase Figura 4.68*). Alternativamente, el menú emergente (*popup menu*) también facilita el acceso a la navegación disponible.

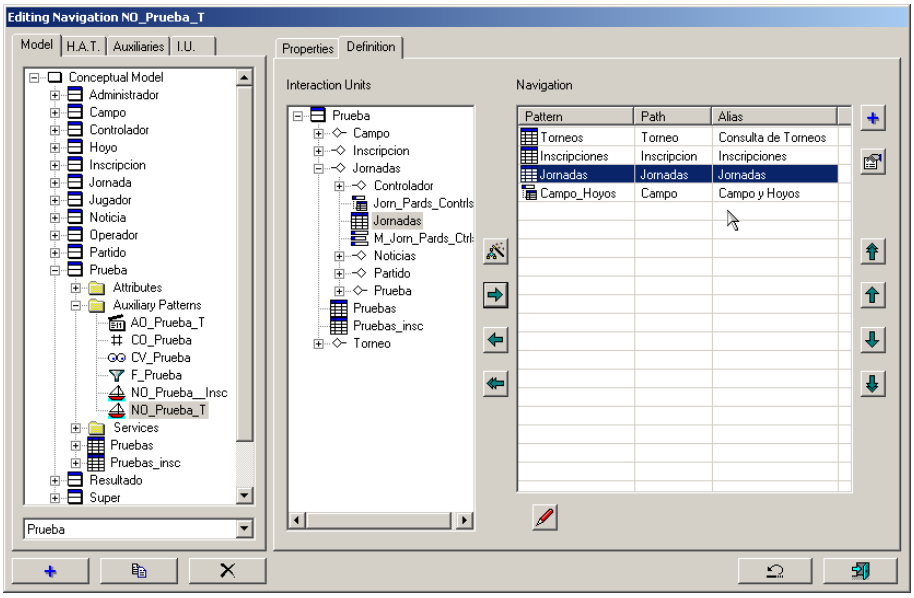


Figura 4.67: Especificación de un patrón de navegación 2/2.

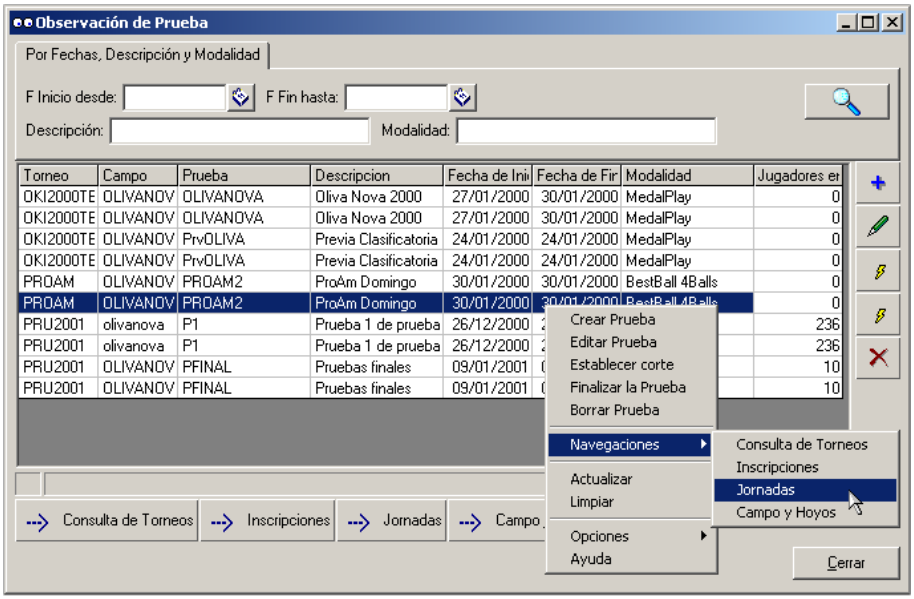


Figura 4.68: Ejemplo de implementación de navegación.

4.3. Notación gráfica del Modelo de Presentación

El lenguaje de patrones presentado está destinado a soportar el proceso de elicitación de requisitos de interfaz de usuario en primer lugar y la especificación de los mismos en una fase posterior. En este sentido se ha pretendido continuar con la filosofía impulsora de OO-Method proporcionando una notación gráfica para el lenguaje de patrones recién propuesto. El objetivo es integrar los métodos de modelado conceptual y recubrirlos de una notación gráfica de diagramado sencilla que oculte la aridez de los lenguajes formales, más fácil de comprender, y sobre todo, realmente escalable para permitir abordar con éxito proyectos industriales.

Constantine y Lookwood [Constantine99] propusieron una técnica para propotipar requisitos de interfaz de usuario usando papel y Post-Its.[15] En dicha técnica, cada Post-It representa un escenario (p.e. un formulario o una página Web en la implementación). El usuario y el analista pueden crear cuantos escenarios necesiten y después, enlazan los escenarios dibujando flechas entre los diferentes Post-Its. Se pretende que la notación gráfica posea un grado de sen-

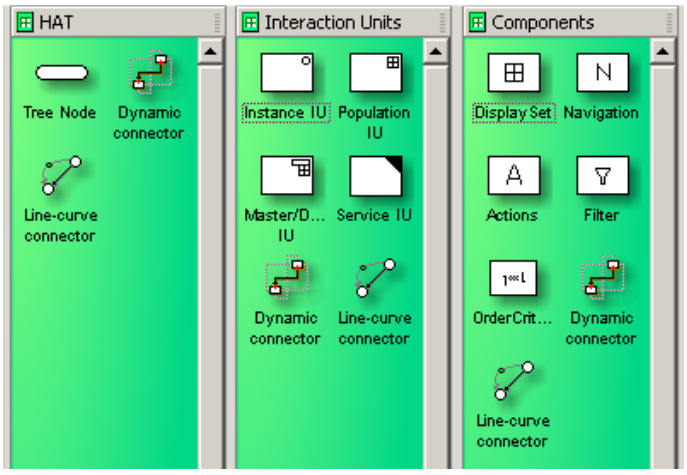


Figura 4.69: Paleta de componentes en VISIO.

cillez similar, de modo que pueda ser usada de un modo semejante: el objetivo perseguido es que los analistas puedan discutir sobre el papel con el cliente para construir y validar las especificaciones de requisitos relativas a la interfaz de usuario.

Por estos motivos, la notación propuesta sigue un método intuitivo para describir interfaces de usuario inspirada en el método propues-

[15] Post-It™ es una marca registrada de 3M Inc.

to por Ruble[Ruble97] y posteriormente por Constantine y Lookwood [Constantine99]. De hecho, los conceptos del Modelo de Presentación pueden ser recogidos y negociados con el cliente del mismo modo: usando tan sólo lápiz y papel.

Al igual que hemos estructurado los patrones en tres niveles (1. acceso, 2. unidades de interacción y 3. patrones elementales), la notación gráfica está compuesta por tres tipos de diagramas que permiten trabajar a tres niveles de abstracción diferentes poniendo el foco de atención y el alcance en los problemas que realmente pretende resolver cada nivel.

Usando la herramienta de diagramado VISIO[VIS02] se han diseñado unas paletas de componentes para prototipar el diseño del lenguaje gráfico (*véase Figura 4.69*). Las paletas están agrupadas por niveles, disponiendo por tanto de diagramas diferentes para cada nivel. Las siguientes secciones describen con mayor detalle cada uno de los niveles.

Notación de nivel 1. Acceso

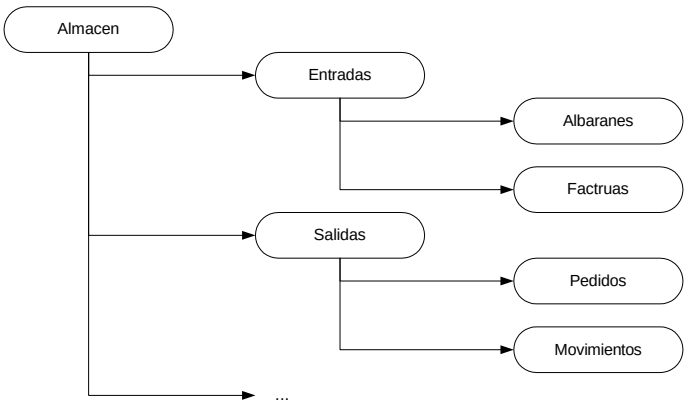


Figura 4.70: Notación gráfica para nivel 1 (AJA).

La notación para el nivel 1. (Acceso) consiste en una notación gráfica sencilla para representar el Árbol de Jerarquía de Acciones. Para representar la estructura de árbol la notación proporciona dos constructores:

- Los correspondientes nodos se representan por medio de rectángulos con aristas redondeadas etiquetados con el alias del nodo.
- La relación de un nodo padre a sus subnodos hijos se denota usando flechas dirigidas en sentido padre → hijo.

Figura 4.71: Notación gráfica propuesta para las Unidades de Interacción.

La Figura 4.70 muestra un ejemplo de diagrama para un Árbol de Jerarquía de Acciones sencillo.

Notación de nivel 2. Unidades de interacción

El nivel 2 está compuesto por las Unidades de Interacción. El diagrama propuesto para este nivel es un grafo dirigido donde:

- los nodos son Unidades de Interacción (representados como cajas □) y
- los arcos representan enlaces navegacionales entre pares de Unidades de Interacción (representados como flechas →).

La Figura 4.71 muestra las primitivas definidas para este tipo de diagramas representando los diferentes conceptos.

Las Unidades de Interacción son representadas como cajas decoradas con un pequeño icono en la esquina superior derecha. Los iconos empleados son:

- Un triángulo negro para las Unidades de Interacción de Servicio.
- Un circulo para las Unidades de Interacción de Instancia.
- Una rejilla para las Unidades de Interacción de Población.
- Un rectángulo junto a una rejilla para las Unidades de Interacción de Maestro/Detalle.

Notación de nivel 3. Patrones elementales.

Por último, el nivel 3 proporciona una representación gráfica para los patrones elementales que forman parte de las unidades de interacción de instancia y población. La Figura 4.72 muestra los distintos símbolos empleados para cada primitiva: conjuntos de visualización, filtros, criterios de ordenación, acciones y navegación.

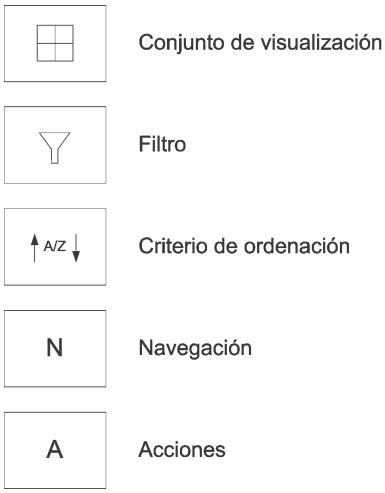


Figura 4.72: Notación gráfica para el nivel 3.

Relación de descomposición por niveles

Cada uno de los niveles puede verse como la descomposición del precedente donde se proporciona un mayor nivel de detalle. En este sentido, la Figura 4.73 muestra un ejemplo donde la unidad de interacción de maestro/detalle UIMD_FacturaLineas. Dicha unidad puede ser descompuesta o despiezada en sus componentes: una unidad de interacción de instancia (UII_Factura) y una unidad de interacción de población (UIP_Lineas) relacionados por medio de una relación de navegación etiquetada con el nombre del rol atravesado. A su vez, estas Unidades de Interacción pueden ser despiezadas en sus componentes básicos: conjuntos de visualización, criterios de ordenación, filtros, acciones y navegación.

Esta estrategia de descomposición por niveles permite implementar de modo natural un editor de modelos gráficos basados en el «*Principio de Aproximación Gradual*» (*véase página 77*), donde cada diagrama de refinamiento proporciona mayores detalles que el precedente.

4.4. Conclusiones del capítulo

En el presente capítulo se ha presentado el lenguaje de patrones conceptuales para la especificación de interfaces de usuario Just-UI. La descripción de cada uno de los patrones basados en una plantilla común permite comparar y contrastar los patrones entre sí y respecto a otras colecciones de patrones existentes como [Tidwell99, Welie00a]. Dichos patrones son empleados como:

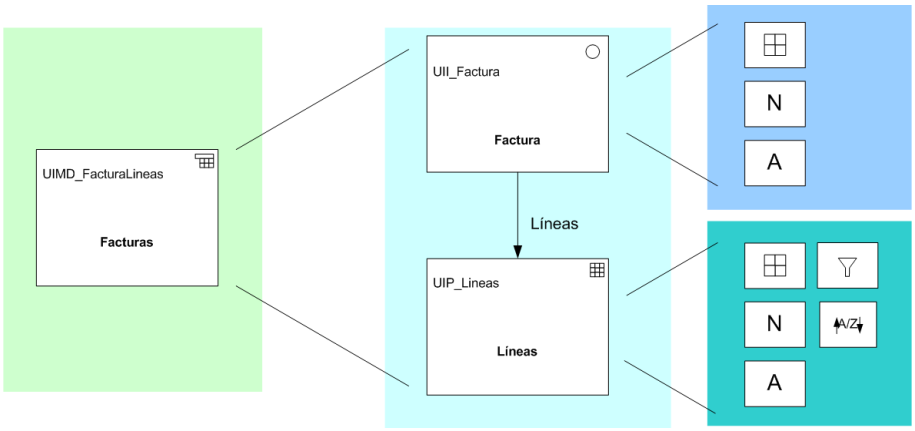


Figura 4.73: Despiece del modelo en sucesivos diagramas.

- conceptos del espacio del problema útiles para la *elicitación de requisitos* de interfaz de usuario,
- *modelo de especificación* conceptual de interfaces de usuario,
- *lenguaje común* entre los analistas, diseñadores y programadores,
- a partir del cuales es posible *generar código completo* (capaz de ser ejecutado).

Por último, se ha propuesto una notación gráfica que permite disponer de un lenguaje de especificación. Las representaciones gráficas son más naturales que lo que pueden ser una representación textual al tiempo que explota las relaciones de continencia y composicionalidad tan frecuentes en las interfaces de usuario.

La notación es muy cómoda para los analistas a la hora de maquetar prototipos y permite ser escalada para proyectos grandes soportada por una herramienta.

Capítulo 5

Formalización del modelo

«Lo bueno, si breve, dos veces bueno.»

— Baltasar Gracian, escritor español, siglo XVII.

«Everything should be as simple as possible, but no simpler.»

— Albert Einstein, Premio Nobel de Física 1921, 1879 – 1955.

Los patrones surgen de la observación, experimentación y finalmente abstracción. Debido a esta naturaleza de carácter fuertemente empírica, es difícil la formalización de los patrones. Sin embargo, es necesario hacer el esfuerzo para describir de modo no ambiguo la semántica capturada por los patrones.

5.1. Meta-modelo

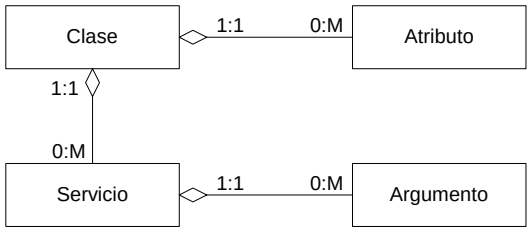


Figura 5.1: Núcleo básico de meta-modelo para un modelo orientado a objetos.

El meta-modelo para cada patrón ha sido ya introducido en el capitulo 4 junto a la definición de cada patrón. La presente sección describe cómo los patrones son combinados entre sí en términos de meta-modelado.

El meta-modelo que da cuenta del Modelo de Presentación es una extensión a un meta-modelo orientado a objetos clásico como puede ser el meta-modelo de OO-Method [Pelechano98] o el meta-modelo de UML [OMG97a, OMG97b]. Como ilustra la Figura 5.1, los únicos prerrequisitos necesarios en el núcleo del modelo orientado a objetos a extender para implementar la extensión correspondiente al Modelo de Presentación son los conceptos de: Clase, Atributo, Servicio (o método) y Argumento, y sus relaciones habituales.

El meta-modelo básico presentado es muy poco restrictivo. Todas las metodologías orientadas a objetos definen como mínimo un núcleo equivalente al presentado en la Figura 5.1.

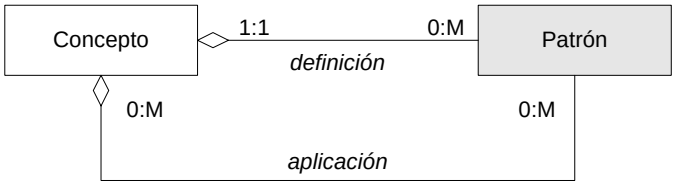


Figura 5.2: Mecanismo básico de relaciones entre patrones.

El mecanismo de extensión está basado en introducir nuevos elementos: los patrones y sus clases y relaciones de soporte. El enlace entre los patrones y los conceptos tradicionales es alcanzado usando dos tipos de relaciones (*véase Figura 5.2*):

- *Definición.* Un patrón siempre está definido sobre un único concepto. Sin embargo, un concepto puede definir patrones al mismo tiempo (define un espacio de nombres «*namespace*» donde los conceptos son definidos). Este tipo de relación es establecida durante la creación (instanciación) del patrón. E.g. filtros, conjuntos de visualización y navegación se definen en el ámbito de una Clase. La unidad de interacción de servicio está definida para un servicio dado.

- *Aplicación.* Una vez creado, un patrón puede ser aplicado o participar en otros patrones. En este sentido, los patrones pueden ser reusados en el modelo (aplicados en diferentes escenarios). Las relaciones de aplicación se establecen por el analista durante el proceso de construcción del modelo. E.g. filtros, conjunto de visualización y navegación son aplicados (o usados) en la unidad de interacción de Población.

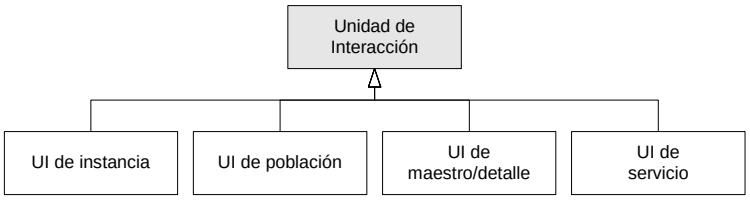


Figura 5.3: Subtipos de unidades de interacción.

El concepto de unidad de interacción es subdividido en cuatro subtipos (*véase Figura 5.3*). Cada subtipo redefine y extiende propiedades y comportamiento.

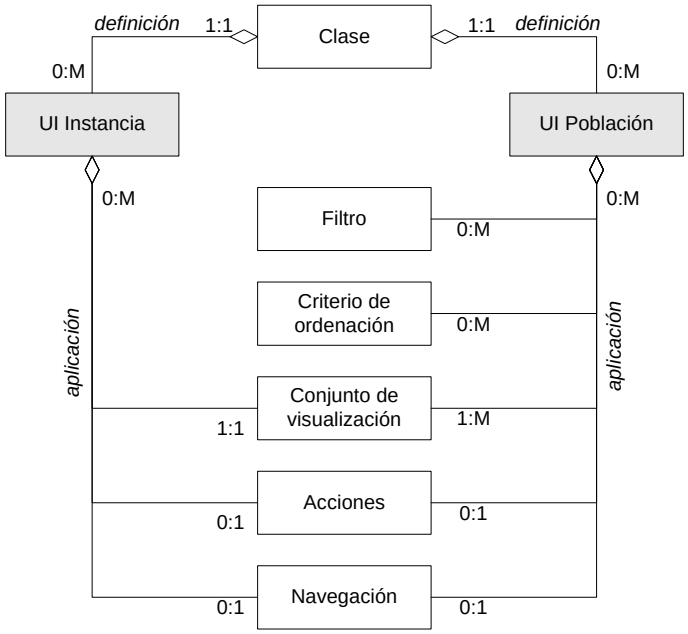


Figura 5.4: Meta-modelo para las unidades de interacción de instancia y población.

El meta-modelo asociado a la unidades de interacción de instancia y de población puede observarse en la Figura 5.4. En dicha figura, las UI de instancia y población están *definidas* en el ámbito de una clase. Conjuntos de visualización, acciones y navegación son *aplicados* a la UI de instancia y población. Filtros y criterios de ordenación son *aplicados* exclusivamente a UI de población.

El meta-modelo para la unidad de interacción de maestro/detalle se muestra en la Figura 5.5. La UI de maestro/detalle se *define* sobre una clase dada.

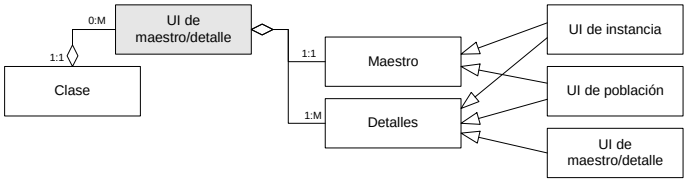


Figura 5.5: Meta-modelo para la unidad de interacción de maestro/detalle.

Las unidades de interacción de instancia y población pueden jugar el papel de componentes maestro y detalle (*patrones aplicados*). Sin embargo, la UI de maestro/detalle (de modo recursivo) puede jugar, a su vez, el papel de detalle (*patrón aplicado*).

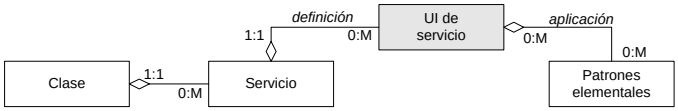


Figura 5.6: Meta-modelo para la unidad de interacción de maestro/detalle.

La Figura 5.6 muestra el meta-modelo para la unidad de interacción de servicio. La UI de servicio se *define* en el ámbito de un servicio. Los patrones elementales (como Introducción, Dependencia, etc.) pueden ser también *aplicados* a la UI de servicio.

De este modo, obtenemos un modelo unificado orientado a objetos comprendiendo en la especificación la estructura, funcionalidad e interfaz de usuario. El uso de las relaciones de definición y aplicación permiten un mayor nivel de reuso de los patrones especificados en el modelo. El reuso a nivel conceptual facilita mantener la homogeneidad en la especificación y, en consecuencia, en el sistema finalmente implementado.

5.2. Semántica del diagrama navegacional

El diagrama navegacional que podemos establecer a partir de las unidades de interacción y relaciones de navegación y acciones es isomorfo a un grafo dirigido donde:

1. cada unidad de interacción puede ser representada como una nodo «□» y donde
2. cada elemento de navegación o acción puede ser representado como arco dirigido «⟶».

Cada elemento de navegación Nav_i o acción Acc_j determinan arcos entre una unidad de interacción origen y una unidad de interacción destino:

$$UI_{origen} \longrightarrow_{Nav_i} UI_{destino} \tag{5.1}$$

o bien:

$$UI'_{origen} \longrightarrow_{Acc_j} UI'_{destino} \tag{5.2}$$

Definición 5.2.1 ***Diagrama navegacional***
El diagrama navegacional $\mathcal{DN}$ es un grafo dirigido compuesto por un conjunto de unidades de interacción (nodos) y un conjunto de patrones de acción y navegación (arcos dirigidos).

$$\mathcal{DN} = \langle \{UI_1, UI_2 \dots UI_n\} \ , \ \{Acc_1, Acc_2 \dots Acc_n, Nav_1, Nav_2 \dots Nav_m\} \rangle \tag{5.3}$$

Dada la isomorfía del diagrama navegacional respecto a un grafo dirigido y aplicando la teoría establecida de gráfos, podemos comprobar:

1. si el grafo es conexo,
2. si existen nodos sumidero (sin arcos de salida),
3. si existen nodos fuente (sin arcos de entrada),
4. si existen ciclos en el grafo, etc.

Estas comprobaciones realizadas de modo automático pueden servir de alerta para el analista a la hora de localizar errores de diseño en un diagrama navegacional.

5.3. Semántica basada en tareas

Las técnicas de Análisis de Tareas son útiles para describir las tareas implicadas en la interfaz de usuario. En este campo, Fabio Paternò [Paternò00] propuso una notación gráfica (notación *ConcurTaskTree*) para construir especificaciones de tareas. Esta notación es un excelente azúcar sintáctico que oculta la complejidad del lenguaje formal LOTOS[1] [ISO88].

En esta sección se propone el uso de la notación *ConcurTaskTree* para describir la semántica del lenguaje de patrones. Esta descripción será de utilidad para entender la semántica de los patrones. Mas aún, cada implementación de

[1] LOTOS es el lenguaje que soporta los operadores temporales en los que se soporta la notación *ConcurTaskTree* .

una especificación abstracta deberá ser compatible con el modelo y respetar su semántica.

La notación de ConcurTaskTree ha sido introducida en el capítulo 2 en la página 40.

5.3.1. Especificación de los patrones usando notación CTT

La principal idea detrás del lenguaje de patrones es usar pequeñas piezas para construir la especificación y reusar tales piezas para construir posteriormente componentes más complejos.

Siguiendo esta misma idea, se propone usar subárboles CTT para describir la semántica de tareas de cada patrón elemental del lenguaje para, a continuación, construir un único árbol de tareas CTT que describa la semántica de tareas de la especificación completa.

Para alcanzar este objetivo comenzaremos definiendo el concepto de *Punto de conexión*.

Definición 5.3.1 ***Punto de conexión***
Un nodo de un subárbol CTT es un Punto de conexión si y sólo si:

1. *El nodo es una hoja y es una tarea de interacción etiquetada con la sintaxis de esterotipo «`xxxx Link`».*
2. *Ningún otro nodo puede ser considerado un Punto de conexión.*

Unidad de interacción de servicio

La unidad de interacción de servicio permite al usuario proporcionar los argumentos necesarios para lanzar un servicio en un escenario dado. La tareas de interacción que pueden ser llevadas a cabo por el usuario son:

- introducir el valor de un parámetro,
- lanzar el servicio solicitado y
- cancelar la invocación del servicio.

La Figura 5.7 muestra el árbol CTT para la unidad de interacción de servicio. El usuario tiene que introducir los valores para los parámetros del servicio. El usuario puede seleccionar cualquier parámetro sin restricciones y a continuación, cambiar su valor. En cualquier momento el usuario puede lanzar dos acciones excluyentes entre si: `Close` (Cerrar. Cancela la ejecución del servicio

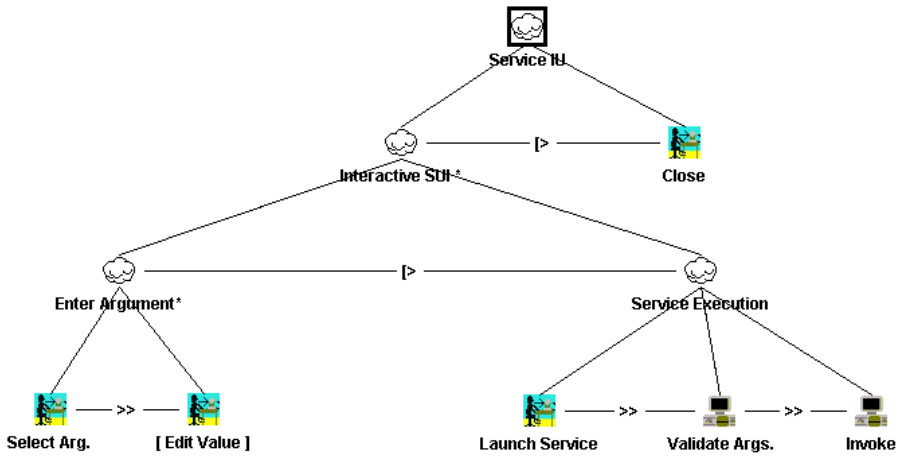


Figura 5.7: Árbol CTT para la unidad de interacción de servicio.

cerrando la unidad de interacción.) o `Launch Service` (Lanzar el servicio. Comprueba los valores introducidos para los parámetros y lanza la ejecución del servicio.) En este último caso, el sistema realiza una tarea de validación y finalmente, invoca la funcionalidad requerida.

No existe punto de conexión en este subárbol.

Unidad de interacción de instancia

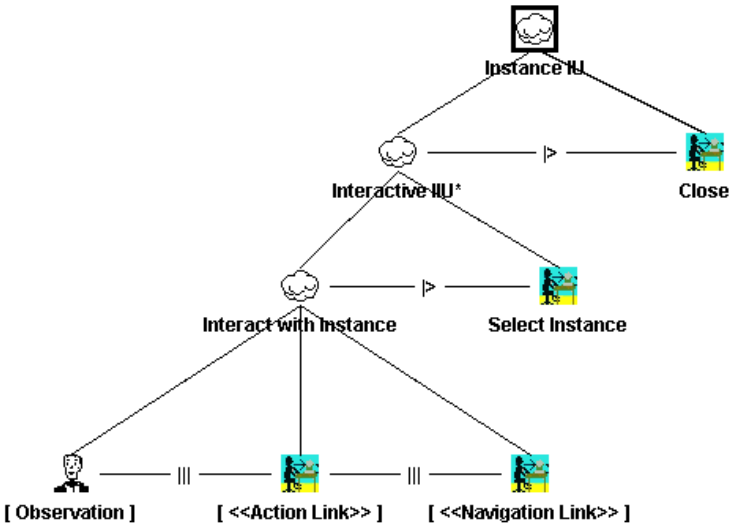


Figura 5.8: Árbol CTT para la unidad de interacción de instancia.

El segundo tipo de unidad de interacción está fuertemente orientada a la tareas de observación de objetos. Las tareas de usuario involucradas son:

- seleccionar un objeto con el cual trabajar,
- observación,
- solicitar un cambio en el estado del objeto,
- demandar información relacionada y
- abandonar la unidad de interacción.

El árbol CTT correspondiente (*véase Figura 5.8*) muestra como las tareas están temporalmente relacionadas. El usuario puede observar los datos del objeto, solicitar un cambio, demandar información adicional o cerrar la unidad de interacción (sin restricciones). En cualquier momento, el objeto de trabajo puede ser seleccionado de nuevo.

Los nodos hoja «`Action Link`» y «`Navigation Link`» son Puntos de conexión para alcanzar el patrón de *acción* y *navegación*, respectivamente.

Unidad de interacción de población

La unidad de interacción de población es útil para representar colecciones de objetos. Las tareas de usuario involucradas son:

- seleccionar un criterio de ordenación,
- seleccionar un criterio de filtrado,
- observación,
- seleccionar un objeto de trabajo,
- solicitar un cambio al estado de un objeto,
- demandar información relacionada y
- abandonar la unidad de interacción.

El árbol CTT (*véase Figura 5.9*) muestra las tareas relacionadas en el escenario y sus relaciones temporales. Un usuario puede buscar objetos, seleccionar una instancia y finalmente interaccionar con él. La tarea de búsqueda contiene subtareas como seleccionar un criterio de ordenación, un filtro, rellenar los valores para las variables del filtro y finalmente ordenar la búsqueda. Una vez los resultados aparecen, el usuario puede seleccionar una instancia e interaccionar con ella.

De modo similar a la UI de instancia, los nodos hoja marcados como «`Action Link`» y «`Navigation Link`» son Puntos de Conexión para alcanzar el patrón de *acción* y *navegación*, respectivamente.

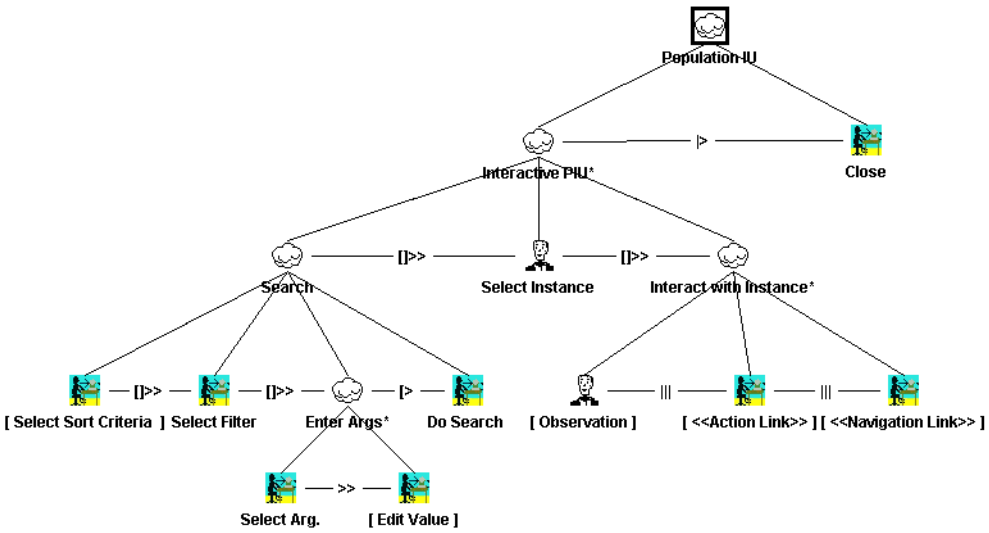


Figura 5.9: Árbol CTT para la unidad de interacción de población.

Unidad de interacción maestro/detalle

La unidad de interacción maestro/detalle está compuesta por diversas unidades de interacción. Las unidades de detalle están fuertemente sincronizadas con la unidad maestro.

El árbol CTT (*véase Figura 5.10*) es simple pero expresa claramente la relación de sincronización entre maestro y detalles. Las tareas detalles están sincronizadas con la tarea maestro usando el operador temporal [ ] >> indicando *habilitación con intercambio de información*. En cualquier momento, el usuario puede interrumpir el diálogo cerrando la unidad de interacción.

Los puntos de conexión en este árbol CTT son los siguientes:

- El nodo hoja «`Master UI Link`». Conecta con una *unidad de interacción* que juega el rol de maestro.
- El nodo hoja «`Detail UI Link`». Conecta con una *unidad de interacción* que juega el rol de detalle.

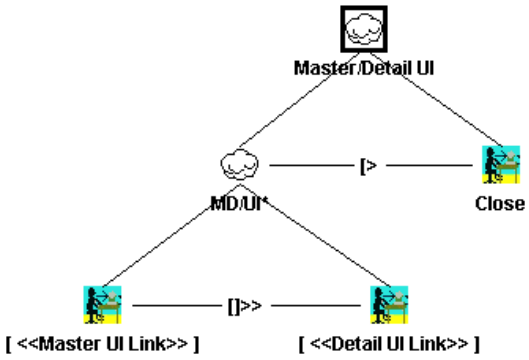


Figura 5.10: Árbol CTT de una unidad de interacción Maestro/Detalle.

Acciones

Las acciones son alternativas puras ([]): el usuario puede seleccionar una acción que le conducirá directamente a otra unidad de interacción. El árbol CTT es muy sencillo (*véase Figura 5.11*). Contiene un nodo hoja para cada elemento de acción. Se usa el operador alternativa ([]) para indicar que el usuario puede elegir una u otra sin restricciones.

Los Puntos de Conexión son cada uno de los elementos de acción (saltos navegables a otras unidades de interacción).

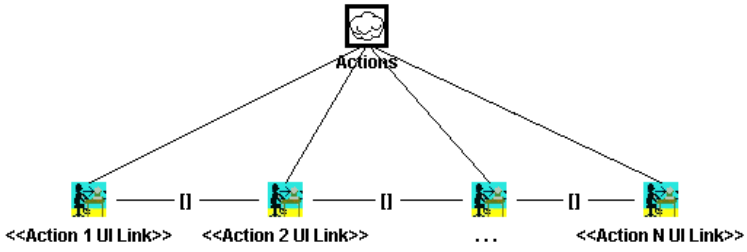


Figura 5.11: Árbol CTT para acciones.

Navegación

La navegación presenta al usuario un conjunto de enlaces para alcanzar información relacionada al objeto actual. De nuevo, la navegación constituye una alternativa. El usuario puede seleccionar un elemento de navegación que le conducirá directamente a otra unidad de interacción para mostrar información relacionada. El árbol CTT es similar al anterior (*véase Figura 5.12*). Contiene

un nodo hoja para cada elemento de navegación usando el operador de lógica temporal alternativa [].

Los puntos de conexión aquí son similares a los de Acciones: cada uno de los nodos hoja que representan a los elementos de navegación seleccionables (saltos navegables a unidades de interacción destino).

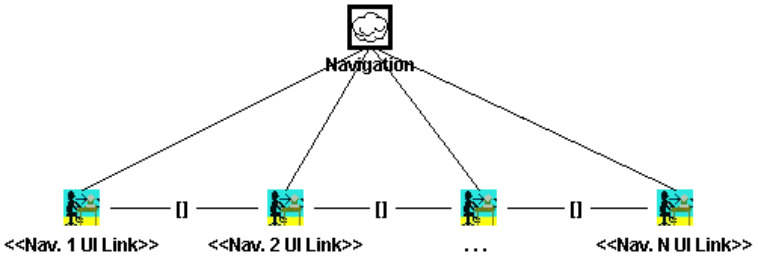


Figura 5.12: Árbol CTT para navegación.

Árbol de Jerarquía de Acciones

El AJA está compuesto por una estructura arborescente para disponer la funcionalidad de la aplicación al usuario. En sus nodos hoja se da acceso a unidades de interacción específicas.

El árbol CTT asociado puede ser expresado como una alternativa en el árbol. El usuario selecciona una nodo hoja usando un control de tipo menú arborescentes. Aunque se podría modelar los nodos intermedios del AJA en el árbol CTT, se han evitado por simplicidad (solamente los nodos hoja han sido considerados). El árbol resultante es de nuevo un árbol de alternativa pura donde los nodos hoja del árbol CTT representan a los nodos hoja del árbol AJA (*véase Figura 5.13*).

El nodo raíz es la raíz del árbol CTT de la especificación. Cada elemento hoja del AJA es un Punto de Conexión con otras unidades de interacción.

5.3.2. Especificación como composición de subárboles

La semántica de tareas del lenguaje de patrones puede ser definida como la composición de la semántica para cada concepto descrito en los apartados precedentes. Las reglas de composición son las siguientes:

Definición 5.3.2 *El árbol de tareas completo es un árbol CTT donde:*

1. *La raíz es el árbol CTT correspondiente al AJA de la vista considerada.*

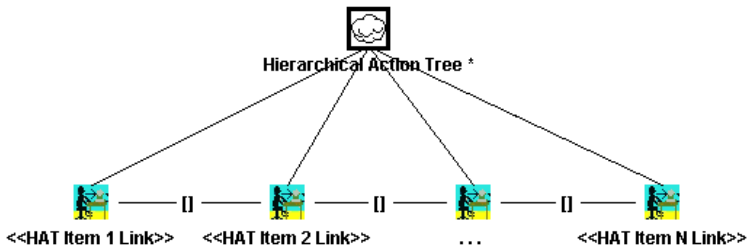


Figura 5.13: Árbol CTT para el árbol de jerarquía de acciones.

2. *Para cada elemento del* AJA *punto de conexión del árbol anterior, sustituirlo usando el correspondiente subárbol CTT para la unidad de interacción destino.*
3. *Para cada punto de conexión* Action *del árbol, substituirlo por el correspondiente subárbol CTT para acciones.*
4. *Para cada punto de conexión* Navigation *del árbol, substituirlo por el correspondiente subárbol CTT para navegación.*
5. *Para cada elemento* Nav_i *y* Action_j *Puntos de conexión del árbol, sustituir por el correspondiente subárbol para la unidad de interacción destino.*
6. *Para cada punto de conexión* Master *y* Detail*, substituir por el correspondiente subárbol CTT para la unidad de interacción destino.*
7. *Aplicar recursivamente los pasos numerados 3, 4, 5 y 6 hasta que no queden más puntos de conexión como nodos hoja.*

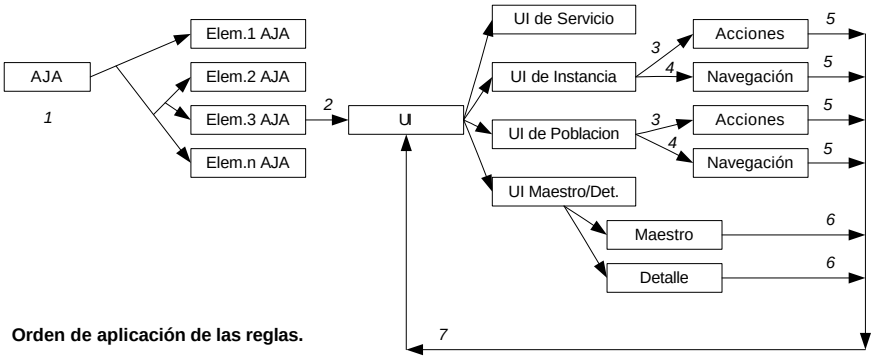


Figura 5.14: Orden de aplicación de las reglas de composición del árbol de tareas.

La Figura 5.14 muestra una representación gráfica del algoritmo de composición del árbol de tareas. El árbol CTT construido de este modo expresa la

semántica de tareas de la interfaz de usuario especificada según el lenguaje de patrones descrito en el capítulo 4. Este árbol CTT puede ser traducido directamente a una especificación en LOTOS equivalente. Por tanto, puede ser validado, animado o verificado usando las herramientas disponibles en el entorno CTT y las desarrolladas para el lenguaje formal LOTOS.

Problema de terminación

El formalismo CTT tiene una estructura de árbol. Sin embargo, las especificaciones basadas en el lenguaje de patrones están basadas en grafos dirigidos (donde las unidades de interacción son nodos y donde las navegaciones y acciones son arcos). Obviamente, las reglas algebraicas dictan que un grafo con ciclos no puede ser representado usando árboles finitos.

Si la especificación tiene ciclos, el árbol obtenido de la definición 5.3.2 no será finito.

Sin embargo, no es necesario construir el árbol por completo. Los ciclos pueden ser detectados y la expansión en tal punto podada. Siempre que un usuario interactúa con una interfaz de usuario realiza un camino finito en el árbol de tareas. Por tanto, en las herramientas de animación y simulación la expansión puede realizarse de modo incremental cuando sea necesario.

5.3.3. Conclusiones

La representación de la semántica de los patrones mediante la notación CTT puede ser útil para:

- validar las especificaciones usando herramientas basadas en CTT [Paternò02] y
- localizar patrones Just-UI en especificaciones CTT.

5.4. Semántica de visibilidad por agente

El Modelo de Presentación en asociación con el modelo conceptual de OO-Method permite establecer una semántica de visibilidad muy precisa para el desarrollo de interfaces adaptadas al usuario.

Para detallar dicha semántica, se introducirán en primer lugar los conceptos de *interfaz* y *vista*. A continuación se expondrán de modo incremental los diferentes conceptos de visibilidad para concluir resumiendo como la visibilidad del sistema se adapta en función del usuario conectado en un momento dado a la interfaz de usuario.

5.4.1. Interfaces

Una interfaz en *OASIS* es una declaración de permisos de visibilidad entre dos clases. En ella, una clase juega el rol de *clase agente* (observadora o cliente) y una segunda clase juega un segundo rol denominado *clase servidora* (u observada). Mediante la interfaz, la clase servidora publica las propiedades que son visibles de ella para la clase agente.

De este modo, el concepto de *actor* de UML, adquiere categoría de *clase agente* en *OASIS* . El propio modelo conceptual permite recoger las propiedades de los agentes y especificar con gran nivel de detalle los permisos de visibilidad y ejecución para cada agente. Desde el punto de vista de las especificaciones clásicas de interfaz de usuario podemos reseñar que esta expresividad semántica constituye un Modelado de Usuario en la línea de lo realizado en MOBI-D [Puerta96, Puerta97a].

Existen diversos tipos de interfaces en *OASIS* v. 2.0 [Pastor95].

- **Interfaz de proyección**, restringe la signatura de una clase observable por otras clases agentes.
- **Interfaz de restricción de población**, restringe la población de una clase observada por otras clases agentes.
- **Interfaz a medida**, permite redefinir restricciones de integridad dinámicas, precondiciones de eventos, relaciones de disparo y atributos para adaptar la interfaz de la clase al uso de un agente específico.
- **Interfaz derivada**, permite añadir nuevos atributos derivados a la signatura de la clase observada.

En *OASIS* 3.0 se describen cuatro tipos de interfaces dependiendo de si el alcance del rol agente y servidor recae sobre una clase o un objeto (*véase Tabla 5.1*).

Tipo de Interfaz	**Agente**	**Servidor(a)**
Interfaz clase-clase	clase	clase
Interfaz objeto-clase	objeto	clase
Interfaz clase-objeto	clase	objeto
Interfaz objeto-objeto	objeto	objeto

Tabla 5.1: Tipos de interfaces en *OASIS* 3.0.

En particular, y para el estudio que nos ocupa, nos centraremos en la *Interfaz de proyección* y en las interfaces de tipo *clase-clase*. La siguiente definición de *Interfaz de proyección* está tomada de *OASIS* 3.0 [Letelier98].

Definición 5.4.1 ***Interfaz de proyección***
Siendo $Cliente$ una especificación de cliente y $Servidor$ una especificación de servidor, si $\sum_{servidor}$ es la signatura de atributos y servicios de la clase del servidor determinado por $Servidor$, entonces una interfaz de proyección de $Cliente$ respecto de $Servidor$ está definida por una función f de la forma:

$$f : (Cliente, Servidor) \rightarrow \sum_{Servidor}^{'} \tag{5.4}$$

La función f define a qué a atributos y servicios tiene acceso el cliente restringiendo su percepción de la signatura del servidor, es decir, se cumple que $\sum'_{Servidor} \subseteq \sum_{Servidor}$. Así, una interfaz de proyección, además de establecer un canal de comunicación, puede restringir los atributos y servicios accesibles por el cliente en la clase servidora.

La interfaz de proyección definida en $\mathcal{OASIS}$ no tiene en cuenta el problema de los roles (extensión introducida en OO-Method para el tratamiento de las relaciones de agregación y su visibilidad). Por estos motivos, en [Sánchez99] se define la interfaz OO-Method como una interfaz de proyección $\mathcal{OASIS}$ extendida para dar cuenta de la visibilidad de los roles.

Definición 5.4.2 ***Interfaz***
Se define la interfaz como la función I entre las clases agente (cliente) A y la clase servidora S como:

$$I(A, S) \rightarrow \sum_{S}^{*} \tag{5.5}$$

La signatura extendida para roles de la clase S es: $\sum_S = \langle A, R, Se \rangle$ donde,

- $A = \{a_1, a_2, \ldots, a_n\}$ es el conjunto de atributos definidos en la clase S.
- $R = \{rol_1, rol_2, \ldots, rol_m\}$ el conjunto de roles accesibles (por relaciones de agregación) desde la clase S.
- $Se = \{Serv_1, Serv_2, \ldots, Serv_p\}$ el conjunto de servicios (eventos, transacciones y operaciones) definidos en la clase S.

Por otro lado, la signatura proyectada por la interfaz, $\sum_S^* = \langle \alpha, \rho, \sigma \rangle$ cumple que: $\alpha \subseteq A$, $\rho \subseteq R$ y $\sigma \subseteq Se$. Por tanto $\sum_S^* \subseteq \sum_S$.

La existencia de una interfaz expresa que la clase agente tiene permisos de:

1. visibilidad sobre los atributos declarados,

2. visibilidad sobre las relaciones de agregación que etiquetan los roles expresados y
3. ejecución sobre los servicios declarados en la interfaz.

Por tanto, un objeto perteneciente a una clase agente A sólo puede tener visibilidad sobre la clase servidora S si existe una interfaz $I(A, S)$ entre la clase agente A y la clase servidora S. La visibilidad está limitada al subconjunto de atributos y roles declarados en la interfaz. Los permisos de ejecución están limitados al subconjunto de servicios declarados en la interfaz.

5.4.2. Vistas

Definición: «Una vista en OO-Method es una lista de interfaces» [Sánchez99].

La vista puede ser considerada como un subconjunto del sistema desde el punto de vista de la interfaz de usuario. Podemos pensar en grandes sistemas (e.g. un sistema de gestión integrado de una empresa) donde puede ser necesario crear diferentes interfaces de usuario:

- para cada departamento (*compras, ventas, almacén, atención al cliente*),
- para cada tipo de usuario (*administradores, operadores, responsables*),
- o bien, para cada tipo de canal (*interfaz de usuario destinadas a la Web, a una intranet o a un dispositivo móvil*).

Estos ejemplos muestran claramente cómo en función del contexto, del tipo de usuario, rol jugado y medio de acceso, la interfaz de usuario necesita ser adaptada.

De este modo, se puede considerar la vista como la unidad de generación para la interfaz de usuario. Esta es la razón por la cual el patrón Árbol de jerarquía de acciones esta definido sobre una vista y no sobre el modelo completo.

Por defecto, si en un modelo no se ha definido una vista, puede asumirse la existencia de una vista implícita conteniendo a todas las interfaces del modelo.

A continuación se propondrán la siguientes definiciones asociadas a las vistas:

Definición 5.4.3 ***Vista Global***
Una vista global V_{Global} es el conjunto de todas las interfaces definidas en un esquema conceptual ($Spec$).

$$V_{Global} = \{I_i \quad / \quad I_i \in Spec\} \tag{5.6}$$

Definición 5.4.4 ***Vista***
Una vista V (o vista parcial, como contrapartida a vista global) es cualquier subconjunto arbitrario no vacío de la vista global.

$$V = \{I_1, I_2, \ldots, I_n\} \quad / \quad V \subseteq V_{Global} \ \wedge \ V \neq \phi \tag{5.7}$$

5.4.3. Definiciones de visibilidad

Los conceptos que son introducidos a continuación determinan los criterios de visibilidad de una agente en una vista dada.

Definición 5.4.5 ***Clases agentes de una vista***
Las clases agentes de una vista V, son el subconjunto de clases que participan como agentes en las interfaces de la vista.

$$C_{Agentes}(V) = \{C_i \quad / \quad \exists \ I \in V \ / \ I(C_i, X)\} \tag{5.8}$$

Definición 5.4.6 ***Clases observables de una vista***
Análogamente, las clases servidoras de una vista V, son el subconjunto de clases que participan como servidoras en las interfaces de la vista.

$$C_{Observables}(V) = \{C_i \quad / \quad \exists \ I \in V \ / \ I(X, C_i)\} \tag{5.9}$$

Definición 5.4.7 ***Objeto agente***
Un objeto $\underline{O_{ag}}$ puede actuar como agente dentro de una vista V y por lo tanto conectarse al sistema para interaccionar con la sociedad de objetos si y sólo si es instancia de una clase agente de la vista V.

$$ObjetoAgente(\underline{O_{ag}}, V) \Longleftrightarrow \underline{O_{ag}} \text{ es instancia de } C \ \wedge \ C \in C_{Agentes}(V) \tag{5.10}$$

Definición 5.4.8 ***Objeto observable***
Análogamente, un objeto $\underline{O_{obs}}$ es observable en la vista V si y sólo si es instancia de una clase observable de la vista V.

$$ObjetoObservable(\underline{O_{obs}}, V) \Longleftrightarrow \underline{O_{obs}} \text{ es instancia de } C \ \wedge \ C \in C_{Observables}(V) \tag{5.11}$$

Las siguientes definiciones que se presentan tienen como ámbito de aplicación:

1. Una Vista V dada y
2. un objeto agente conectado $\underline{O_a}$ instancia de la clase A.

Definición 5.4.9 ***Clase visible***
Una clase C_{Obs} se dice que es visible para los objetos agentes instancias de la clase A en la vista V si y sólo si existe una interfaz entre A y C_{Obs} incluida en la vista V.

$$ClaseVisible(C_{Obs}, A, V) \Longleftrightarrow \exists\ I \in V\ /\ I(A, C_{Obs}) \tag{5.12}$$

Definición 5.4.10 ***Atributo visible***
Un atributo Atr definido en la clase C_{Obs} es visible para los objetos agentes instancias de la clase A en la vista V si sólo si existe una interfaz entre A y C_{Obs} y el atributo está incluido en la interfaz.

$$AtributoVisible(Atr, A, V) \Longleftrightarrow \exists\ I \in V\ /\ I(A, C_{Obs}) = \langle \alpha, \rho, \sigma \rangle\ \ \wedge\ \ Atr \in \alpha \tag{5.13}$$

Definición 5.4.11 ***Rol visible***
Un rol (Rol) alcanzable desde la clase C_{Obs} es visible para los objetos agentes instancias de la clase A en la vista V si sólo si existe una interfaz entre A y C_{Obs} y el rol está incluido en la interfaz.

$$RolVisible(Rol, A, V) \Longleftrightarrow \exists\ I \in V\ /\ I(A, C_{Obs}) = \langle \alpha, \rho, \sigma \rangle\ \ \wedge\ \ Rol \in \rho \tag{5.14}$$

Definición 5.4.12 ***Servicio visible***
Un servicio $Serv$ definido en la clase C_{Obs} es visible para los objetos agentes instancias de la clase A en la vista V si sólo si existe una interfaz entre A y C_{Obs} y el servicio está incluido en la interfaz.

$$ServicioVisible(Serv, A, V) \Longleftrightarrow \exists\ I \in V\ /\ I(A, C_{Obs}) = \langle \alpha, \rho, \sigma \rangle\ \wedge\ Serv \in \sigma \tag{5.15}$$

Definición 5.4.13 ***Expresión alcanzable***
Una expresión exp de la forma «$rol_1.rol_2.\ldots.rol_n[.atr]$» es alcanzable para los objetos agentes instancias de la clase A en la vista V si sólo si existe visibilidad sobre cada uno de los roles rol_i y sobre el atributo atr.

$$\begin{aligned} ExpAlcanzable(exp, A, V) \iff & \forall\ rol_i,\ RolVisible(rol_i, A, V)\ \wedge \\ & AtributoVisible(atr, A, V) \end{aligned} \tag{5.16}$$

Definición 5.4.14 ***Unidad de interacción visible***
Para determinar si una unidad de interacción UI_i es visible para los objetos agentes instancias de A en la vista V debemos atender al tipo de la unidad de interacción.

$$\begin{aligned} UIVisible(UI_i, A, V) \iff & UISVisible(UI_i, A, V)\ \vee \\ & UIIVisible(UI_i, A, V)\ \vee \\ & UIPVisible(UI_i, A, V)\ \vee \\ & UIMDVisible(UI_i, A, V) \end{aligned} \tag{5.17}$$

Definición 5.4.15 ***Unidad de interacción de servicio visible***
Una unidad de interacción de servicio UIS definida para el servicio $Serv$ es visible para los objetos agentes instancias de A en la vista V si y sólo si el servicio $Serv$ es visible.

$$UISVisible(UIS, A, V) \iff UIS \text{ definida en } Serv \ \wedge \ ServicioVisible(Serv, A, V) \quad (5.18)$$

Definición 5.4.16 ***Unidad de interacción de instancia visible***
Una unidad de interacción de instancia UII definida en la clase S es visible para los objetos agentes instancias de A en la vista V si y sólo si la clase S es visible.

$$UIIVisible(UII, A, V) \iff UII \text{ definida en } S \ \wedge \ ClaseVisible(S, A, V) \quad (5.19)$$

Definición 5.4.17 ***Unidad de interacción de población visible***
Una unidad de interacción de población UIP definida en la clase S es visible para los objetos agentes instancias de A en la vista V si y sólo si la clase S es visible.

$$UIPVisible(UIP, A, V) \iff UIP \text{ definida en } S \ \wedge \ ClaseVisible(S, A, V) \quad (5.20)$$

Definición 5.4.18 ***Unidad de interacción de maestro/detalle visible***
Una unidad de interacción de maestro/detalle $UIMD$ definida en la clase S con $UI_{Maestro}$ como UI maestro y (exp_i, $UI_{Detalle_i}$) como par: expresión y UI detalle i-ésimo, es visible para los objetos agentes instancias de A en la vista V si y sólo si:

1. *$UI_{Maestro}$ es visible,*
2. *cada $UI_{Detalle_i}$ es visible y*
3. *cada exp_i es alcanzable.*

$$\begin{aligned} UIMDVisible(UIMD, A, V) \iff\ & UIMD = \langle UI_{Maestro}, \{(exp_i, UI_{Detalle_i})\}\rangle \\ & \text{definida en } S \ \wedge \\ & UIVisible(UI_{Maestro}, A, V) \ \wedge \\ & (\forall\, i,\ UIVisible(UI_{Detalle_i}, A, V) \ \wedge \\ & ExpAlcanzable(exp_i, A, V)) \end{aligned} \quad (5.21)$$

Definición 5.4.19 ***Elemento de conjunto de visualización visible***
Un elemento de un conjunto de visualización $ItemCV = \langle Alias, expr\rangle$ es visible si

y sólo si la expresión $expr$ es alcanzable para los objetos agentes instancias de A en la vista V.

$$ItemCVVisible(ItemCV, A, V) \iff ItemCV = \langle Alias, expr\rangle \ \wedge \ ExpAlcanzable(expr, A, V) \quad (5.22)$$

Definición 5.4.20 ***Elemento de acción alcanzable***
Un elemento de acción $ItemAcc = \langle Alias, UI_{dest}\rangle$ es alcanzable si y sólo si la unidad de interacción destino UI_{dest} es visible para los objetos agentes instancias de A en la vista V.

$$ItemAccAlcanzable(ItemAcc, A, V) \iff ItemAcc = \langle Alias, UI_{dest}\rangle \ \wedge \ UIVisible(UI_{dest}, A, V) \quad (5.23)$$

Definición 5.4.21 ***Elemento de navegación alcanzable***
Un elemento de navegación $ItemNav = \langle Alias, expr, UI_{dest}\rangle$ es alcanzable si y sólo si la expresión $expr$ es alcanzable y la unidad de interacción destino UI_{dest} es visible para los objetos agentes instancias de A en la vista V.

$$ItemNavAlcanzable(ItemNav, A, V) \iff ItemNav = \langle Alias, expr, UI_{dest}\rangle \ \wedge \ ExpAlcanzable(expr, A, V) \ \wedge \ UIVisible(UI_{dest}, A, V) \quad (5.24)$$

Definición 5.4.22 ***Conjunto de visualización efectivo***
El conjunto de visualización efectivo CV_{ef} respecto al conjunto de visualización CV es la sublista (preservando el orden) de elementos de CV que son visibles para los objetos agentes instancias de A en la vista V.

$$CV_{ef}(CV, A, V) = \langle ItemCV_i \ / \ ItemCV_i \in CV, \ \wedge \ ItemCVVisible(ItemCV_i, A, V)\rangle \quad (5.25)$$

De la definición 5.4.22 se deduce trivialmente que $CV_{ef} \subseteq CV$.

Definición 5.4.23 ***Acciones efectivas***
Las acciones efectivas Acc_{ef} respecto al patron de acciones Acc es la sublista (preservando el orden) de elementos de Acc que son alcanzables para los objetos agentes instancias de A en la vista V.

$$Acc_{ef}(CV, A, V) = \langle ItemAcc_i \ / \ ItemAcc_i \in Acc, \ \wedge \ ItemAccAlcanzable(ItemAcc, A, V)\rangle \quad (5.26)$$

De la definición 5.4.23 se deduce trivialmente que $Acc_{ef} \subseteq Acc$.

Definición 5.4.24 ***Navegación efectiva***
La navegación efectiva Nav_{ef} respecto al patron de navegación Nav es la sublista (preservando el orden) de elementos de Nav que son alcanzables para los objetos agentes instancias de A en la vista V.

$$Nav_{ef}(CV, A, V) = \langle ItemNav_i \quad / \quad ItemNav_i \in Nav, \quad \wedge \quad ItemNavAlcanzable(ItemNav, A, V)\rangle \tag{5.27}$$

De la definición 5.4.24 se deduce trivialmente que $Nav_{ef} \subseteq Nav$.

Definición 5.4.25 ***Árbol de jerarquía de acciones***
Un Árbol de jerarquía de acciones AJA se define sobre una vista V. El árbol puede ser expresado como:

1. *un conjunto de nodos, donde cada nodo está compuesto por un alias más un conjunto de nodos (de modo recursivo) $Nodo_i = \langle Alias, \langle Nodo_j \ldots\rangle\rangle$ o bien,*
2. *contiene un alias y una unidad de interacción destino $Nodo_i = \langle Alias, UI_{dest}\rangle$*

$$AJA = \langle Nodo_i, Nodo_{i+1}, \ldots\rangle \quad / \quad \forall\, Nodo_i, \; Nodo_i = \langle Alias, \langle Nodo_j, Nodo_{j+1}, \ldots\rangle \quad \vee \; Nodo_i = \langle Alias, UI_{dest}\rangle \tag{5.28}$$

Ejemplo: Un AJA para un vista podría haber sido definido como sigue:

- Compras
  - Proveedores → UIMDProveedores
  - Pedidos → UIPPedidos
  - Nuevo Pedido → UISNuevoPedido
- Ventas
  - Clientes → UIMDClientes
  - Facturas → UIPFacturas
- Logística
  - Envíos, → UIMDEnvios
  - Nuevo Envío → UISNuevoEnvio
  - Redes de destribución → UIMDRedesDist

Usando la sintaxis matemática recién descrita, representaríamos como sigue el AJA:

$$
\begin{aligned}
AJA \;=\; \langle & \\
& \langle \text{'Compras'}, \\
& \quad \langle \\
& \quad\quad \langle \text{'Proveedores'}, UIMDProveedores \rangle, \\
& \quad\quad \langle \text{'Pedidos'}, UIPPedidos \rangle, \\
& \quad\quad \langle \text{'Nuevo Pedido'}, UISNuevoPedido \rangle \\
& \quad \rangle, \\
& \langle \text{'Ventas'}, \\
& \quad \langle \\
& \quad\quad \langle \text{'Clientes'}, UIMDClientes \rangle, \\
& \quad\quad \langle \text{'Facturas'}, UIPFacturas \rangle \\
& \quad \rangle, \\
& \langle \text{'Logística'}, \\
& \quad \langle \\
& \quad\quad \langle \text{'Envíos'}, UIMDEnvios \rangle, \\
& \quad\quad \langle \text{'Nuevo Envío'}, UISNuevoEnvio \rangle, \\
& \quad\quad \langle \text{'Redes de distribución'}, UIMDRedesDist \rangle \\
& \quad \rangle \\
\rangle &
\end{aligned}
$$

Definición 5.4.26 ***Árbol de jerarquía de acciones efectivo***
Un Árbol de jerarquía de acciones efectivo AJA_{ef} es el subárbol de un árbol de jerarquía de acciones AJA visible para los objetos agentes de la clase A en una vista V (AJA ha sido necesariamente definido en V). Dicho subárbol AJA_{ef} se obtiene como sigue:

1. *Inicialmente: $AJA_{ef}(A, V) = AJA$*
2. *Se eliminan todos los nodos hojas Nod_i pertenecientes a AJA_{ef} cuyas unidades de interacción no visibles para la clase A.*

$$
\forall\, Nod_i \in AJA_{ef}(A,V),\; Eliminar(Nod_i) \iff Nod_i = \langle Alias_i, UI_{dest_i} \rangle \wedge \neg UIVisible(UI_{dest_i}, A, V) \tag{5.29}
$$

3. *Se eliminan todos los nodos Nod_j que queden sin hijos como resultado de la aplicación del punto 2.*

$$\forall\, Nod_j \in AJA_{ef}(A,V),\ Eliminar(Nod_j) \iff Nod_j = \langle Alias_j, \langle\rangle\ \rangle \tag{5.30}$$

El árbol de jerarquía de acciones efectivo $AJA_{ef}(A,V)$ construido de este modo contiene sólo las unidades de interacción que son alcanzables para los objetos agentes instancias de la clase A en la vista V.

De la definición 5.4.26 se deduce trivialmente que $AJA_{ef}(A,V) \subseteq AJA$.

5.4.4. Visibilidad de un agente en una vista

Como colofón a los conceptos introducidos describiremos como la interfaz de usuario final es restringida dinámicamente en tiempo de generación o ejecución en función del usuario conectado al sistema.

1. Un usuario que pretende usar el sistema accede a una aplicación que le proporciona una interfaz de usuario del sistema. Dicha interfaz de usuario implementa una vista V dada del sistema (*véase Definición 5.4.4*) y representa una primera elección (consciente o no, ya que el dispositivo de acceso puede predeterminar la vista a utilizar).

2. Nada más acceder a la aplicación, ésta solicita que el usuario se autentique como un objeto agente del sistema. Para lo cual muestra una lista de clases agentes en dicha vista $C_{agentes}(V)$(*véase Definición 5.4.5*).

3. El usuario selecciona entonces una clase agente A y proporciona un identificador y una prueba de identidad, como podría ser una clave secreta o un dato biométrico. Si la autenticación es errónea, pueden permitirse diversos reintentos o bien, denegar el acceso al sistema. Si la autenticación es satisfactoria, el usuario se ha conectado con éxito al sistema como un objeto agente $\underline{O_{ag}}$ (*véase Definición 5.4.7*).

4. Acto seguido, al usuario se le presenta el árbol de jerarquía de acciones efectivo $AJA_{ef}(A,V)$ en función de la clase agente a la cual pertenece (*véase Definición 5.4.26*).

5. El usuario puede interaccionar con la aplicación y navegar a todas las unidades de interacción UI_j sobre las que tiene permisos de visibilidad $UIVisible(UI_j, A, V)$ (*véase Definición 5.4.14*).

6. Cada conjunto de visualización se han restringido a un conjunto de visualización efectivo en función de la clase agente $CV_{ef}(A, V)$ (*véase Definición 5.4.22*).

7. Cada patrón de acciones se han restringido a un patrón de acciones efectivo en función de la clase agente $Acc_{ef}(A, V)$ (*véase Definición 5.4.23*).

8. Cada patrón de navegación se han restringido a un patrón de navegación efectivo en función de la clase agente $Nav_{ef}(A, V)$ (*véase Definición 5.4.24*).

En resumen, se ha definido la semántica de visibilidad en función de la vista usada y el objeto agente conectado al sistema. Los mecanismos de interfaz y vista permiten disponer de un potente mecanismo de visibilidad que permite construir un mismo modelo de interfaz de usuario para un grupo de clases agentes y posteriormente restringir dinámicamente en tiempo de generación o de ejecución en función de la clase agente conectada al sistema.

El *Modelo de Usuario* definido a través de clases agentes y sus interfaces determina con gran precisión los permisos, responsabilidades y necesidades de cada grupo de usuarios.

5.5. Conclusiones del capítulo

El capítulo actual ha permitido revisar la significado del modelo propuesto en capítulo previo desde diversos puntos de vista complementarios entre sí:

1. En primer lugar, se ha presentado una visión de las propiedades básicas del meta-modelo Just-UI. El cual, completa la especificación basada en meta-modelo presentada para cada patrón en el capítulo previo.

2. A continuación, se ha descrito el modelo en términos algebraicos descubriendo la isormorfía de las partes del modelo respecto a árboles y grafos dirigidos.

3. Posteriormente, los principales patrones han sido descritos formalmente usando la notación para el modelado de tareas CTT.

4. Por último, la semántica de visibilidad introducida en el modelo, ha sido expresada con detalle usando fórmulas lógicas de primer orden. El mecanismo de visibilidad puede verse como una extensión al Modelo de Ejecución en términos de visibilidad de un agente en una vista dada.

Como complemento a este capítulo, cabe citar los apéndices A y B:

- El apéndice A presentan una extensión sintáctica para el lenguaje de especificación formal $\mathcal{OASIS}$ que da cuenta de los patrones presentados.
- Mientras que el apéndice B describe un DTD para la representación en XML de las especificaciones basadas en Just-UI.

Capítulo 6

Inferencia

«El buen carpintero mide dos veces, corta una.»

— Proverbio hondureño.

Este capítulo describe una serie de estrategias introducidas a nivel de modelo, herramienta de modelado y de generadores para soportar el prototipado rápido, no solo de la interfaz de usuario, sino de toda la aplicación que está siendo construida a través de la especificación conceptual. Los resultados de este trabajo fueron presentados en las Jornadas IDEAS 2002. [Molina02a].

6.1. Introducción

En el mundo de la industria, existe una necesidad clara de prototipado rápido que permita la pronta validación de los requisitos con el usuario. Por otra parte, este prototipo debe ser lo más completo posible para que la validación realizada por el usuario sea útil.

Estos dos requisitos están claramente contrapuestos, ya que si se prototipa rápidamente, el prototipo es incompleto y si se crea un prototipo completo no puede hacerse de un modo rápido.

Para intentar paliar este problema se propone en este trabajo de tesis un proceso denominado *Inferencia*, que permite incrementar la velocidad de prototipado sin incrementar el esfuerzo destinado a completar el modelo conceptual.

El proceso de *Inferencia* se realiza sobre el Modelo de Presentación y consiste en derivar la información de interfaz de usuario faltante permitiendo prototipar rápidamente sin la necesidad de especificar por completo el Modelo de

Presentación[1].

6.2. Inferencia

Durante la especificación de un esquema conceptual, el analista modela de forma incremental la información necesaria para la obtención de la aplicación final. Inicialmente se detalla la parte esencial (estática y dinámica básica) y posteriormente se completa de forma iterativa (dinámica compleja y presentación). El proceso de inferencia está constituido por un conjunto de heurísticos que completan la información que el analista no ha especificado en el modelo utilizando la información disponible en el modelo. El proceso de inferencia persigue dos objetivos básicos:

- permitir la obtención de prototipos de interfaz de usuario ejecutables sin la introducción de todos los datos del Modelo de Presentación, y
- beneficiar el *caso más frecuente*, es decir, que el analista especifique únicamente aquellos casos que requieran de cierta funcionalidad especial. El resto de casos con funcionalidad *estándar* se inferirán correctamente y no será necesario especificarlos, se asumirán como comportamiento por defecto.

El proceso de inferencia recibe como entrada un esquema conceptual donde el modelo de presentación está especificado parcialmente o incluso, no ha sido especificado en absoluto. El proceso de inferencia produce como salida un modelo de presentación, en el que se han inferido nuevos componentes y completado datos en los existentes.

A continuación se detallará los distintos casos donde se aplican los procesos de inferencia.

6.2.1. Inferencia de vistas

El prototipo que se le presentará al usuario debe estar siempre basado en una vista. Una vista está compuesta por un conjunto de interfaces entre una clase servidora y una clase agente, donde se indican los atributos que la clase agente puede visualizar, los servicios que puede ejecutar y las relaciones de agregación por las que puede navegar.

[1] En el caso extremo, es posible crear un prototipo sin Modelo de Presentación de partida, infiriéndolo por completo.

Si no se especifica una vista con la que obtener el prototipo, se inferirá una vista por defecto que contiene todas las clases y todas las interfaces definidas en el modelo. Esta vista se usará para la posterior obtención del AJA.

En la vista hay información explícita sobre los servicios que se podrán ejecutar (interfaz), por lo que se conoce cuáles son las UI de Servicio que se le ofrecerán al agente conectado. Por el contrario, no se define información explícita sobre el resto de unidades de interacción, por lo que se presentarán en la vista todas aquellas que aparezcan en el AJA más las que sean accesibles desde éstas, es decir, desde argumentos de servicios, variables de filtro, navegación por agregación o herencia y desde acciones.

6.2.2. Inferencia del árbol de jerarquía de acciones

Si no se ha definido un AJA debe inferirse uno por defecto, que permita al usuario el acceso a todas las unidades de interacción a las cuales tiene acceso desde el escenario principal de la aplicación.

Una vez se tiene una vista se pasa a la inferencia del árbol. Para ello, se crea la raíz del árbol (*nivel 0*) y a primer nivel se añade un nodo por cada una de las clases de la vista etiquetado con el alias efectivo[2] de la clase (*nivel 1*). Por cada nodo intermedio, se crea a su vez tantos nodos hoja como unidades de interacción de la clase existan en la vista, etiquetando cada nodo final con el alias efectivo de la unidad de interacción (*nivel 2*). El orden en el que se añadirán estos nodos hijos será: UI de Instancia, UI de Población, UI de Maestro/Detalle y UI de Servicio.

Para las transacciones globales se añade un nodo de *nivel 1* etiquetado como 'Procesos generales' y como subnodos de éste se agregan las unidades de interacción de servicio de las transacciones globales etiquetadas como las anteriores con el alias efectivo de la unidad de interacción correspondiente.

En el caso en el que existan relaciones de herencia entre clases cada una de las clases descendientes se sitúan en el *nivel 2* incluidas dentro del nodo intermedio creado para su clase ascendiente. De forma recursiva, se añaden sus unidades de interacción y sus clases descendientes.

6.2.3. Inferencia de unidades de interacción

Los siguientes procesos de inferencia se aplican tanto para crear unidades no definidas en el modelo de presentación como para completar información en las unidades existentes (incluyendo también las unidades inferidas).

[2]El alias efectivo para un elemento es el alias definido sobre el elemento si éste existe, y si no al alias inferido para este elemento.

Inferencia de unidades de interacción de servicio

Si un servicio es ofertable[3] al usuario, aparece en la vista, pero no tiene unidad de interacción asignada, se le crea una unidad de interacción de servicio por defecto, que solicitará al usuario los argumentos declarados en el servicio.

Alias Si una unidad de interacción de servicio no tiene alias definido, se le aplica el alias definido del servicio, y si éste a su vez no existe, entonces se le asigna como alias el nombre del servicio.

ARGUMENTOS DEL SERVICIO

Alias Si un argumento no tiene alias definido se le asigna el alias del atributo del que procede. Si bien no se ha podido inferir el atributo origen del argumento, o bien dicho atributo no tiene alias, se asigna el nombre del argumento.

Patrón de introducción Si un argumento dato-valuado de un servicio no tiene patrón de introducción asociado se le aplica el patrón de introducción de su atributo origen. En su defecto, no se aplica ninguno.

Patrón de selección definida Si un argumento dato-valuado de un servicio no tiene patrón de selección definida asociado se le aplica el patrón correspondiente de su atributo origen. En su defecto, no se aplica ninguno.

Información complementaria Si a un argumento objeto-valuado no se le ha asignado información complementaria, se le aplica la información complementaria por defecto de la clase del tipo del argumento. En su ausencia, no se aplica ninguno.

Unidad de interacción de población Si a un argumento objeto-valuado no se le ha asignado la unidad de interacción destino para selección de objetos, se asigna la unidad de interacción de población por defecto de la clase a la que pertenece el tipo del argumento.

Inferencia de unidades de interacción de instancia

Si una clase pertenece a la vista y no tiene, al menos, una unidad de interacción de instancia, se debe derivar una *por defecto* que estará formada por: un conjunto de visualización inferido, un patrón de navegación inferido y patrón de acciones inferido (los tres componentes serán descritos en el apartado 6.2.4).

[3]El servicio está incluido en una interfaz entre el usuario y la clase del servicio y dicha interfaz pertenece a la vista a generar.

Alias Si no tiene alias definido, se aplica como alias el efectivo de la clase[4].

Inferencia de unidades de interacción de población

Si para una clase perteneciente a la vista no se especifica ninguna unidad de interacción de este tipo, se infiere una por defecto.

En esta unidad de interacción se añadirán: todos los filtros definidos en la clase, todos los criterios de ordenación definidos en la clase, conjunto de visualización inferido como *conjunto principal*[5] y como conjuntos alternativos el resto de conjuntos de visualización definidos en la clase, navegación inferida y acciones inferidas.

Alias Si no tiene alias definido, se aplica el alias efectivo de la clase.

FILTRO

Si no se ha definido ningún filtro, se proporcionará un *filtro nulo* que devuelve toda la población de la clase.

Alias Si el filtro no tiene alias, se aplica como alias el nombre del filtro.

VARIABLE DE FILTRO

Las variables de filtro son huecos que el usuario puede completar para restringir la búsqueda. Son variables libres que participan en la fórmula de búsqueda restringiendo la consulta.

Información complementaria Si a una variable de filtro objeto-valuada no se le ha asignado información complementaria, se le aplica la información complementaria por defecto de la clase del dominio de la variable. En su ausencia no se aplica ninguno.

Unidad de interacción de población Si a una variable de filtro objeto-valuada no se le ha asignado la unidad de interacción destino a la que debe saltar, se le asigna la unidad de interacción de población por defecto de la clase a la que pertenece el dominio de la variable.

Inferencia de unidades de interacción maestro/detalle

En este caso, no se realiza inferencia para crear estas unidades, ya que se trata de escenarios muy particulares, que es necesario que sean detallados por el analista explícitamente.

[4] El alias efectivo de la clase es el alias asignado a la clase y en su ausencia, el nombre de la clase.
[5] Primer conjunto de visualización de la U.I. que se presentará al usuario.

Alias Si la unidad de interacción no tiene alias definido, se aplica como alias el de la clase que actúa como maestro. Si la clase que actúa como maestro no tiene alias definido, en su lugar se usa el nombre de la unidad de interacción.

6.2.4. Inferencia de patrones auxiliares

A continuación se detallará la inferencia que se realiza en aquellos patrones auxiliares que constituyen las piezas básicas para la construcción de las unidades de interacción simples, en particular para: conjuntos de visualización, navegación y acciones. Estos patrones por defecto, sólo serán inferidos si son necesarios para completar alguna unidad de interacción.

Inferencia de conjunto de visualización

Si es necesario inferir un conjunto de visualización para una clase, se crea uno al que se añaden como elementos de visualización, todos los atributos de la clase ordenados por el orden de definición de los atributos.

Para favorecer el prototipado rápido, se permite indicar explícitamente en su definición un conjunto de visualización como **(Auto)**. El uso de **(Auto)** en la especificación del Modelo de Presentación es similar al uso del asterisco '*' en las sentencias SELECT de SQL. Al usarlo, el analista se independiza de los atributos de la clase en el momento de la especificación, de manera que si se añaden o eliminan posteriormente siguen estando incluidos en el conjunto de visualización.

ELEMENTOS DEL CONJUNTO DE VISUALIZACIÓN

Se infiere su información faltante del atributo del que proceden. Esta es:

Alias Si un elemento del conjunto de visualización definido no tiene alias, se le asigna el alias definido del atributo del que procede. Si el atributo a su vez tampoco tiene, se deja como alias el nombre del atributo.

Patrón de introducción Si el elemento del conjunto de visualización no tiene patrón de introducción asignado, se intenta aplicar el del atributo origen.

Patrón de selección definida Si el elemento del conjunto de visualización no tiene patrón de selección definida asignado, se intenta aplicar el del atributo origen.

Inferencia de navegación

La inferencia de una navegación se realiza creando una navegación que contiene saltos a todas las relaciones de agregación (tanto propias como heredadas) y relaciones de herencia.

Alias Si un elemento de navegación no tiene alias definido, se asigna el alias en función del tipo de relación por la que navegará.

Si se trata de una relación de agregación, se utiliza como alias, el alias definido del último rol que conforma el path de roles atravesado. Si éste no existe, se empleará en su lugar el alias efectivo de la unidad de interacción destino.

Si la relación es de herencia, se usará como alias el alias definido de la última clase del *path* de herencia y en su ausencia, el alias efectivo de la unidad de interacción destino.

Inferencia de acciones

Si se desea inferir un conjunto de acciones para una clase se crea un patrón que permite el acceso a todos los servicios ofertables al usuario de la clase en cuestión para la vista actual del sistema. Cada salto tiene como unidad de interacción destino, la unidad de interacción del servicio correspondiente. Se debe tener en cuenta que además de los servicios propios de la signatura de la clase, también se deben ofertar los servicios heredados (servicios definidos en la signatura de las clases ascendentes).

Al igual que los conjuntos de visualización y la navegación, se pueden crear acciones tipo **(Auto)**.

ELEMENTOS DE LAS ACCIONES

Los elementos de las acciones, indican a qué unidad de interacción se va a saltar desde la unidad de interacción actual.

Alias Si un elemento de acciones no tiene alias definido se utiliza el alias efectivo de la unidad de interacción destino.

6.3. Caso de estudio

Vamos a presentar la gestión de un aparcamiento de automóviles. El aparcamiento tiene una serie de plazas de garaje que se alquilan por fracción de horas (la primera hora se paga completa). También es posible alquilar una plaza

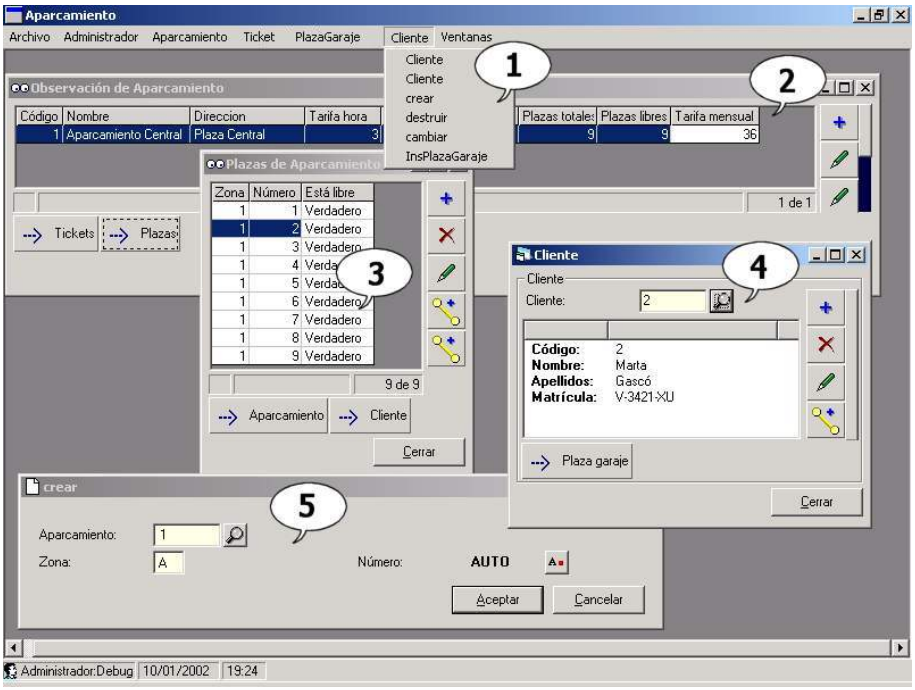


Figura 6.1: Ejemplo: Gestión de un Aparcamiento de Automóviles

de garaje a un cliente a cambio de una cuota mensual. Con este sistema se pretende mantener información actualizada sobre el número de plazas ocupadas para su posterior facturación.

El modelo conceptual que representa este sistema es muy sencillo. Consta de las siguientes clases: `Aparcamiento` (contiene información sobre la ubicación del aparcamiento, así como las tarifas a aplicar), `PlazaGaraje` (almacena la ubicación de la plaza), `Cliente` (almacena información sobre los clientes que tienen un contrato de cuota mensual; a cada cliente se le asigna una plaza de garaje), `Ticket` (representa el ticket que toma cada automóvil al entrar al garaje, registrando la hora de entrada y salida para el cálculo del importe a pagar) y `Administrador` (del sistema).

En el caso de estudio, se ha especificado completamente todo el esquema conceptual a excepción del modelo de presentación, por lo que el prototipo de interfaz de usuario obtenido será completamente inferido.

Tras realizar el proceso de inferencia, se obtiene como resultado una aplicación con un menú principal cuyos elementos a primer nivel son las clases del modelo. En cada submenú (*véase Figura 6.1-1*) se encuentra una entrada para cada formulario traducido[6] para cada clase. Cada tipo de unidad de interac-

[6]Cada Unidad de Interacción da lugar a un formulario

ción genera un tipo de formulario distinto. Cuando la inferencia es total[7] se generan tres tipos de formularios: formularios de población, instancia y servicio (los formularios maestro/detalle proceden de UI especificadas, no inferidas). Los formularios de población (Figuras 6.1-2 y 6.1-3) permiten la interacción con conjuntos de objetos (filtrado, ordenación, etc.). Cada formulario de instancia (Figura 6.1-4) permite visualizar los atributos de un objeto, la navegación a objetos relacionadas y el acceso a los servicios disponibles. Por último los formularios de servicio (Figura 6.1-5) solicitan los argumentos para la ejecución de cada servicio.

De esta manera, el prototipo de la interfaz permite comprobar rápidamente cuestiones tales como la navegación por todas las relaciones definidas en el modelo, visualización de los atributos definidos y la ejecución todos los servicios, así como la comprobación de los permisos de los agentes del sistema.

6.4. Aplicación a un proceso industrial

En CARE Technologies han sido implementados dichos procesos de inferencia en dos vertientes:

En una herramienta de modelado: La herramienta de modelado (que permite construir esquemas conceptuales y especificaciones de interfaz de usuario) soporta el proceso de inferencia. De este modo, puede ir guiando al analista. Muestra qué información será inferida y las consecuencias de dicha inferencia, ahorrando, de este modo, tiempo de especificación para aquello que tiene un comportamiento por defecto.

En traductores de interfaz de usuario: Los traductores de código implementan el proceso de inferencia para completar la especificación con la información faltante antes de proceder a la traducción de la interfaz.

En un estudio[Myers92], Myers prueba que en el desarrollo de aplicaciones, el esfuerzo dedicado al desarrollo de interfaces de usuario supone casi el 50 % del esfuerzo total. En la especificación de modelos conceptuales hemos comprobado que la recogida de los requisitos funcionales corresponde al 60 % del esfuerzo mientras que los requisitos de interfaz de usuario consumen el 40 % restante. Esta última tarea es la que se ve mejorada con las técnicas de inferencia anteriormente descritas.

La especificación de un modelo conceptual transcurre en una espiral de ciclos de construcción y evaluación destinados a verificar que los requisitos

[7] La inferencia total se produce cuando el analista no especifica modelo de presentación.

están correctamente expresados. Estos ciclos concluyen cuando se da la especificación por validada y completa.

En los primeros ciclos, la especificación se centra en recoger y validar los requisitos funcionales. En estos ciclos, el esfuerzo de especificación de interfaz de usuario debe ser el mínimo imprescindible para alcanzar los prototipos. Aquí hemos apreciado que los procesos de inferencia reducen la especificación de UI en un 85 % respecto a una especificación sin inferencia.

Conforme se va alcanzando la especificación final, el analista completa la especificación con los requisitos del usuario. Dicho esfuerzo de especificación sigue siendo menor (respecto a especificación sin inferencia) debido al uso de valores por defecto y el principio de *beneficiar el caso frecuente*. Aquí hemos apreciado una mejora del 50 %.

El uso de estas técnicas para la obtención de aplicaciones ha mejorado la productividad notablemente. Los ciclos de prototipado-evaluación se han acortado ostensiblemente y en menos tiempo han podido llevarse a cabo más iteraciones, lo que redunda en una validación más profunda de los requisitos por parte del usuario.

Este tipo de prototipado no sólo permite obtener interfaces rápidamente, sino que además permite verificar el correcto funcionamiento de la lógica de negocio especificada al ser la interfaz obtenida totalmente funcional y no una mera fachada. Al contrario, el prototipo final puede ser empleado como aplicación final si cumple con los requisitos de calidad.

6.5. Conclusiones del capítulo

El proceso de inferencia permite obtener prototipos más rápidamente respecto a los métodos convencionales. Dichos prototipos ofrecen la funcionalidad completa del sistema de un modo estandarizado para ayudar al analista en la validación de los requisitos del sistema con el usuario.

Debido a que la inferencia es un conjunto de heurísticos, el resultado del proceso de inferencia puede no ser siempre el deseado por el analista. Aún así, en un porcentaje muy alto de casos ofrece un resultado aceptable que permite reducir notablemente el tiempo invertido en la especificación de la interfaz de usuario. Es decir, incrementa notablemente la productividad en su elaboración y además simplifica el modelo resultante.

Capítulo 7

Generación automática

> *«Vivimos en la caverna de Platón. Vivimos observando sombras que se mueven y creemos que eso es la realidad. Hoy la llamamos realidad virtual»*
>
> — José Saramago, escritor y periodista portugués, Premio Nobel de Literatura, 1998.

Al igual que en el «*mito de la caverna*» de Platón, o en el largometraje «*The Matrix*», podemos imaginar la existencia de otro mundo (el mundo de las ideas) que no podemos ver, diferente al mundo material. Podemos crear representaciones de ese otro mundo mediante modelos abstractos, pero nunca llegar a tocarlo.

Es precisamente la capacidad de abstracción, la que nos permite trabajar en ese mundo de las ideas y conocer en términos de clases y relaciones lo que en el mundo real son objetos y enlaces.

Si bien hemos necesitado los seis capítulos precedentes para llegar hasta este nivel de abstracción donde proporcionar descripciones basadas en modelos, ahora ha llegado el momento de descender de nuevo a la caverna para traer a la vida los modelos creados y construir interfaces de usuario reales a partir de ellos.

Los programadores saben bien como interpretar los modelos para construir las herramientas adecuadas. Sin embargo, si podemos automatizar en mayor o menor medida este trabajo, estaremos aumentando su productividad de modo apreciable. El presente capítulo tiene como objetivo presentar las técnicas y arquitecturas para la generación de interfaces de usuario orientadas a construir aplicaciones a partir del modelo de especificación propuesto en el capítulo 4.

A principios de los años ochenta, Balzer [Balzer83] en un artículo publicado en *IEEE Computer* vaticinaba el auge de la generación de código con el Paradigma de la Programación Automática (*Automatic Programming Paradigm*). Actualmente la industria tiende a evolucionar las herramientas CASE que asisten en la gestión y diseño del código fuente hacia esquemas más evolucionados como MDA (*Model Driven Architecture*) [OMG01]. Los resultado de la conferencia UML World 2001 [Weber01] pronostican que en los próximos años habrá un movimiento significativo en el mercado hacia la ejecución de modelos (*Executable Models*).

Hay muchos trabajos en el campo de generación automática de código que logran obtener buenos resultados para dominios pequeños y restringidos [Cleaveland01, Czarnecki00, Genera00, Arg02]. Sin embargo, los dominios más complejos (como la producción completa de aplicaciones) suelen escapar al mundo de la generación automática por algunos de los siguientes motivos:

1. falta de modelos sencillos que supongan un ahorro de tiempo (ley de Novak, página 98),
2. falta de expresividad o adecuación del modelo para describir el problema,
3. complejidad de los generadores o
4. falta de flexibilidad para adaptar el código producido.

7.1. Objetivos

Los objetivos perseguidos a lo largo de este capítulo son los siguientes:

1. producir automáticamente interfaces de usuario para aplicaciones,
2. garantizar la calidad del código producido,
3. soportar el prototipado rápido e
4. incrementar notablemente la productividad del proceso de desarrollo.

Estos objetivos son descritos con más detalle en los siguientes apartados.

7.1.1. Generación automática

Se pretende que un generador proporcione todo el código necesario en un lenguaje de tercera generación listo para compilar (o interpretar) y ejecutar sin

retoques adicionales. De este modo, se pretende reducir al máximo el retoque manual del código por parte de programadores. Con esta estrategia se favorece que el mantenimiento del software se haga sobre el esquema conceptual en lugar de hacerlo directamente sobre el código fuente generado.

La interfaz generada deberá implementar todos y cada uno de los requisitos reflejados en el esquema conceptual del cual deriva. Esta interfaz de usuario implementa el cliente de una arquitectura cliente/servidor clásica; por lo tanto, también contendrá código de enlace o comunicación con la parte servidora y gestionará los errores que en la comunicación que pudieran producirse.

7.1.2. Interfaces de alta calidad

El generador de código proporcionará interfaces de usuario de alta calidad. Este objetivo es alcanzado siguiendo las siguientes pautas:

- El generador no comete errores. Una vez traducido de modo exitoso un concepto abstracto a su correspondiente implementación, dicha traducción es llevada a cabo por el generador de manera ciega una y otra vez sin introducir errores a los que son propensos los programadores humanos. Por tanto, el código generado para los casos probados está 100 % libre de errores de programación, lo cual redunda en la calidad de la aplicación final.
- El generador seguirá escrupulosamente una guía de estilo adecuada al entorno destino elegido. De este modo, no solo se consigue una homogeneidad dentro de la aplicación, sino también con el resto de aplicaciones del entorno destino. La homogeneidad de las interfaces de usuario incide directamente en el grado de ergonomía alcanzado y tiene impacto sobre la facilidad de aprendizaje, contribuyendo todo ello a mejorar la calidad de la aplicación.
- El generador sigue una serie de patrones de traducción o casos de traducción, donde para cada caso bien conocido, se ha determinado una configuración en el espacio de la solución que lo resuelve (*véase capítulo 4*). De este modo, problemas similares en el espacio del problema, tienen respuestas similares en el espacio de la solución aportando de este modo un grado más de homogeneidad.

7.1.3. Prototipado rápido

A la vez que se construye el esquema conceptual, puede emplearse el generador de interfaz para obtener prototipos. Estos prototipos pueden ser mostra-

dos al usuario final para ayudar en el proceso de validación de requisitos.

El generador puede emplearse para realizar prototipación vertical u horizontal, en función de sí el analista decide especificar con mucho detalle una parte del sistema, o bien prefiere comenzar especificando todo el sistema a un nivel de abstracción mayor, respectivamente.

7.1.4. Mayor productividad

Una vez completado el ciclo de análisis del sistema, se obtiene un esquema conceptual completo que describe mediante el sistema real.

Si todos los requisitos del sistema han podido ser expresados en en modelo conceptual, los prototipos generados en esta fase pueden considerarse como aplicación final sin cambios adicionales, en el extremo, puede constituir la interfaz de usuario final lista para ser entregada.

Si por el contrario, no todos los requisitos han podido ser especificados o traducidos a código, deberemos implementarlos de modo manual e integrarlos con el código generado. En aplicaciones reales desarrolladas en CARE Technologies hemos encontrado que porcentaje de código que necesita ser modificado a mano ha sido reducido a valores marginales (*véase capítulo 8*). Este hecho ha permitido aumentar espectacularmente la productividad al emplear muy poco tiempo en tareas de codificación manual.

7.2. Ventajas y retos de la generación automática de código

El empleo de técnicas de generación automática de código presenta las siguientes ventajas frente a una producción manual:

- **Mayor productividad:** Cuando el esfuerzo de especificación y generación es menor que el coste de producción manual, la generación merece la pena.
- **Menos errores:** La generación automática evita los errores de codificación, evitando por tanto, esfuerzo de depuración.
- **Más calidad:** El código producido es más homogéneo y estructurado. Generando comentarios al código, esté puede estar a su vez mejor documentado y ser más fácil de mantener.
- **Proceso estandarizado:** El proceso es repetible, auditable, controlable. Lo cual permite establecer mejores controles de calidad. El código producido

es similar al resto de su misma familia (otro código producido del mismo a partir de otro modelo). Los desarrolladores sólo tienen que aprender un modo estandarizado de trabajar y las herramientas de soporte pueden adaptarse para explotar dicha estructuración.

Sin embargo, también existen contrapartidas que debemos sopesar:

- **Dominio limitado:** El generador de código suele estar orientado a un dominio restringido más o menos amplio. Fuera del dominio original, el código producido puede no ser adecuado.
- **Expresividad limitada:** Un generador de código que produce código está limitado por la potencia expresiva del modelo de especificación empleado. Es posible incrementar el nivel de complejidad del modelo de especificación, sin embargo, conllevará un incremento de la complejidad de modelado y de construcción de los generadores asociados.
- **Falta de flexibilidad:** Un generador puede ser parametrizado en función de las preferencias de un usuario o del problema particular a resolver. Sin embargo la parametrización puede conllevar serios inconvenientes:
  - Una parametrización demasiado flexible en *tiempo de generación* permite al diseñador introducir cambios que pueden reducir la calidad del código generado o incluso introducir inadvertidamente errores.
  - Una parametrización demasiado flexible en *tiempo de diseño* (código generado) permite al diseñador alterar muchas variables del código producido a costa de una mayor complejidad en el código producido (y por tanto, una mayor dificultad de mantenimiento).
  - Una parametrización demasiado flexible en *tiempo de ejecución* (parametrización de usuario final) permite al usuario alterar parámetros del programa sobre la marcha. La complejidad del código producido se incrementa ostensiblemente puesto que lo que antes era considerado estático se convierte en dinámico y paramétrico.

7.3. Fundamentos de generación

La presente sección introducirá los siguientes principios básicos para la generación como son:

1. la separación de responsabilidades,
2. los tiempos de decisión y
3. arquitecturas tipo para generación de código.

7.3.1. Separación de responsabilidades

Uno de los conceptos clave de la generación de código [Cleaveland01] es la definición de interfaces de abstracción entre especificaciones e implementaciones (*véase Figura 7.1*). La separación de responsabilidades (*Separate of Concerns*) permite dividir el trabajo en módulos mínimamente acoplados y fuertemente cohesionados.

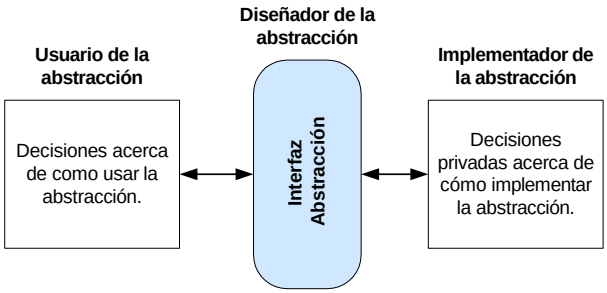


Figura 7.1: Abstracción vía interfaces (adaptado de [Cleaveland01]).

7.3.2. Tiempos de decisión

Todo proceso de reificación (traducción) implica el paso de un nivel de abstracción a otro nivel de abstracción inferior. En este paso se han de tomar las decisiones adecuadas para proporcionar los detalles necesarios en el segundo nivel.

Este descenso a un nivel de menor abstracción implica un aumento de la entropía: del mismo modo que cuando un terrón de azúcar se deshace al disolverse en un vaso de agua, el cambio de nivel supone introducir más desorden en el sistema considerado. Más aun, tal vez no sea posible la vuelta atrás, al estado anterior. Es decir, no siempre es posible la reingeniería inversa completa, sino solo estructural y parcial.

La toma de decisiones es, por tanto, clave en el proceso de generación de código. No sólo importa *cómo* se toma la decisión, sino *quién* (el cliente, el analista, el diseñador, el programador, el administrador o el usuario final) y *cuándo* (en que nivel se toma la decisión).

Algunos instantes temporales donde pueden tomarse las decisiones aparecen reflejados en la figura Figura 7.2.

Tiempo de ingeniería de dominio Hace referencia al periodo de estudio y desarrollo de las herramientas para automatizar el dominio. En esta fase se identifican las propiedades y se clasifican como constantes o variables.

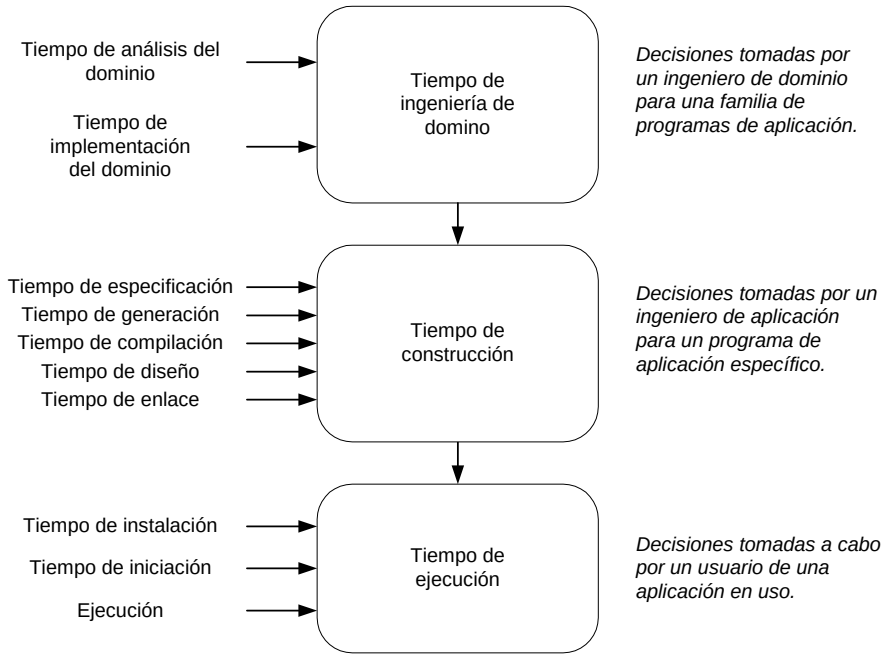


Figura 7.2: Instantes temporales de decisión (adaptado de [Cleaveland01]).

Tiempo de construcción Incluye la fase de ingeniería de aplicación donde se especifica, genera, compila, diseña y enlaza la aplicación. En todos estos instantes pueden tomarse decisiones respecto a la aplicación.

Tiempo de ejecución Por último, la fase final compete al administrador de sistemas y al usuario final que pueden tomar decisiones en tiempo de instalación, de iniciación (la aplicación arranca) y ejecución.

7.3.3. Arquitecturas para generadores

Un esquema muy básico de producción de aplicaciones usando tecnología de generación de código se muestra en la Figura 7.3. La especificación se usa como entrada para el generador, el cual produce el código fuente del programa. Este, a su vez, es compilado y enlazado con otras librerías para producir el programa ejecutable final.

La Figura 7.4 muestra posibles tres alternativas para incorporar la parte variable (especificación) en el programa.

- La opción A. muestra como el programa se compila y la especificación es usada como meta-datos interpretados en tiempo de ejecución. La ventaja

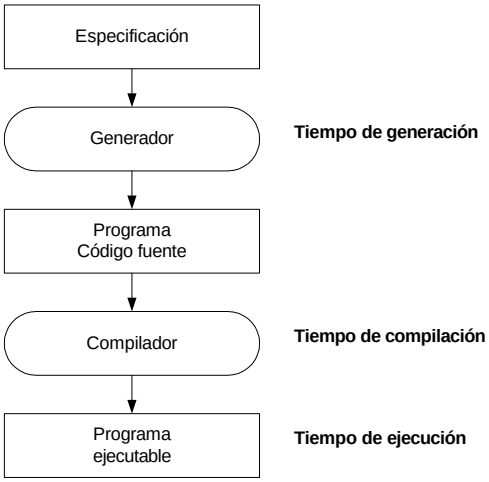


Figura 7.3: Esquema básico para generación de código.

principal de este primer método es que los meta-datos pueden ser cambiados sin necesidad de recompilar la aplicación. Tenemos entonces un **interprete de modelos**.

- La opción B enlaza código común y parte variable en tiempo de compilación para producir la aplicación. No es tan versátil como la primera opción, pero el compilador, por contra, permite detectar errores en la parte variable en tiempo de compilación.
- Por ultimo, la opción C integra un generador de código basado en plantillas. Las plantillas dan lugar a documentos (ficheros) al ser instanciados por los meta-datos o parte variable. Una vez generados, los ficheros son compilados para producir la aplicación final.

El mundo de la industria tiende a almacenar la información de diversos modelos usando tecnología XML [W3C00]. XML es un formato muy versátil para el almacenado de datos y para aplicar transformaciones sobre esos datos. Disponer de los meta-datos o la especificación en XML presenta grandes ventajas: existe muchas herramientas estandar para trabajar con este formato.

En particular, la lectura o carga de la información basada en XML es asistida por API de analizadores léxicos (*scanners*) como SAX (*Simple Access XML*) y léxico-sintácticos (*parsers*) como DOM (*Document Object Model*).

La Figura 7.5 muestra tres ejemplos generadores donde la especificación está codificada en XML. Por norma general una carga basada en DOM es más lenta que una carga basada en SAX, sin embargo, los cargadores SAX son más complejos de mantener.

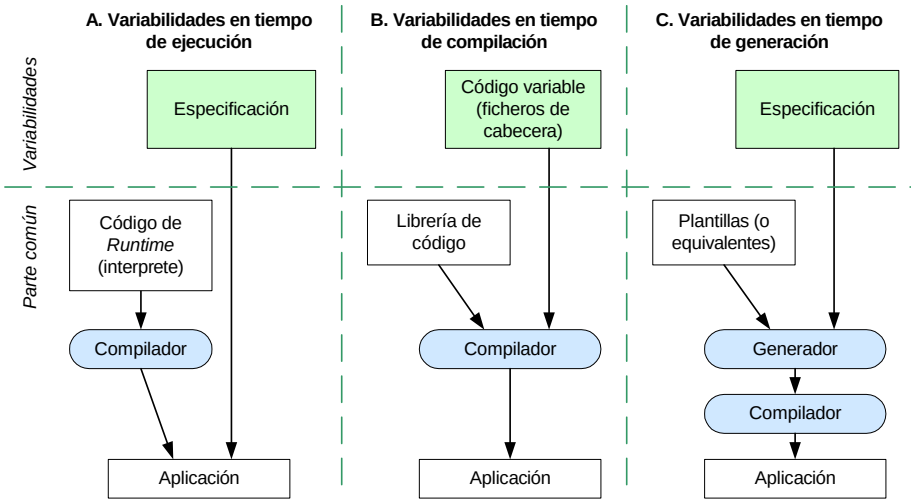


Figura 7.4: Alternativas a la evaluación de las variabilidades (adaptado de [Cleaveland01]).

- La opción A. hace uso exclusivo de DOM para contruir una representación arborescente del modelo en memoria y generar código a partir de este árbol directamente. La desventaja principal radica en que el código del generador depende del API DOM.
- La opción B. es similar a la primera opción planteada donde se ha añadido una etapa de transformación y manipulación de la información para construir unas estructuras intermedias, independientes de la tecnología de carga.
- Por último, la opción C. presenta un esquema similar a la opción A. donde se ha empleado un cargador SAX para llevar a cabo la tarea de carga. Presenta la ventaja de que las estructuras intermedias tampoco dependen de la tecnología XML.

7.4. Técnicas de generación

Una vez presentados los fundamentos, la presente sección describirá una serie de técnicas de generación. Dichas técnicas han sido subdividas en dos grandes grupos:

1. técnicas generales, aplicables en principio a cualquier proceso de generación y

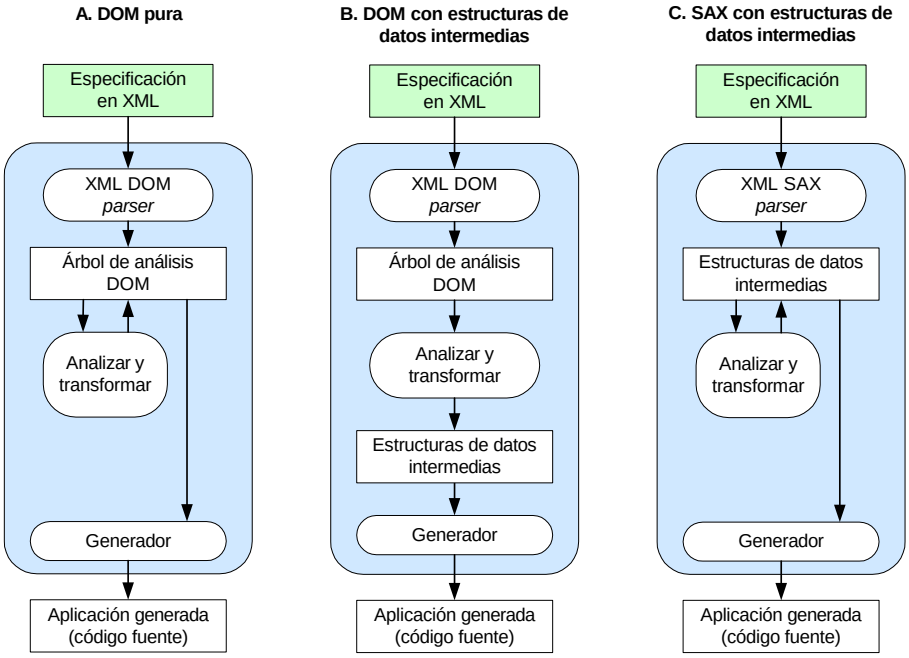


Figura 7.5: Procesado de especificaciones basadas XML y generación (adaptado de [Cleaveland01]).

2. técnicas específicas para interfaces de usuario que tienen en cuenta las peculiaridades del código destino a producir.

7.4.1. Técnicas generales

Las técnicas presentandas son:

1. Clonación,
2. Concatenación de cadenas,
3. Reglas de reescritura,
4. Plantillas,
5. Autómatas finitos y
6. Análisis de expresiones mediante gramáticas.

Clonación

Si el código destino a producir contiene ficheros que permanecen constantes independientemente del modelo a traducir, la traducción puede llevarse a cabo por medio de *clonación*, o copia directa del fichero origen al directorio de generación. Las librerías de funciones genéricas o ficheros binarios representando imágenes o iconos constantes pueden ser tratados de este modo: construidos una única vez y añadidos mediante un proceso de copia al producto final.

Concatenación de cadenas

La concatenación de cadenas es un modo sencillo de ir construyendo el código destino desde un lenguaje de programación clásico (p.e. véase tabla 7.1).

```
text += "Begin\n"
text += "   Print " + GetData("Name")    + "\n";
text += "   Print " + GetData("Surname") + "\n";
text += "   Print " + GetData("Points")  + "\n";

                          ⋮

text += "End\n"

file.open()
file.Save(text);
file.Close();
```

Tabla 7.1: Ejemplo de generación por concatenación de cadenas.

El nombre proviene del proceso de ir concatenando cadenas de texto (*strings*) que contienen todo el código a producir. Finalmente, la cadena es volcada a un fichero.

Herramientas como System Architect [Sys01] o Rational Rose [Rat02] proporcionan lenguajes de script (normalmente Basic o derivados de este) extendidos con un API para la lectura de la información de modelo. De este modo es factible producir código interpretando los scripts en Basic. Al proporcionar generadores en código abierto (script Basic) resulta relativamente sencillo retocar el código producido alterando el generador para adaptarlo a las necesidades especificas.

Se dispone de toda la potencia del lenguaje de programación para determinar que código debe ser generado, permitiendo cálculos, sustituciones y proce-

sados complejos. Sin embargo, entre las desventajas más sobresalientes figura la necesidad de proteger los caracteres de control propios del lenguaje empleado para la generación. Por ejemplo: dentro de una cadena de texto en C o C++ no es posible usar el carácter comillas " (indicaría el final de la cadena), para su empleo, es necesario protegerlo con el carácter de escape \". Del mismo modo, deben protegerse otros caracteres como por ejemplo el retorno de carro (\n). El mayor inconveniente de estas protecciones radica en que dificulta la lectura del código objetivo dentro del código del generador.

```
genCode << "<cfoutput>if (#value#!=\""+code+"\"){\n";
genCode << "               write(\"<A>\"); \n";
genCode << "          }else{\n";
genCode << "               write(\"</A>\");\n";
genCode << "          }</cfoutput>\n";
```

Tabla 7.2: Duro ejemplo de traducción a varios niveles.

Como anécdota sufrida por el propio autor cabe comentar que durante el desarrollo de un traductor para Web se formaba una cadena como muestra la tabla 7.2). Dicha tabla ilustra un hipotético código C++ que:

1. genera una cadena para el lenguaje ColdFusion,
2. la cual generará JavaScript en tiempo de ejecución en el servidor, que será enviada al navegador del cliente,
3. el código Javascript será interpretado en el navegador del cliente, dando como resultado un elemento de HTML,
4. el cual será finalmente interpretado de nuevo por el navegador.

Este pequeño ejemplo, aunque extremo, es real y demuestra la dificultad que supone reconocer y comprender con claridad qué va ha ser generado leyendo el código del generador. La protección de los caracteres de control como las dobles comillas \" y los retornos de carro \n propios de la sintaxis de C++ no facilitan la legibilidad, al contrario, dificultan su mantenimiento. ¡Los programadores están obligados a conocer y mostrar habilidad en la sintaxis y los caracteres de escape de cuatro lenguajes de programación de modo simultáneo![1]

[1]Sin duda alguna, estos virtuosos sufridores deberían estar bien pagados: es una labor muy poco gratificante mantener este tipo de código.

Reglas de reescritura

Sistemas como Oblog [ESDI93] emplean un lenguaje de script de generación más próximo a reglas de reescritura como las empleadas en los lenguajes de programación declarativos como Haskell o Prolog. Los generadores están constituidos por ficheros que contienen reglas de reescritura. Un motor basado en *pattern-matching* y sustitución se encarga de ejecutar la generación dirigida por reglas. Busca las reglas que encajan (*match*) y realiza la sustitución correspondiente hasta que ya no pueden ser aplicadas ninguna regla más.

La facilidad de modificar el código del traductor se favorece, basta con sustituir el conjunto de reglas por otras. Sin embargo, la legibilidad del traductor obtenido es muy cuestionable. Las plantillas están ocultas, codificadas tras las reglas, dificultando por tanto su mantenimiento.

Plantillas

La generación mediante plantillas puede ser empleada cuando el código destino a producir tiene un patrón de repetición bien caracterizado, de modo que todos los ficheros generados son idénticos salvo los datos procedentes directamente del modelo a generar. En este caso puede definirse una plantilla genérica a partir de ejemplares del código objetivo. En esta plantilla, las dependencias del modelo son sustituidas por marcadores (o huecos) con nombre único.

Aparejado a la plantilla, podemos definir un proceso de generación que, dado un elemento de modelo, la plantilla sea instanciada a código mediante un proceso de sustitución de cadenas: los marcadores son sustituidos por los datos de modelo correspondiente.

Los elementos involucrados en esta aproximación son los siguientes (*véase Figura 7.6*):

1. *Documentos* en un lenguaje dado (objetivo de la generación).
2. Una *plantilla* de documento que da cuenta de la parte común y de los puntos donde aparecen las variabilidades.
3. Un *modelo* o *especificación*, que describe las variabilidades.
4. Una *gramática* o *meta-modelo* que establezca las reglas de formación de los modelos.
5. Un *algoritmo de transformación* que traduce la plantilla en un documento a partir de la información de un modelo.

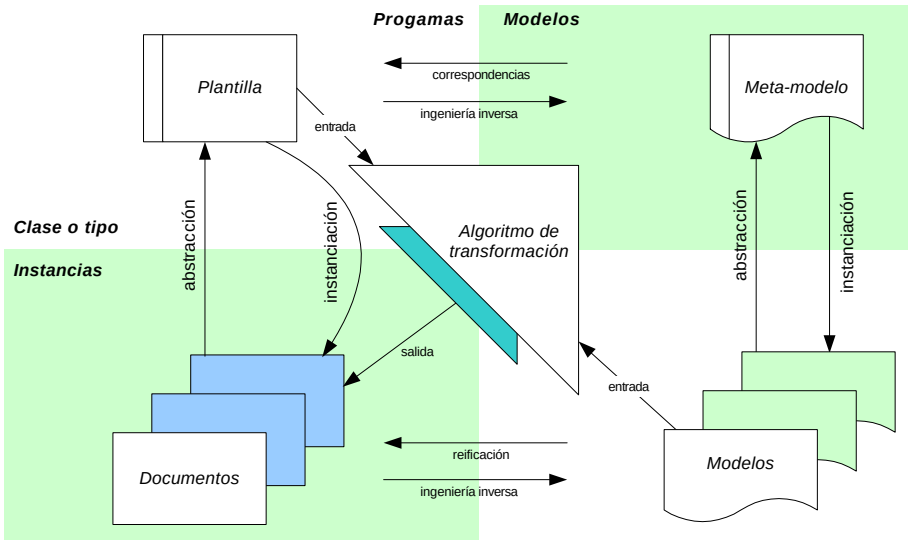


Figura 7.6: Elementos en la generación mediante plantillas.

Este pequeño ejemplo ilustra la generación basada en plantillas:
Documento Doc_1:

```
Estimado Sr. Pablo Molina:
    ha sido agraciado con 8 puntos.
```

Documento Doc_2:

```
Estimado Sr. Justo López:
    ha sido agraciado con 7 puntos.
```

Plantilla P_1 donde se ha separado la parte común y la parte variable. Los marcadores para partes variables han sido denotadas y nombradas usando marcas del tipo: `<Name>`.

```
Estimado Sr. <Nombre> <Apellido>:
    ha sido agraciado con <Puntos> puntos.
```

Sea $Spec$ el modelo correspondiente a las variabilidades de los documento:

```
Spec{
    Persona
    {
        Name:       Pablo
        Surname:    Molina
        Points:     8
    }
    Persona
    {
```

```
        Name:       Justo
        Surname:    López
        Points:     7
    }
}
```

Sea G_1 una gramática expresada en BNF para las especificaciones de la forma Spec:

```
Spec         := personaBlock*
personaBlock := Persona { nameBlock surnameBlock pointsBlock }
nameBlock    := Name: <id>
surnameBlock := Surname: <id>
pointsBlock  := points: <number>
```

Por último, sea A_1 el algoritmo de generación asociado:

```
1   ForEach p = Persona in Spec Do
2       Doc = InstanciateTemplate(p, P1)
3       Doc.Substring ('<Nombre>',   p.Name)
4       Doc.Substring ('<Apellido>', p.Surname)
5       Doc.Substring ('<Puntos>'  , p.Points)
6   End ForEach
```

En el algoritmo A_1 se realiza un recorrido para cada persona definida en el modelo (líneas 1-6). Durante el bucle se crea una instancia de documento a partir de la plantilla P_1 (línea 2). Las líneas 3-5 realizan las substituciones de los marcadores en el documento a partir de los meta-datos (información del modelo). Finalmente el procesado se repite para cada elemento de modelo a tratar en la generación (en este caso para cada *Persona*) (línea 6).

Las especificaciones de la forma $Spec$ que cumplen con la gramática G_1 pueden ser usadas como entrada junto con la plantilla P_1 por el algoritmo A_1 para generar los documentos de la forma indicada: $Doc_1, Doc_2, \ldots, Doc_k$

Este ejemplo puede refinarse con facilidad para dar mayor potencia al lenguaje de plantillas:

- soportando operaciones para realizar cálculos en las substituciones y
- dar al lenguaje de sustitución más potencia para soportar bucles, búsquedas y reemplazamiento condicional.

Antes de pasar a describir una nueva técnica, merece la pena detenerse para comentar dos implementaciones de generación basada en plantillas: transformaciones XSL y Velocity.

A. Transformaciones XSL Dentro del mundo de XML, existe un lenguaje estándar para la transformación de documentos: XSL (*Extensible Stylesheet Language*) [W3C01]. Esté lenguaje tiene como principal característica la necesidad de definir en un mismo documento la plantilla y las reglas de transformación simultáneamente. Precisamente por estos motivos, la legibilidad y la facilidad de mantenimiento de dichos documentos no es toda lo buena que sería deseable.

Si trasladamos el estudio presentado para las plantillas al mundo dirigido por XML y XSL podemos repasar como quedan configurados los cinco elementos que necesitamos con los siguientes estándares listos para ser usados:

1. **Documentos** en un lenguaje dado (objetivo de la generación) Puede ser XML o cualquier otro lenguaje destino.

2. Documento XSL. **Plantilla** de documento.

3. Documento XML. Hace las veces de **modelo** o especificación, que describe las variabilidades.

4. Fichero DTD o Schema que define las **reglas de formación** de los modelos (3).

5. El **algoritmo de transformación**. Codificado con sintaxis XSL dentro del propio documento plantilla (2).

La mezcla de la plantilla (2) y el algoritmo de transformación (5) en un solo documento XSL provoca que la plantilla o patrón pierda su forma original. Deja de ser evidente su forma y ya no permite presagiar el resultado. Si a eso unimos la extraña sintaxis empleada en XSL para codificar las transformaciones, tenemos un lenguaje poco amistoso para un lector/mantenedor humano.

Esté método es perfectamente válido para generar código. JSP, ASP o PHP pueden ser usados del mismo modo, embebiendo instrucciones de control dentro de una plantilla. Sin embargo, esta aproximación presenta los siguientes puntos desfavorables:

- la ya comentada, dificultad de mantenimiento y

- la ligazón entre plantilla y algoritmo de transformación. Un desacople entre uno y otro permitiría reutilizar plantillas con diversos algoritmos o mejor aún, aplicar un mismo algoritmo a diversas plantillas (generación a diversas plataformas).

B. Velocity Como respuesta los problemas presentes en XSL para su aplicación a la generación de código, han aparecido lenguajes de plantillas alternativos como Velocity [Group.02a] o los lenguajes de pantillas personalizados como sugiere Cleaveland [Cleaveland01]. En Velocity, a diferencia de XSL, la plantilla y algoritmo de transformación han sido desacoplados. Cada uno de ellos se especifica por separado, en ficheros diferentes.

1. Velocity proporciona un lenguaje de plantillas VTL (*Velocity Template Language*) para el desarrollo de plantillas. En dicha plantilla aparecen los caracteres de control imprescindibles para indicar los marcadores (o huecos de la plantilla) y su nombre.
2. El algoritmo de transformación es implementado un lenguaje de 3^a generación como es Java. Aquí puede usarse toda la potencia de Java para calcular las transformaciones necesarias a los datos e instanciar la plantilla.

La comparativa entre las dos aproximaciones comentadas XSL y Velocity nos permite argumentar porqué la segunda opción es preferible para estrategias de generación de código. La razón fundamental proviene de la separación de responsabilidades[2] que se obtiene al desligar la plantilla del algoritmo de transformación:

1. La plantilla preserva con mayor facilidad la forma del documento a generar.
2. Los algoritmos son más legibles al no aparecer mezclados con los datos.
3. Las dos propiedades previas redundan en una mayor facilidad de mantenimiento.
4. Permite la convivencia del trabajo de diseñadores y programadores y el trabajo en paralelo entre ambos grupos. (Una propiedad muy demanda en el desarrollo de aplicaciones Web). Los diseñadores gráficos o expertos en el lenguaje destino pueden concentrarse en el diseño de plantillas. Mientras que los ingenieros expertos en generación de código pueden concentrarse en los algoritmos de transformación.
5. Permite el reuso de plantillas variando el algoritmo de transformación (*útil para la generación a partir de diversas fuentes de datos –modelos–*).
6. Permite el reuso de algoritmos de transformación variando las plantillas aplicadas (*útil para la generación a diversas plataformas*).

[2] *Separate of Concerns*: Esgrimida como uno de los principios al comienzo de este capítulo.

Autómatas finitos

Los diagramas de estados basados en máquinas de Harel [Harel87] pueden ser transformados en autómatas finitos dirigidos por tablas. Ejemplos de ellos son, por ejemplo, lenguajes como SDL (*Telecommunications Standard Specification and Description Language*) [SDL99] o por metodologías como ROOM (*Real-time Object-Oriented Method*) [Selic94] se basan su generación de código a partir de diagramas de estados.

Análisis de expresiones mediante gramáticas

Existen ocasiones donde las estrategias previas de generación se vuelven insuficientes. En particular, un caso muy claro es el del tratamiento de expresiones que siguen una gramática dada. Aquí las técnicas de compilación clásicas (Aho, Sethi y Ullman [Aho86]) son las más adecuadas para producir el código necesario.

La expresión puede ser convertida en una estructura con forma arborescente en memoria: Un analizador léxico, sintáctico y semántico realizan esté trabajo. Después, un algoritmo puede recorrer dicho árbol que representa la expresión para analizarla y decidir el tipo de código a producir. El algoritmo incluso puede aplicar optimizaciones si el caso lo permite.

7.4.2. Técnicas para interfaces de usuario

A la hora de generar interfaces de usuario, debemos considerar aspectos específicos del dominio diferentes a los que podrían darse en el ámbito de generación de componentes de lógica de negocio, de capas de persistencia o documentación, por citar algunos ejemplos.

Aspectos como la ergonomía, facilidad de uso, homogeneidad, coherencia, facilidad de aprendizaje, etc. deben ser consideradas a la hora de producir interfaces de usuario. Para garantizar un mínimo de calidad se hace imprescindible el empleo de guías de estilo [Vanderdonckt01]. Como ejemplo, el apéndice E (página 331) contiene una guía de estilo para plataforma Windows.

A continuación se describen las estrategias más empleadas en la generación de código para interfaces de usuario. En particular se describirán:

1. Orientación a componentes
2. Reificación AIO → CIO
3. Modelos MVC, PAC y ARCH

Orientación a componentes

Las interfaces de usuario no son realizadas de modo completamente artesanal, dibujando directamente sobre el dispositivo gráfico. Al contrario, el diseñador de interfaces dispone de una serie de componentes prediseñados (en ocasiones incluso pueden ser extendidos) que sirven de ladrillos básicos para la construcción de interfaces.

De este modo, todas las herramientas para el diseño de interfaces de usuario modernas proporcionan etiquetas, editores de texto, botones, paneles, contenedores, barras de desplazamiento, etc.

Estos componentes facilitan la creación de las interfaces de usuario. De un modo simplista, y como primera aproximación desde el punto de vista estático, podríamos asemejar una interfaz de usuario a un árbol de continencia de componentes. Cuando un componente tiene una serie de subcomponentes hijos, se dice que el componente actúa de contenedor de los componentes hijos. Los componentes hijos están físicamente representados dentro del área del componente padre o contenedor.

Cada componente de interacción encapsula no sólo aspectos de presentación, sino también de comportamiento y funcionalidad. El reuso de componentes en la interfaz de usuario redunda en un alto grado de homogeneidad en la interfaz de usuario. Los usuarios aprenden la funcionalidad de un componente una sola vez. Si la presentación es homogénea, recordarán como interactuar con el componente cuando vuelvan a encontrarse con él.

La interfaz de usuario no es completa sin una descripción de su comportamiento dinámico. Conforme el usuario interacciona con la aplicación, los componentes pueden alterar su estado y comunicarse entre sí. Este comportamiento y sincronización entre componentes debe ser programado en la aplicación para dar vida a la interfaz de usuario.[3]

En ocasiones, un determinado componente prefabricado no satisface los requisitos deseados. En estos casos, puede ser recomendable crear un nuevo componente adaptado a las necesidades particulares de la aplicación.

Por término general, cuanto más ricos y especializados son los componentes, más sencilla y dirigida se vuelve la construcción de interfaces de usuario para un dominio dado. Por contra, el precio a pagar es la libertad de movimientos del diseñador que puede verse restringida al disponer de un conjunto de primitivas más limitado y específico.

A la hora de generar código para interfaces de usuario complejas, poder disponer de las primitivas adecuadas (componentes de interacción adaptados a las tareas a solventar) se convierte en un factor crítico de éxito.

[3] También llamada *lógica de cliente*.

La reificación AIO → CIO

Un CIO [Vanderdonckt93] (*Concrete Interface Object*[4])es un control o *widget*[5] o componente gráfico empleado en un entorno de ventanas dado para construir una interfaz de usuario. Los CIO son dependientes de una librería o implementación dada. Por ejemplo: en la librería Swing [Walrath99] disponible en el lenguaje Java, el control JTextField implementa un campo de entrada donde el usuario puede introducir texto. En el lenguaje Visual Basic, existe un control similar para realizar el mismo tipo de funciones: TextBox.

Por contra, un AIO [Vanderdonckt93] (*Abstract Interface Object*[6]) es una abstracción de un control que lo desliga de consideraciones de implementación y diseño. En estos términos, podemos considerar la existencia de AIO que abstrae la funcionalidad esencial de los controles. Por ejemplo, un AIO para la entrada de texto como abstracción de JTextField (Java/Swing) y TextBox (Visual Basic 6.0).

La dualidad AIO/CIO ha permitido abstraer las propiedades relevantes de los elementos de una interfaz de usuario. A nivel de modelado, el razonamiento en términos de AIO en lugar de CIO permite desligar la especificación de una implementación particular haciéndola, por tanto, más portable y reusable.

Esta buena propiedad de los AIO puede ser satisfecha si y solo si aparejado a un modelo basado en AIO se proporcionan las correspondencias o traducciones para cada AIO a sus correspondientes CIO según la plataforma destino.[7]

Modelos MVC, PAC y ARCH

El patrón MVC [Goldberg83, Bergin02] apareció en ADA en la década de los 80.[8] Consiste en desacoplar la funcionalidad de una aplicación (*Modelo*) de sus interfaces de usuario (*Vistas*). Los (*Controladores*) sirven de mediador y sincronizador entre Vistas y Modelo.

PAC (*Presentation, Abstraction, Control*) [Coutaz87] es un modelo propuesto por Coutaz para la descomposición de la interfaz de usuario de nuevo en tres aspectos (Presentación, Abstración y Control). En esta arquitectura y a diferencia de MVC el objeto de Control sólo media entre el objeto de Abstracción y el de Presentación.

[4]Objeto concreto de interfaz.
[5]Del inglés: *window* + *gadget*.
[6]Objeto abstracto de interfaz.
[7]El problema de establecer correspondencias entre modelos abstractos y concretos puede encontrarse en la literatura referido como «*the mapping problem*». Consúltese Puerta y Einsenstein [Puerta99] para una excelente exposición del problema.
[8]MVC: *Model-View-Controller*, Modelo-Vista-Controlador.

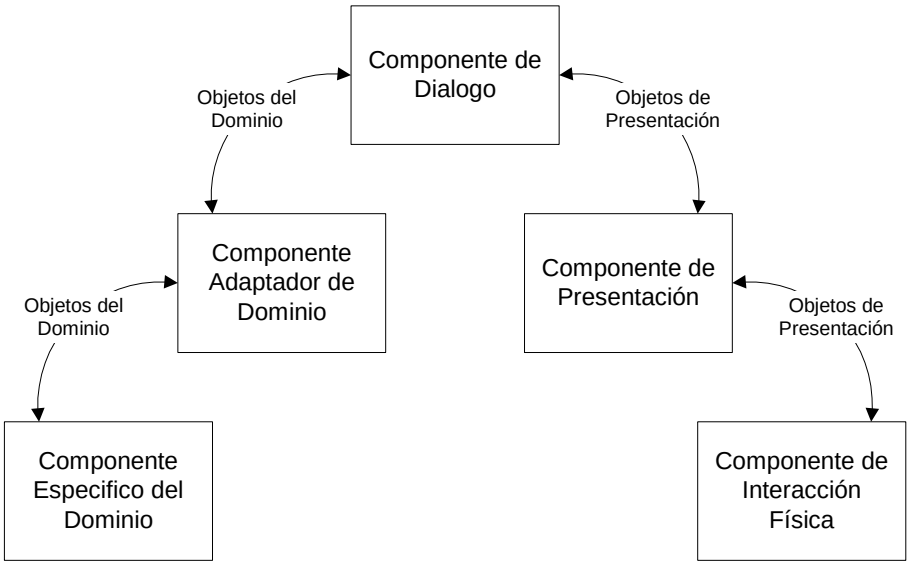


Figura 7.7: Meta-modelo Arch (adaptado de [IPI92]).

Por otro lado, el meta-modelo Arch [IPI92] es uno de las meta-modelos más genéricos para aplicaciones interactivas. Podemos compararlo con el modelo PAC donde la arquitectura ha sido refinada. Arch resalta cinco componentes que pueden apreciarse en la Figura 7.7. Podemos ver Arch como un refinamiento de PAC:

1. El **Componente Específico del Dominio** controla, manipula y recupera información del dominio y encapsula toda la funcionalidad del sistema (en términos de PAC: *Abstracción*).

2. El **Componente de Interacción Física** implementa la interacción física con el usuario final haciendo uso para ello de librerías gráficas o de controles propios del entorno donde se implementa el sistema (en términos de PAC: *Presentación*).

3. El **Componente de Diálogo** tiene responsabilidades a nivel de secuenciación de tareas, tanto para la parte correspondiente al dominio de la aplicación como para la que depende del usuario (en términos de PAC: *Controlador*).

4. El **Componente de Presentación** media entre el Compopnente de Diálogo y el Componente de Interacción Física que proporciona un conjunto de herramientas y objetos de interacción independientes del nivel físico de presentación para ser empleados por el Comp. de Diálogo.

5. Por último el **Componente Adaptador de Dominio** media entre el Comp. de Diálogo y el Comp. Especifico de Dominio.

7.5. Lenguajes intermedios para interfaz de usuario

En el siguiente apartado se describen cinco lenguajes basados en XML para la descripción de interfaces de usuario.

Desde que se presento el estándar XML [W3C00], muchas han sido las aplicaciones que este metalenguaje ha tenido en diversos campos de la ingeniería del software. El campo de las interfaces de usuario no se ha mantenido al margen. Al contrario, se ha sacado también partido de XML para definir lenguajes capaces de describir interfaces de usuario.

7.5.1. AUIML

AUIML [Azevedo00] (*Abstract User Interface Markup Language*) es un lenguaje basado en XML diseñado para permitir describir la interacción con el usuario. Se permite una codificación independiente de dispositivo. Consiste en un lenguaje que abstrae los detalles de implementación para describir la interfaz de usuario en términos de AIO en lugar de CIO.

7.5.2. UIML

UIML [Abrams99a, Abrams99b] es un lenguaje de descripción de interfaces de usuario basado en XML. Permite construir una descripción de una interfaz de usuario de una manera declarativa e independente de dispositivo. Para crear una interfaz de usuario, se debe escribir un documento UIML que incluye los estilos de presentación adecuados a los dispositivos finales. UIML establece automáticamente las correspondencias a un lenguaje disponible en en la plataforma destino, como pueden ser HTML, Java/JFC, WML, HTML, PalmOS o VoiceXML.

7.5.3. XUL

XUL [Ginda00] es un lenguaje de descripción de interfaces de usuario basado en XML y Javascript. Está especialmente diseñado para aplicaciones de red como navegadores, agendas, programas de correo, etc. Gracias a la portabilidad del su interprete (o *render*) Gecko y el uso de la tecnología XPCOM, un diseño XUL puede ser mostrado en diferentes plataformas.

Lenguaje	Modelos	Salida	Información de diseño	Metodología	Soporta XML	Herramienta de soporte	Orientación
UIML	Diálogo Presentación Dominio (parcial)	UI totalmente funcional	Ninguno	Ninguna	Sí	Generadores	Abierta
AUIML	Diálogo Presentación	UI totalmente funcional	Ninguno	Ninguna	Sí	Intérprete	Abierta
XUL	Diálogo Presentación	UI totalmente funcional	Ninguno	Ninguna	Sí	Intérprete	Aplic. de red
XFORMS	Presentación Lógica Datos	Formularios Web	Lenguaje de diseño	Ninguna	Sí	Intérprete	Formularios Web
XIML	Tareas Dominio Usuario Diálogo Presentación Plataforma Diseño	UI totalmente funcional	Guías de diseño Heurísticos	Basado en modelos	Sí	Editor Generadores Intérprete	Abierta
Just-UI	Tareas Dominio Usuario Diálogo Presentación	Aplicación totalmente funcional	Guías de estilo Heurísticos Patrones de diseño	OO-Method *OASIS* Métodos formales	Sí	Editor Generadores Intérprete	Sistemas de información

Tabla 7.3: Comparativa de lenguajes para la descripción de IU (adaptado de [Pribeanu01]).

7.5.4. XFORMS

XFORMS [Group02b] es una propuesta en fase de estandarización propuesto por el World Wide Web Consortium (W^3C). Define el nuevo estándar para la definición de formularios en la Web.

7.5.5. XIML

XIML [Puerta02, Software02] (*eXtensible Interface Markup Language*) es un lenguaje extensible basado en XML para soportar multiples modelos de especificación de interfaces de usuarios. Una especficación XIML puede conducir a una interpretación en tiempo de ejecución o a una fase de generación de código en tiempo de diseño.

La Tabla 7.3 muestra una comparación entre los diferentes lenguajes de descripción de Interfaces de Usuario (adaptada de [Pribeanu01]) donde se ha incluido la propuesta presentada en esta tesis: Just-UI.

7.6. Entornos

En esta sección se pretende caracterizar los diferentes dispositivos en función de los recursos disponibles para construir interfaces de usuario.

7.6.1. Entornos de ventanas

Los tradicionales entornos WIMP, o entornos de ventanas proporcionan un rico conjunto de primitivas como son las librerías gráficas, controles estándar. Al mismo tiempo, los terminales de sobremesa permiten interaccionar con el sistema empleando pantallas de 14 pulgadas o superiores, teclado y un dispositivo de tipo puntero como puede ser un ratón o una pantalla táctil.

Dentro de este tipo de entornos podemos clasificar a las principales plataformas de sobremesa como: Windows, Apple OS X, X11, o aplicaciones Java.

Estos entornos se caracterizan por:

- El número de recursos (memoria, disco, recursos gráficos, controles) es considerable y por tanto no suelen existir limitaciones graves a la hora de construir interfaces de usuario.
- Existen guías de estilo estrictas acerca de la presentación y disposición de los elementos en la interfaz de usuario.
- Pueden proporcionar una interacción muy rica con el usuario.
- El trabajo de diseño gráfico (artístico) en la interfaz de usuario es poco o moderado.

7.6.2. Entornos hipermediales

Por el contrario, existen un segundo tipo de entornos que han cobrado mucho más auge con la llegada de Internet. Los sistemas hipermediales distribuyen la información entre nodos de una red. La interfaz de usuario deja de tener como gránulo de trabajo la ventana, para pasar a ser la página Web. Ejemplos de estos dispositivos son la Web en sí misma o intranets que pueden plasmarse en diferentes interfaces de usuario según el dispositivo empleado: Navegadores de escritorio, WebTV, dispositivos móviles, teléfonos con tecnología WAP, etc.

Este tipo de entornos se caracterizan por:

- Los recursos y primitivas de trabajo están fuertemente restringidos: empleo de los elementos (*tags*) de HTML, Javascrit, CSS, Flash, etc.
- Diversos navegadores soportan diversos subconjuntos de la tecnología. Lo cual obliga a tomar una de las dos estrategias siguientes:
  - Emplear solamente el máximo común denominador, es decir, aquella tecnología de mínimos soportada por todos los navegadores: HTML, Javascript y CSS. Esta estrategia minimiza el coste de mantenimiento, pero puede conducir a pobres interfaces de usuario.
  - Emplear toda la tecnología que se considere necesaria y discriminar en tiempo de ejecución el tipo de navegador del cliente para emplear la tecnología adecuada. Esta estrategia, aunque puede incrementar considerablemente la calidad de la interfaz construida, sufre de un coste de mantenimiento muy elevado debido a la necesidad de reimplementar y mantener sincronizadas partes de la interfaz de usuario en diversas tecnologías excluyentes.
- Existen algunas guías de estilo acerca de la presentación y disposición de los elementos en la interfaz de usuario. Sin embargo, el mundo corporativo requiere de guías de estilo personalizadas para integrar la apariencia (*look & feel*) de la aplicación con la apariencia del sitio web corporativo.
- Por el motivo recién mencionado, el papel de los diseñadores y artistas gráficos tiene una gran relevancia en este area (a diferencia de las interfaces de escritorio donde su trabajo es menos necesario).
- Una aplicación distribuida en la red puede sufrir problemas de retroalimentación continua debido a las inevitables latencias y tiempos de retardo provocados por la red.

7.6.3. Entornos embebidos

Por último, cabe mencionar un entorno que por sus particularidades merece la pena describir como un entorno diferente. En los últimos años se están popularizando los dispositivos móviles como teléfonos, asistentes personales (PDA), agendas electrónicas, etc.

Estos dispositivos presentan las siguientes características para el desarrollo de interfaces de usuario:

- El número de recursos (memoria, disco, recursos gráficos, controles, tamaño de la pantalla) es limitado. La aplicación, y en particular la interfaz de usuario, debe diseñarse teniendo en cuenta estas limitaciones.

- Existen guías de estilo estrictas acerca de la presentación y disposición de los elementos en la interfaz de usuario encaminadas a ahorrar el espacio de pantalla que se convierte en uno de los principales problemas.
- Pueden proporcionar una interacción limitada con el usuario. Por este motivo, la interfaz de usuario debe ser lo más sencilla posible.
- El trabajo de diseño gráfico (artístico) en la interfaz de usuario es poco o moderado. Sin embargo, el trabajo relativo al estudio de la ergonomía de la aplicación debe ser redoblado.

7.7. Requisitos de construcción

Antes de proceder a describir con más detalle un generador, es preciso describir el producto final que se pretende obtener con el empleo del generador. La presente sección cubre esta necesidad, describiendo los requisitos que ha de cumplir el producto final proporcionado por el generador.

7.7.1. Entorno de las aplicaciones producidas

Para comenzar a fijar el entorno de trabajo de las aplicaciones que implementan la interfaz, comenzaremos fijando la plataforma destino empleada. Dentro de los entornos gráficos de usuario nos encontramos las siguientes plataformas: Windows de Microsoft Corp., System 10 de Apple, X11 (Gnome, KDE y otros) para entornos Unix, AWT o Swing para plataformas Java y HTML y lenguajes de script para navegadores Web.

La mayoría de ellos, disponen de las herramientas adecuadas para la realización de aplicaciones gráficas que explotan todos los conceptos WIMP (*Window, Icon, Menu, Pointer*). El único que está limitado en parte y que tiene características especiales es el entorno Web: el canal tiene una serie de restricciones aparejadas a la tecnología empleada, las aplicaciones se funden con los contenidos de los portales y suelen tener una estética diferenciada y una sencillez de operatoria mayor.

Por motivos industriales, de cuota de mercado y de soporte a la programación en el entorno, la plataforma Windows fue la seleccionada en primera instancia como entorno destino para las aplicaciones generadas con el primer generador de interfaces de usuario desarrollado en CARE Technologies. Posteriormente, traductores para Java y para entornos Web fueron también desarrollados.

Una vez fijados los entornos, se barajaron diversos lenguajes de programación y herramientas RAD que se adecúen a la manipulación de interfaces de

usuario. Delphi, C++ Builder, Visual C++, Power Builder y Visual Basic fueron considerados los candidatos por excelencia para la plataforma Windows por su fuerte orientación a componentes y desarrollo rápido de interfaces de usuario. Finalmente se optó por la solución Visual Basic debido a la facilidad de edición de las interfaces y a que éstas se almacenan en ficheros de texto plano,[9] lo cual simplifica la generación de interfaces a la mera producción de ficheros planos de texto estructurado. La existencia previa de un generador de código para lógica de negocio que produce código Visual Basic ActiveX permitió a su vez, poder emplear la interfaz como cliente nativa de estos componentes servidores producidos por este otro generador.

Los traductores Web se implementaron usando tres de los lenguajes más implantados para soluciones Web: JSP, ASP y ColdFusion. Las diferencias entre ellos son básicamente de sintaxis entre los lenguajes.

7.7.2. Funcionalidad requerida

Las aplicaciones generadas para implementar la interfaz de usuario soportarán de las siguientes características:

- La interfaz de usuario a generar constituye la tercera capa, dentro de una arquitectura tres capas. Existirá por tanto, un subsistema de comunicación capaz de enviar peticiones a la segunda capa –lógica de negocio– y de recibir y procesar las respuestas a las peticiones realizadas.
- Existirá un sistema de autentificación de usuarios previo al uso de la aplicación.
- Se proporcionará un menú de acceso a la aplicación en función de las capacidades y permisos del agente conectado al sistema.
- El usuario deberá poder ejecutar todos aquellos servicios de los cuales tiene acceso en la vista actual.
- En el ámbito de ejecución de un servicio, el sistema proporcionará valores predeterminados a los argumentos, validará los datos introducidos por el usuario (como son: tipo, rango, existencia de objeto, etc.) e invocará la activación de un servicio a través del envío de un mensaje a la segunda capa.
- Adicionalmente, se deberán controlar los errores que pudieran producirse a raíz de las acciones realizadas por el usuario o derivadas de la comunicación con el componente servidor.

[9] Formato ASCII, no binario.

7.7.3. Integración con el resto de la aplicación

La interfaz de usuario constituye, sin duda, el aspecto más visible de las aplicaciones. Sin embargo, por sí solas, son completamente inútiles sin una serie de componentes adicionales que implementen los requisitos del sistema.

Desde el punto de vista de la *separación de responsabilidades*, podemos hablar de la interfaz de usuario y de la funcionalidad de la aplicación.[10]

Desglosando un poco más, y siguiendo una división típica en capas, podemos hablar de interfaz de usuario, lógica de negocio y capa de persistencia de datos. Esta división, ha demostrado ser una respuesta arquitectónica exitosa para la construcción de sistemas de información de tamaño medio a grande, donde las necesidades de rendimiento y escalabilidad son esenciales.

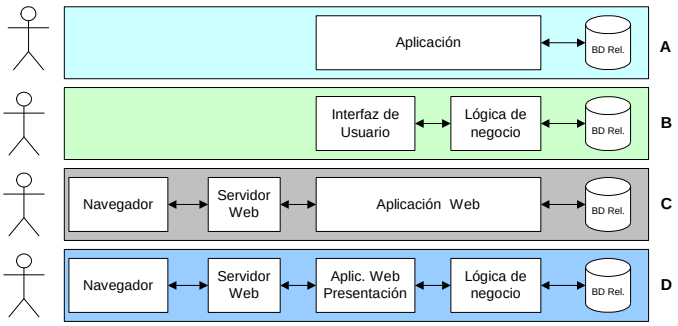


Figura 7.8: Arquitectura en diversas capas para sistemas de información.

Sin embargo, existen otras alternativas de distribución en capas a considerar. Las cuatro arquitecturas más usadas a la hora de desacoplar una aplicación para un sistema de información pueden verse la Figura 7.8:

- **A**. La primer arquitectura conocida como *monolítica* encapsula la lógica de negocio e interfaz de usuario en un solo componente. El único desacople se encuentra en la capa de persistencia, encargada de almacenar la información.

- **B**. Una segunda arquitectura separa lógica de negocio de interfaz de usuario lo cual permite desacoplar dos aspectos fundamentales: funcionalidad e interacción. Ambos componentes pueden ser desarrollados simultáneamente. De este modo, los componentes de lógica de negocio pueden escalarse o ser replicados com mayor comodidad.

[10]Separación de responsabilidades: traducción libre del inglés: *Separate of Concerns*

- **C**. El tercer ejemplo de arquitectura, representa una aplicación basada en Web. Aunque se han introducido nuevas capas correspondientes al navegador y al servidor Web para soportar la distribución, esta aproximación está más próxima a la arquitectura **A** que a la arquitectura **B**. Esto es debido a que la presentación y la lógica de negoció vuelven a confundirse en una sola capa: la Aplicación Web. (Aplicaciones Web basadas en ColdFusion, JSP, ASP o PHP que acceden directamente a la capa de persistencia son claros ejemplos de este tipo de arquitectura).
- **D**. Finalmente, y también para aplicaciones Web, la arquitectura **D** vuelve a disponer de modo separado lógica de negoció y capa de presentación. Si bien, ha de tenerse en cuenta que esta arquitectura es sensiblemente más compleja que la arquitectura **B** debido a la capa de presentación propiamente dicha aparece distribuida (física y lógicamente) a lo largo de tres capas.

A la vista de los pros y contras planteados para estas arquitecturas, se decidio emplear arquitecturas de tipo **B** para aplicaciones de escritorio y aplicaciones de tipo **D** para aquellas soportadas en Web. De este modo, la escalabilidad de la lógica de negocio no se verá condicionada.

La Figura 7.9 muestra una representación gráfica de la arquitectura seguida para las aplicaciones producidas automáticamente. En la figura aparecen reseñadas las tres grandes capas o niveles:

1. **Capa de persistencia.** Puede ser implementada por un sistema gestor de bases de datos (SGBD): relacional, basado en XML, objetual, etc.
2. **Capa de lógica de negocio.** Implementa la funcionalidad de la aplicación. Las plataformas más frecuentemente usadas incluyen objetos COM+/MTS, objetos Enterprise Java Beans (EJB) o código C++.
3. **Capa de interfaz de usuario.** Implementa la capa visible por el usuario proporcionando mecanismos de interacción y presentándole los resultados. Esta capa puede aparecer físicamente dispersa, como por ejemplo en las interfaces basadas en Web donde parte de la lógica de la interfaz puede ser ejecutada en el servidor de aplicaciones y servidor Web y el resto ser ejecutada en el navegador cliente.

Al mismo tiempo, la Figura 7.9 muestra a modo de barras horizontales etiquetadas con letras los diferentes protocolos de comunicación que permiten las interconexión de las capas en sus diferentes niveles.

a. La comunicación entre la capa de persistencia y la capa de lógica de negocio puede llevarse a cabo mediante protocolos estándares diseñados al efecto como ODCB o JDBC.

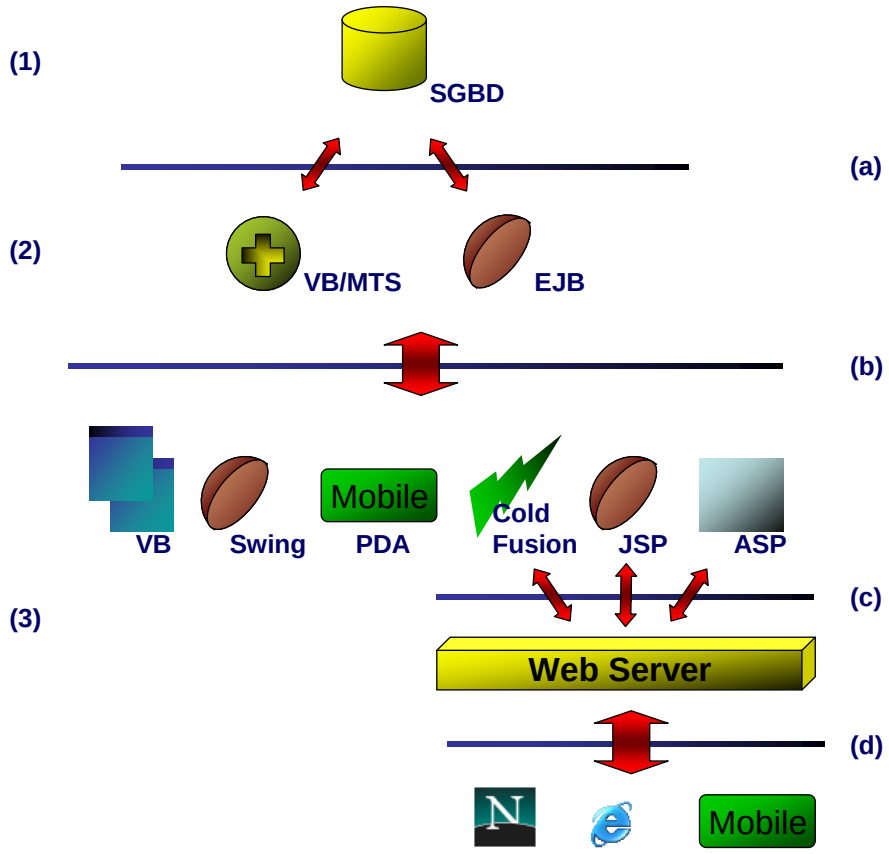


Figura 7.9: Arquitectura n-capas empleada para las aplicaciones producidas.

b. La comunicación entre la capa de lógica de negocio y la capa de interfaz de usuario puede llevarse a cabo mediante mecanismos de RPC como COM+, Java/RMI, CORBA o basados en XML como SOAP o XProtocol.

c. Para disponer de interfaces de usuario en la Web, es necesario que el servidor de aplicaciones (p.e. ColdFusion Server, librería ASP o librería de JSP) se comunique con el servidor Web empleado. Dependiendo del servidor Web, diferentes métodos de comunicación deben ser usados (NSAPI, ISAPI, Apache API, CGI, FastCGI, etc.).

d. Por último, las aplicaciones Web hacen uso de comunicación basada en el protocolo HTTP o su variante segura HTTPS para establecer la comunicación entre el navegador y el servidor Web.

Sea como fuere, es necesario implementar mecanismos para la comunicación entre la interfaz de usuario con el resto de la aplicación.

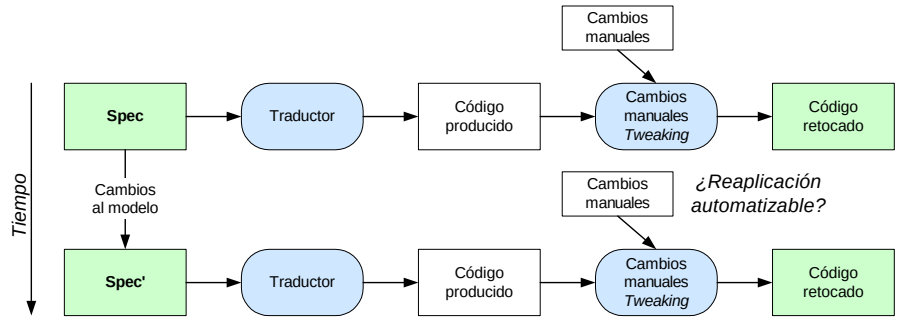


Figura 7.10: Problema del *Round-Trip*.

Un método adecuado para llevar a cabo esta tarea es el establecimiento de una interfaz o API (*Application Programmer Interface*) que será implementada y publicada por el módulo de funcionalidad de la aplicación. Dicho API publica los siguientes elementos:

1. Cada uno de los servicios ofertados por la lógica de negocio para alterar el estado del sistema.

2. Funciones para la consulta de la información por parte de la interfaz de usuario.

De este modo, el componente de interfaz de usuario puede acceder a la funcionalidad publicada y disparar dicha funcionalidad cuando está resulte necesaria.

7.7.4. Integración con cambios manuales

Uno de los problemas a considerar seriamente al usar aproximaciones basadas en generación de código es el problema de los cambios manuales realizados sobre el código generado [Bergman02].[11] El problema aparece cuando la especificación cambia y el código es vuelto a generar, entonces los cambios manuales (*Tweaking*) se pierden y deben ser aplicados de nuevo (*véase Figura 7.10*).

Sin embargo, es posible dar soporte a este problema mediante la gestión de versionado de código fuente y mecanismos de sincronización soportado por herramientas.[12]

[11] *Round-Trip Problem.*

[12] En esta línea en CARE Technologies se ha desarrollado una aplicación de estas características para uso interno y así reducir los efectos del *Round-Trip Problem.*

El código que se pretende obtener como aplicación final está fuertemente condicionado por la plataforma a emplear.

7.8. Traductores implementados

En esta sección describiremos la familia de los generadores de código implementados en CARE Technologies para la generación de interfaces de usuario para diversas plataformas:

- Visual Basic (solución de escritorio sobre S.O. Windows),
- Java/Swing (solución de escritorio sobre máquina virtual Java),
- ColdFusion (solución Web sobre ColdFusion Server),
- JSP (solución Web sobre Java Server Pages),
- ASP (solución Web sobre Active Server Pages) y
- PocketPC (solución para dispositivo móvil basado en S.O. Windows CE) (esté último en desarrollo).

Como ejemplo de implementaciones, se mostrarán las particularidades de tres implementaciones diferentes.

- La primera de ellas corresponde a una implementación de la interfaz de usuario sobre Visual Basic 6.0 sobre plataforma Windows.
- La segunda de ellas corresponde a una implementación sobre ColdFusion Server 4.0. Esta implementación está basada en tecnología de páginas Web, lo cual la hace apta para su uso en Internet.
- Por último, existen también una propuesta para generar interfaces de usuario para dispositivos móviles PocketPC a partir de modelos Just-UI [Belenguer02].

Existe una estructura general en la que se basan todos los traductores. Esta estructura será descrita en la siguiente sección.

7.8.1. Estructura general

Todos los traductores implementan las siguientes fases en este orden:

1. Carga,

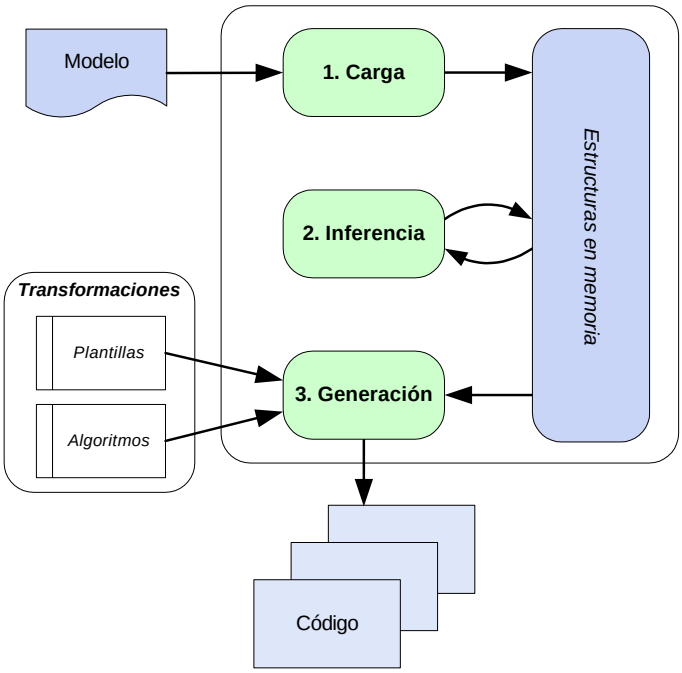


Figura 7.11: Fases de traducción.

2. Inferencia y

3. Generación.

La Figura 7.11 muestra las diferentes fases en el proceso de generación. Con más detalle, se describe a continuación cada fase.

1. Carga

La fase de carga consiste en la lectura desde un repositorio (puede ser un fichero binario, XML, una base de datos, o cualquier otra capa de persistencia) del modelo a traducir y crear una representación de esté en memoria. La representación de este modelo en memoria, no tiene porque ser completa (cargar toda la información de modelo), ni tampoco seguir la misma estructura del meta-modelo. Al contrario, la carga y las estructuras pueden ser adaptadas para cargar sólo la información necesaria y disponerla del modo que sea más conveniente para la tarea de traducción a realizar. A estas estructuras construidas en memoria con la información del modelo las denominaremos *«estructuras del modelo en memoria»* o meta-datos.

2. Inferencia

Tal y como se describió en el capítulo 6, se han definido una serie de mecanismos de inferencia que completan la información de modelado faltante. Las estructuras del modelo en memoria son completadas y extendidas. Es necesario llevar a cabo este proceso antes de proceder a la generación propiamente dicha. El proceso (descrito en el capítulo 6) es responsable de realizar precálculos útiles y de disponer adecuadamente la información para la fase posterior.

3. Generación

La última fase es la de generación propiamente dicha. En función de código destino a producir, se recorren secuencialmente, y de modo anidado, los elementos de modelo en sucesivas pasadas. Por ejemplo: para cada clase, para cada servicio, para argumento, generar un argumento de función Visual Basic. Existen ficheros constantes que son añadidos al código destino sin cambios. Estos pueden ser añadidos por un mero mecanismo de copia. Otros requieren de técnicas más sofisticadas como las descritas previamente: generación basada en plantillas o generación basada en análisis y generación de expresiones en una gramática dada.

7.9. Estrategias de traducción de los patrones

Las interfaces de usuario a generar ha sido prototipadas en sucesivas ocasiones para seleccionar las correpondencias más adecuadas, sopesando para ello criterios como adecuación de la solución a la funcionalidad deseada, usabilidad, escabilidad (en número de elementos o complejidad), eficiencia, facilidad de mantenimiento, etc.

Las Tablas 7.4 y 7.5 muestras las principales correspondencias entre los conceptos del modelo y las representaciones empleadas en el espacio de la la solución para implementar su comportamiento. La primera columna corresponde a conceptos y patrones del modelo de especificación. La segunda columna corresponde con la correspondencia empleada en Microsoft Visual Basic 6.0 sobre plataforma Windows para entornos de escritorio, mientras que la tercera corresponde a las elecciones tomadas para aplicaciones Web implementadas con Macromedia ColdFusion MX.

Aspecto de modelo	**Componente Visual Basic 6.0**	**Componente Web ColdFusion MX**
Vista	Ventana MDI	Página con marco (*frames*)
Árbol de jerarquía de acciones	Menú de aplicación	Árbol Javascript / Componente Flash
Unidad de interacción	Formulario hijo MDI	Página Web
UI de servicio	Formulario hijo MDI con controles de entrada	Página Web con formulario: `<Form>`
UI de instancia	Componente genérico `Instance`	Página web mostrando etiquetas
UI de población	Componente genérico `Population`	Página Web mostrando filtros, tabla de datos, acciones y navegación
UI de maestro/detalle	Implem. por composición de los componentes `Instance` y `Population`	Página Web compuesta
Argumento de servicio de tipo simple	Control de entrada	`<InputBox>`
Argumento de servicio de tipo objeto-valuado	Control genérico OIDSelector	Composición de texto e `<InputBox>`
Filtro	Control `Filter`	Plantilla filtro
Variable de filtro	Control de entrada	`<InputBox>`
Acciones	Botonera	Conjunto de links
Ítem de acción	Botón	`<A HREF="">`
Navegación	Botonera	Conjunto de links
Ítem de navegación	Botón	`<A HREF="">`
Cjto. de visualización	Rejilla o secuencia de etiquetas	`<Table>` o secuencia de texto
Patrón de introducción	Código Visual Basic para la validación de datos	Código Javascript para la validación de datos
Patrón de Selección Definida	Lista desplegable (*ComboBox*) o Botón de radio	`<SELECT>`

Tabla 7.4: Ejemplos de correspondencias de traducción 1/2.

Aspecto de modelo	**Componente Visual Basic**	**Componente Web ColdFusion MX**
Lanzamiento de servicio	Botón etiquetado como "Aceptar" más código VB de validación e invocación de servicio	`<InputBox type="Button">` etiquetado como "Aceptar" más código javascript de validación e invocación de servicio
Cancelación	Botón etiquetado como "Cancelar" más código VB de cierre de formulario	`<InputBox type="Button">` etiquetado como "Cancelar" más código VB de cierre de formulario
Invocación de servicios	Código VB de comunicación con capa de lógica de negocio para invocación de servicios	Código Javascript de comunicación con capa intermedia ColdFusion Server más código CFScript de comunicación con la capa de lógica de negocio para invocación de servicios
Petición de consultas	Código VB de comunicación con capa de lógica de negocio para solicitud de consultas	Código Javascript de comunicación con capa intermedia ColdFusion Server más código CFScript de comunicación con la capa de lógica de negocio para solicitud de consultas
Presentación de datos	Código VB de recuperación y formateado de los datos	Código CFScript de recuperación y formateado de los datos

Tabla 7.5: Ejemplos de correspondencias de traducción 2/2.

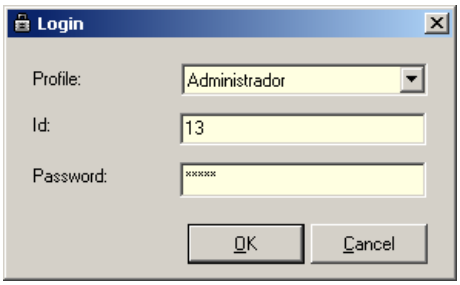


Figura 7.12: GUI de escritorio 1/3.

7.10. Ejemplos de aplicaciones producidas

Esta sección pretende ejemplificar el aspecto de tres aplicaciones basadas en los patrones conceptuales de interfaz de usuario sobre tres entornos bien diferentes:

- Entorno de escritorio.
- Aplicaciones web.
- Aplicaciones móviles para PDA.

7.10.1. Aplicaciones de escritorio

Como ejemplo de aplicación se muestra en las Figuras 7.12, 7.13 y 7.14 una aplicación completamente generada de modo automático para Microsoft Visual Basic 6.0 para entornos Windows. La aplicación modela una biblioteca donde se recoge información sobre libros, autores materias, localización de los libros por estanterías y gestión de prestamos.

La Figura 7.12 muestra la ventana inicial de la aplicación donde los usuarios son validados. Pueden conectarse al sistema lo usuarios y los bibliotecarios.

La Figura 7.13 es un ejemplo de Unidad de Interacción de Población donde se muestran los libros disponibles en una determinada estantería. Nótese como dicha ventana esta compuesta a partir de dos componentes más sencillos. Un componente para unidades de interacción de instancia que muestra la estantería actúa como maestro y un componente para unidades de interacción de población como detalle para mostrar los libros de dicha estantería.

Por último la Figura 7.14 muestra un ejemplo de implementación de unidad de interacción de servicio. El servicio ilustrado es el empleado por los bibliotecarios para mantener actualizada la información relativa a los libros.

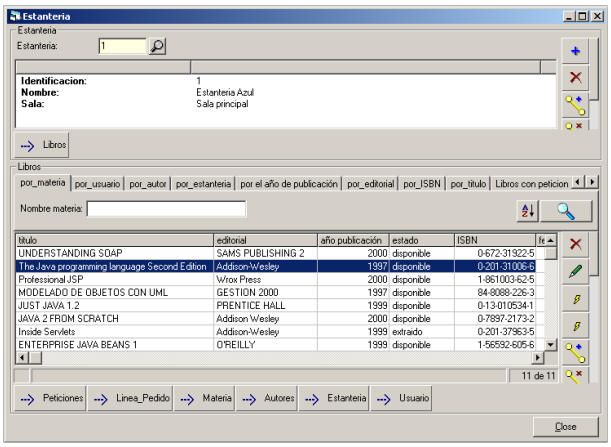


Figura 7.13: GUI de escritorio 2/3.

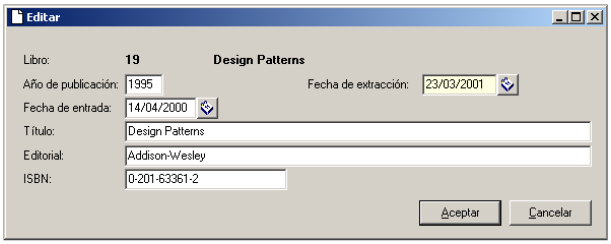


Figura 7.14: GUI de escritorio 3/3.

7.10.2. Aplicaciones Web

Como ejemplo de aplicación web se utilizará el mismo ejemplo de aplicación Las Figuras 7.15, 7.16 y 7.17 muestras el aspecto de una implementación de la interfaz de usuario totalmente generada de modo automático basada en Macromedia ColdFusion MX que puede ser visualizada en cualquier navegador compatible con HTML 3.2 y con soporte a Javascript (como por ejemplo Internet Exporer, Netscape u Opera).

La Figura 7.15 muestra de nuevo el aspecto de una ventana de autenticación. La Figura 7.16 ilustra una búsqueda de materias y el aspecto de la implementación en Web de una unidad de población. Obsérvese la implementación de los filtros como pestañas y las acciones y navegaciones como enlaces. Finalmente la Figura 7.17 muestra una página de servicio empleada por los usuario de la biblioteca para solicitar la compra de nuevos ejemplares.

Figura 7.15: GUI en Web 1/3.

7.10.3. Aplicaciones móviles para PDA

Por último, se mostrarán ejemplos de como los patrones conceptuales para interfaces de usuario son también aptos para dispositivos móviles. En particular, se ha llevado un estudio y un prototipado [Belenguer02] sobre dispositivos Pocket PC soportados por el sistema operativo WindowsCE e implementados en Visual Basic for CE y Visual C++.

El ejemplo ilustrado muestra un sistema para la gestión de notas de gasto. La Figura 7.18 muestra dos pantallas. La pantalla de la izquierda representa un ejemplo de implementación del Árbol de Jerarquía de Acciones en un Pocket PC haciendo uso del CIO TreeView. Por otra parte la pantalla de la derecha muestra un ejemplo de implementación de unidad de interacción de población.

Por último, la Figura 7.19 vuelve a ilustrar dos pantallas. La pantalla de la izquierda corresponde a una implementación de una unidad de interacción de servicio. Mientras que la pantalla de la derecha corresponde a un ejemplo de unidad de interacción de instancia.

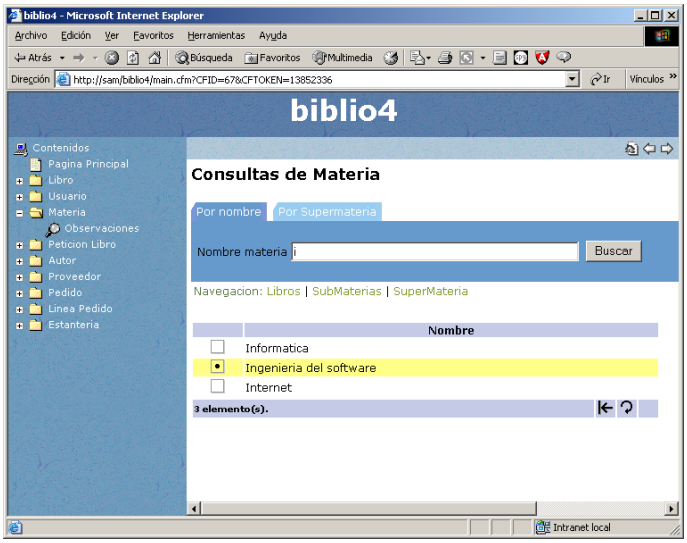


Figura 7.16: GUI en Web 2/3.

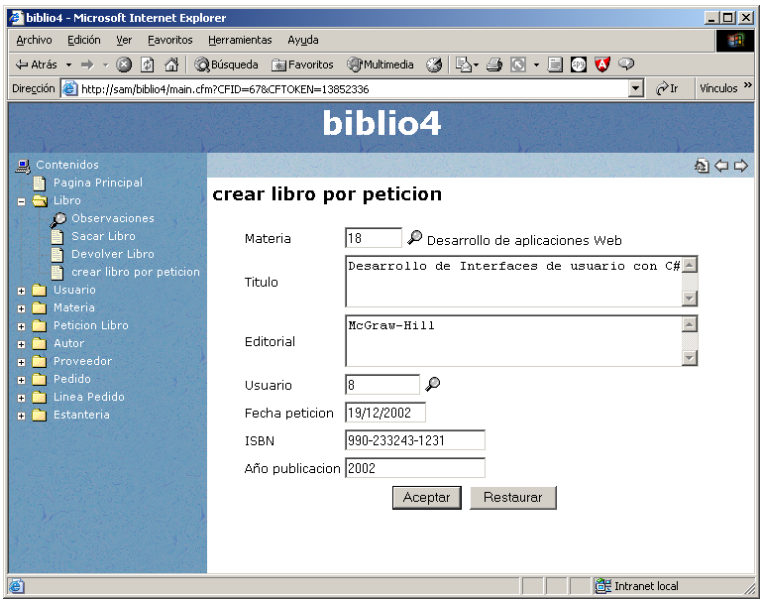


Figura 7.17: GUI en Web 3/3.

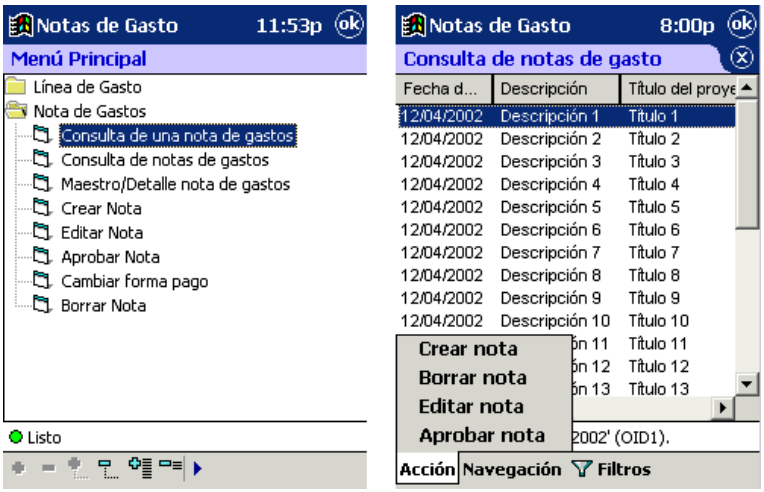


Figura 7.18: GUI en Pocket PC 1/2 [Belenguer02].

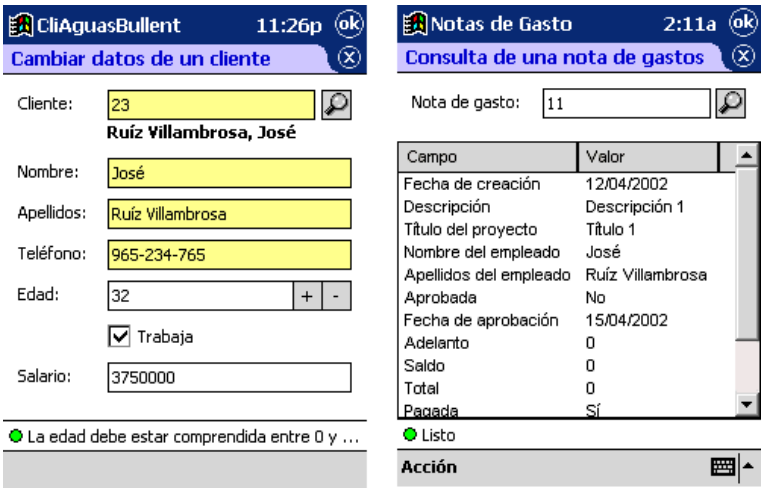


Figura 7.19: GUI en Pocket PC 2/2 [Belenguer02].

7.11. Conclusiones del capítulo

En el presente capítulo se ha comentado las ventajas y retos que supone la adopción de un proceso basado en generación de código. A través de un recorrido sobre las diversas técnicas para generación de código se han descrito las ventajas de cada una de las técnicas presentadas. Desde el punto de vista de la generación de interfaz de usuario, la generación de código debe dar cuenta de diferentes aspectos como son la funcionalidad, las relaciones de continencia de controles, la disposición de los componentes así como el cumplimiento con guías de estilo.

En este sentido, los lenguajes basados en XML se están perfilando como serios candidatos a soportar las especificaciones gracias a la versatilidad de mantenimiento, extensión y capacidades de refinamiento que proporcionan los documentos XML.

Por último, se ha presentado ejemplos de aplicaciones producidas para diversos entornos de escritorio, Web y móvil. Demostrando de este modo, la utilidad de una especificación de interfaz de usuario abstracta y la viabilidad de los generadores para diferentes plataformas tan diferentes como las ilustradas.

Capítulo 8

Impacto sobre el Proceso Software

«Somos lo que hacemos día a día, de modo que la excelencia no es un acto sino un hábito.»

— Aristóteles, filósofo griego, 384 a.C. – 322 a.C.

Este capítulo presenta cómo las ideas expuestas en la tesis han sido aplicadas a productos comerciales para el desarrollo de software en un ciclo de *ingeniería de dominio* como son: editores de modelos (herramientas CASE) o traductores, y más específicamente, cómo el uso de dichas herramientas en ambientes de *ingeniería de aplicación* han variado el ciclo habitual de desarrollo de software.

8.1. Productos software

En CARE Technologies se han desarrollado las herramientas necesarias para la producción automática de software. Esta familia de herramientas se denomina **Oliva Nova Model Execution®** (*ON Model Execution*). La familia esta constituida por siguientes elementos:

- Una herramienta CASE de modelado conceptual, **Oliva Nova Modeller®**, incluyendo diagramado gráfico y validación de modelos.
- Y un conjunto de traductores, **Oliva Nova Transformation Engines®**. Incluyendo generación de capas de persistencia (SQL92 y diversos

SGBD), de lógica de negocio (COM+ y EJB), de presentación (Windows, Java, Web), calculo de Puntos de Función sobre el modelo así como documentación y ayuda en línea para la aplicación final.

Dichas herramientas dan soporte no sólo a la parte funcional del sistema sino que también incorporan los conceptos e ideas para la especificación y generación de interfaces de usuario descritos en la presente tesis.

8.2. Método de ingeniería de dominio

Al mismo tiempo que se desarrollaban y refinaban las herramientas, se ha ido refinando un método de ingeniería de dominio soportado sobre estas herramientas para la construcción de aplicaciones.

El proceso de producción de aplicaciones (o proceso de *ingeniería de aplicación*) tiene las siguientes fases:

1. Toma de requisitos
   - Entrevistas con el cliente
   - Documentación
2. Análisis
   - Construcción de un modelo conceptual
   - Generación automática para obtener prototipos
   - Validación de los requisitos con el usuario soportado en el prototipo
3. Diseño
   - Automático
   - Optimizaciones como código manual
4. Implementación
   - Generación automática
   - Código manual
   - Integración del código generado con los cambios manuales
5. Pruebas
   - Al código manual
   - A la integración del código manual
   - A la aplicación completa

6. Instalación
7. Mantenimiento
    - Sobre modelo
    - Sobre el código manual

No es necesario finalizar cada fase antes de comenzar la siguiente. En realidad, compaginar la fase de toma de requisitos con la de análisis, permite obtener prototipos ejecutables de modo muy rápido gracias a las capacidades de generación de código que permiten validar más rápidamente los requisitos con el usuario. Una vez validados los requisitos, puede completarse la fase de análisis y proseguir con el resto de las fases.

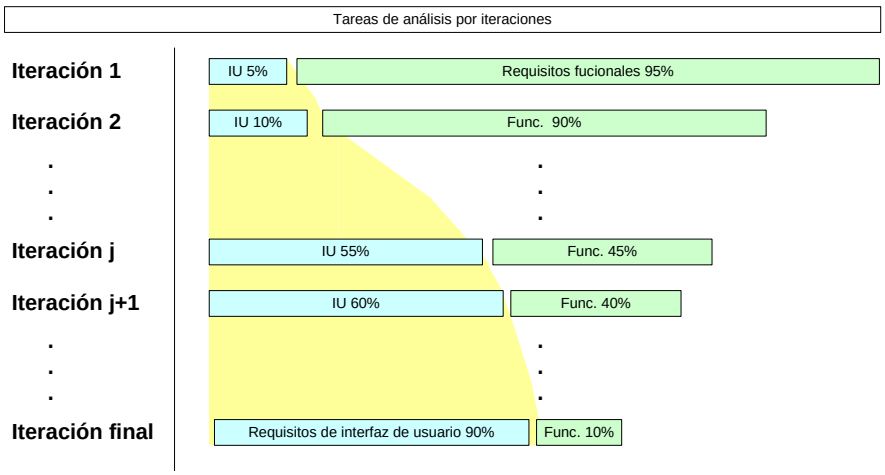


Figura 8.1: Iteraciones en el proceso de análisis.

La Figura 8.1 muestra las diversas iteraciones de toma de requistos, análisis y prototipado. En las primeras fases, el analista puede concentrarse casi exclusivamente en los requisitos fucionales. Los prototipos son obtenibles gracias al proceso de inferencia descrito en el capítulo 6. Conforme los requisitos funcionales son validados, el peso de la interfaz de usuario crece. El analista comienza a detallar los requisitos correspondientes a la interfaz de usuario. Generalmente, tras cada iteración el número de temas pendientes es menor, por tanto, el esfuerzo en las sucesivas iteraciones decrece.

8.3. Coste del software

Existen muy pocos estudios empíricos dedicados a medir del esfuerzo en interfaces de usuario en el desarrollo de aplicaciones. Una honrosa excepción

es el informe de Myers et al. [Myers92] «*Survey on User Interface Programming*» presentado en 1992 en la conferencia CHI donde se puso de manifiesto que, por término medio, alrededor del 48 % al 50 % de los esfuerzos de desarrollo están dedicados a la interfaz de usuario.

Para poder estimar costes, se hace necesario medir el tamaño de los proyectos. Una de las técnicas más empleadas para este tipo de mediciones son los Puntos de Función. Los Puntos de Función (PF) dan una idea del tamaño y complejidad del software. El IFPUG [IFPUG01] (*International Function Point User Group*), es el organismo que vela por la estandarización de los criterios de medición de los Puntos de Función.

Según Gartner Group [Gartner01], la media de PF desarrollados por día en la industria del software oscila entre 0,8 y 3,3 PF/día.[1]

Con el objeto de medir el peso en Puntos de Función de un proyecto OO-Method, se desarrolló una adaptación del proceso de medida con puntos de función para modelos orientados a objetos [Pastor01]. Dicho método constituye de por si un algoritmo preciso para realizar mediciones totalmente automáticas a partir de la especificación formal, evitando de este modo, errores del cálculo asociados a una medición manual.

Este método de medida fue aceptado como válido por la consultora Gartner Group. Así lo reafirman en la prueba comparativa (*benchmark*) [Gartner01] donde se certifico la consistencia del método automático con el método empleado por la propia consultora Gartner Group.

8.4. Estudio comparativo

En Marzo de 2001 la consultora Gartner Group realizo para CARE Technologies S.A. un estudio comparativo (o *benchmark*) para medir la productividad y calidad de las aplicaciones desarrolladas con el método de producción automática de software empleado en CARE [Gartner01].

Durante dicha comparativa se contrastaron 8 proyectos desarrollados con el proceso de producción automática de software de CARE contra un grupo de 34 proyectos de referencia (o *peer group*) similares en tamaño, características y tecnologías empleadas (aplicaciones para Unix, Visual Basic y entornos Web). Los tamaños de los ocho proyectos de CARE se muestran en la Tabla 8.1. Los datos de los proyectos del grupo de referencia no fueron revelados para preservar la privacidad de los clientes de Gartner Group. Del mismo modo, los nombres de los proyectos en CARE Technologies tampoco son revelados aquí para preservar la confidencialidad de los clientes de CARE Technologies. Sin embargo, de

[1] Puntos de Función por día.

ID	Tamaño en PF	Esfuerzo (días)
ABO01	929	33,2
ABO02	771	11,2
ABO03	1.080	36,7
ABO04	253	3,2
ABO05	3.512	61,5
ABO06	317	2,7
ABO07	775	11,4
ABO08	216	5,6

Tabla 8.1: Proyectos seleccionados, tamaños y esfuerzo.

cara a perfilar las características del grupo de comparación, podemos comentar que entre los proyectos seleccionados por CARE en la comparativa se incluye diversos sistemas como el sistema de tarificación de una compañía de suministro de aguas, el sistema de seguimiento de una prueba deportiva, sistemas de gestión de ventas, alquileres y facturación así como proyectos internos de menor tamaño.

Las principales conclusiones del informe hacen referencia a las siguientes áreas: tiempo de entrega, productividad y calidad.

8.4.1. Tiempo de entrega

La Figura 8.2 muestra un gráfico de dispersión de los proyectos respecto a duración total (días) frente a tamaño (PF). En el gráfico también se aprecia la tendencia del grupo de referencia (línea superior) y del grupo de proyectos de CARE (línea inferior). Claramente, a igualdad en tamaño, los proyectos desarrollados en CARE presentan una tendencia a ser concluidos en un tiempo considerablemente menor.

La Figura 8.3 muestra una comparativa de los ratios PF/día respecto al tiempo de entrega de la aplicación.

- Las columnas 1^a a 8^a, etiquetadas como ABO0n:AVG muestran los valores obtenidos para cada uno de los proyectos desarrollados por CARE Technologies.
- La 9^a columna, etiquetada como ABO-CARE:AVR muestra la media sobre las ocho columnas previas.
- La 10^a columna, etiquetada como CRPG:AVG muestra el valor medio obtenido sobre el grupo de control (*Reference Peer Group*) empleado por Gartner Group como medida de referencia.

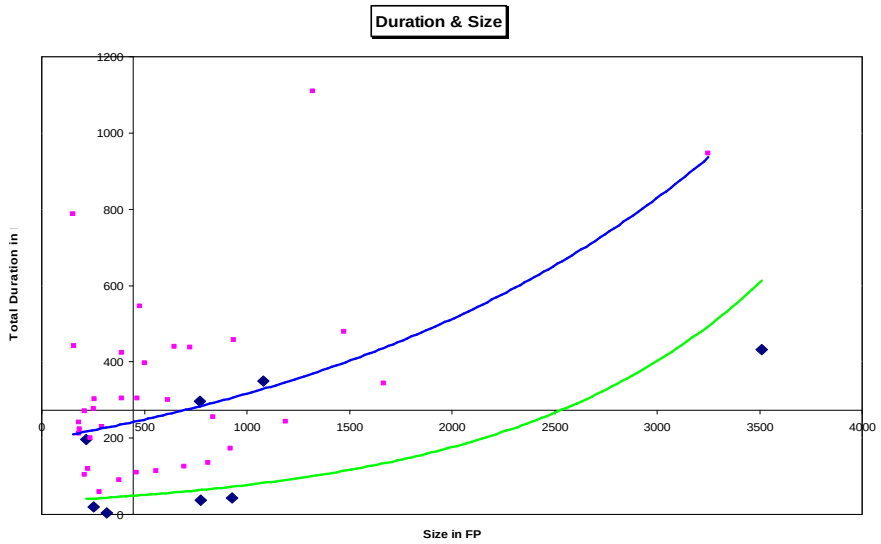


Figura 8.2: Duración total (días) respecto a tamaño (FP) [Gartner01].

- Las columnas 11^a a 13^a etiquetadas como CRPG:Qn muestran la distribución de los valores medios según el primer, segundo y tercer cuartiles, respectivamente.
- Por último, las 14^a columna etiquetada como CRPG:WAV muestra una media ponderada respecto al grupo de control.

La media para los proyectos desarrollados en CARE (columna 9 etiquetada como ABO-CARE:AVR) indica 18,67 FP/día frente a 2,37 FP/día como media para grupo de referencia (columna 10, etiquetada como CRPG:AVR) proporcionados por Gartner Group.

Resultados más destacables en términos de tiempo de entrega (según Gartner Group):

- Tiempo de entrega inferior a la media.
- Gran variación de unos proyectos a otros: La disponibilidad de los usuarios es clave para una temprana entrega.

8.4.2. Productividad

En términos de productividad, la Figura 8.4 muestra un resumen de los índices PF/día por proyectos. La media para los proyectos desarrollados en

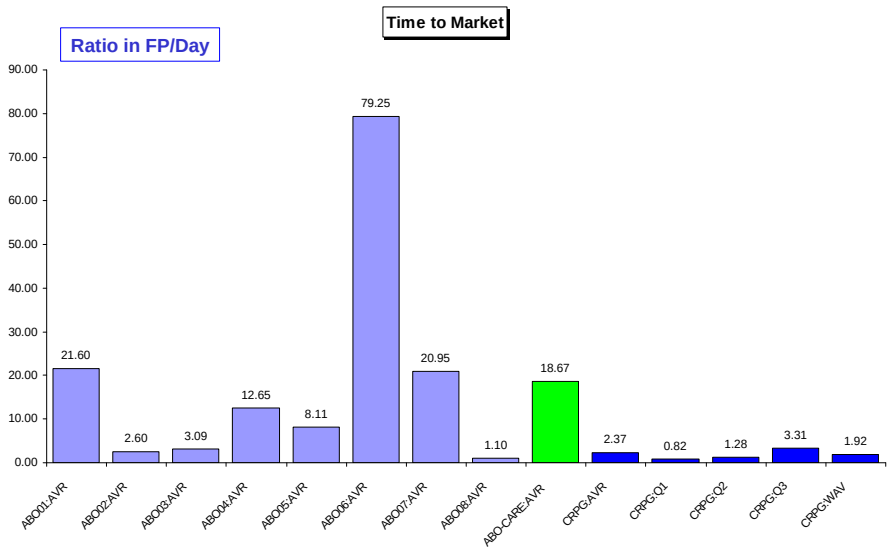


Figura 8.3: Tiempo de entrega (ratio PF/día) [Gartner01].

CARE (9^a columna) indica 65,10 FP/día frente a 5,21 FP/día como media para grupo de referencia (10^a columna) proporcionados por Gartner Group. La diferencia es más que apreciable: más de un orden magnitud respecto al grupo de control.

La diferentes tendencias en cuanto a términos de productividad se refiere, pueden observarse en la Figura 8.5. La línea superior muestra una productividad promedio de 60 FP/día ligeramente decreciente con el aumento del tamaño de los proyectos. Por el contrario el grupo de referencia muestra una productividad media de 5 PF/día ligeramente creciente con el tamaño de los proyectos.

La Figura 8.6 muestra la productividad comparada respecto al grupo de control para cada una de las fases de desarrollo. Las diferencias, en todos los casos considerablemente mejores en los proyectos desarrollados en CARE. La Figura 8.7 resume el coeficiente de mejora en la productividad en cada fase.

Los resultados más destacables en términos de productividad (según Gartner Group) son:

- Productividad superior a la media.
- Productividad global oscila entre 30 PF/día y 120 FP/día.
- El rendimiento global esta fuertemente influenciado por la cantidad de código manual a desarrollar.

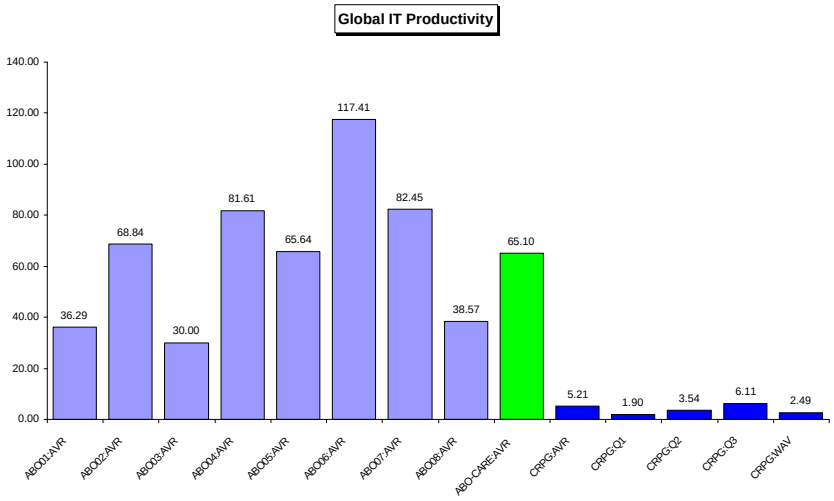


Figura 8.4: Productividad global (ratio FP/día) [Gartner01].

- Todas las fases tienen una productividad superior a la media.

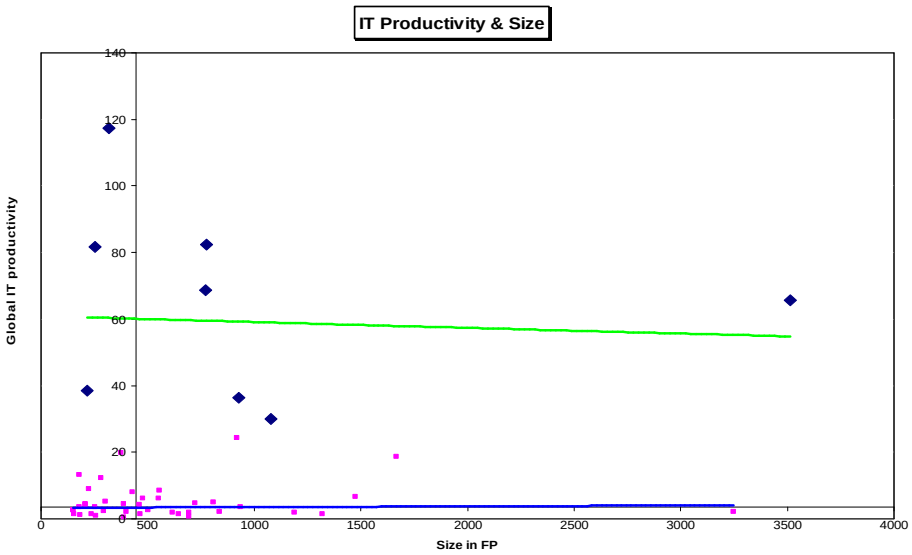


Figura 8.5: Productividad respecto al grupo de referencia [Gartner01].

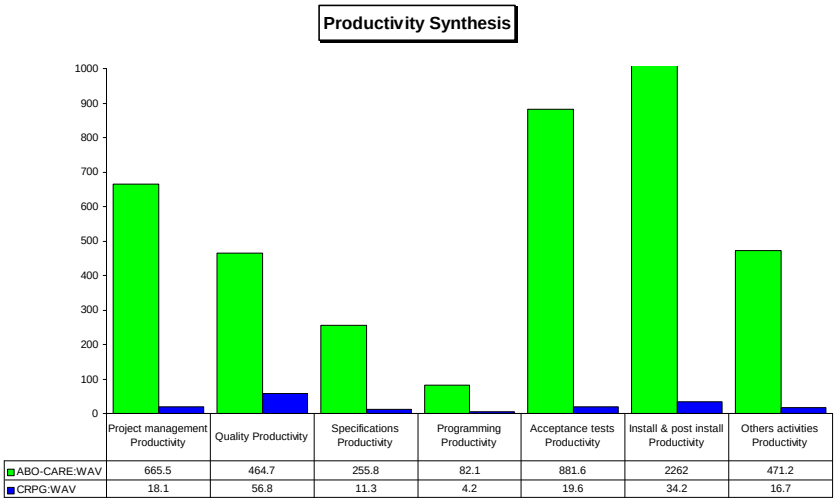


Figura 8.6: Productividad comparada por fases [Gartner01].

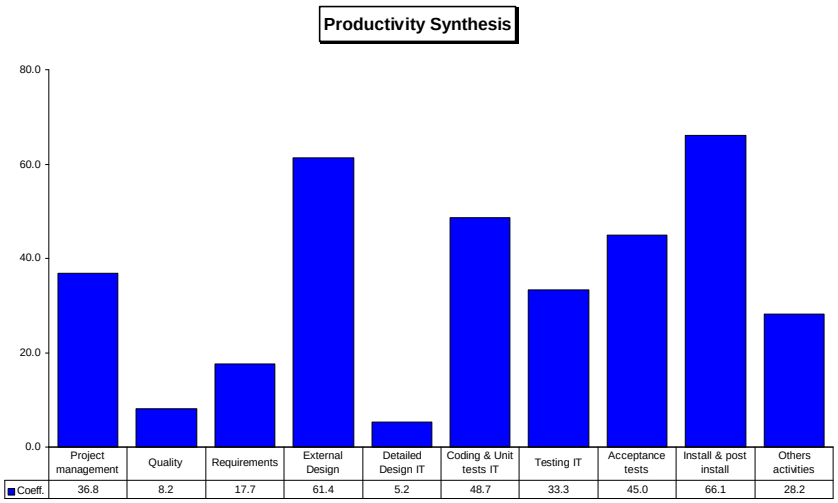


Figura 8.7: Coeficientes de mejora de la productividad respecto al grupo de referencia [Gartner01].

8.4.3. Calidad

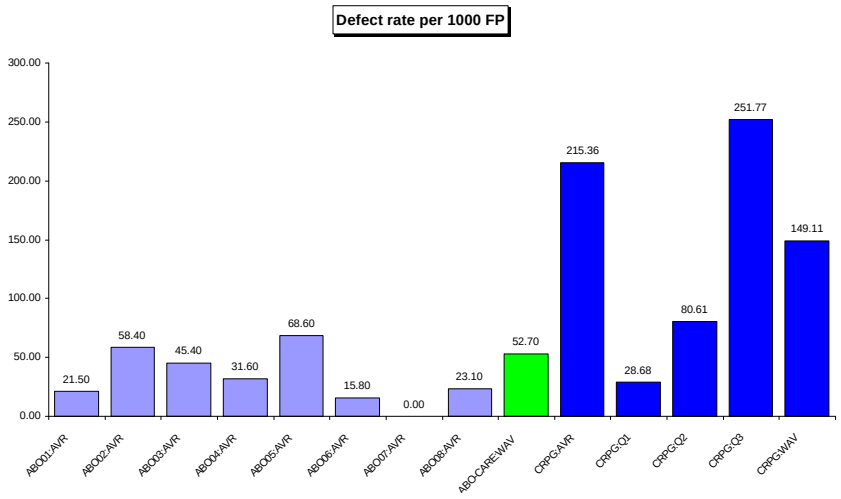


Figura 8.8: Tasa de defectos por FP [Gartner01].

Los defectos por PF constituyen buen indicador de la calidad de un producto software. En la Figura 8.8 puede apreciarse como la media de tasa de defectos de los proyectos desarrollados en CARE es de 52,70 defectos/1.000 PF frente a los 215,36 defectos/1.000 PF habituales en el grupo de referencia.

Mas aún, Gartner Group reconoce que el perfil de distribución de los errores en las distintas fases en términos de origen y detección no concuerda respecto al grupo de referencia. Este efecto puede apreciarse en la Figura 8.9 donde los errores de los proyectos desarrollados en CARE son detectados en las fases tempranas (más de la mitad en la fase de requisitos) con una fuerte tendencia hacia la disminución en las siguientes fases. Por el contrario, en el grupo de control los errores se concentran en fases posteriores. Una detección temprana de los errores abarata el coste y reduce el esfuerzo de su corrección.

Los resultados más destacables en términos de calidad (según Gartner Group) son:

- La robustez de la fase de modelado previene la aparición de defectos críticos.
- Los errores provienen en su mayoría del código manual introducido. El código generado origina pocos defectos.
- Los defectos son detectados por lo general en fases más tempranas que respecto a los proyectos del grupo de control. Esto facilita su corrección

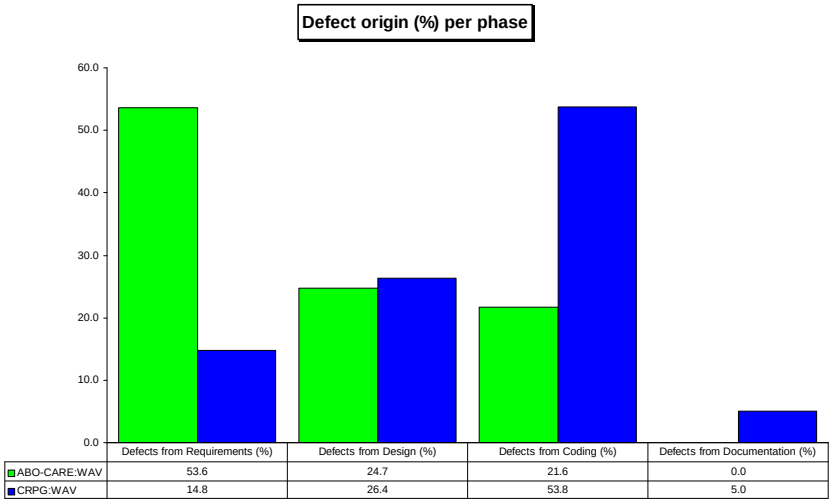


Figura 8.9: Origen de los defectos [Gartner01].

antes y a un coste inferior.

8.4.4. Impacto sobre el ciclo de desarrollo

Desarrollo tradicional		*Desarrollo basada en generación*	
Fase	**%**	**Fase**	**%**
Requisitos	10 %	Requisitos	25 %
Análisis	20 %	Análisis	40 %
Diseño	10 %	Diseño	4 %
Codificación	40 %	Codificación	6 %
Pruebas	15 %	Pruebas	20 %
Implantación	5 %	Implantación	5 %

Tabla 8.2: Porcentajes de esfuerzo dedicados a cada fase.

El uso un método soportado mediante generación de código altera radicalmente el ciclo de vida del desarrollo. La Tabla 8.2 y la Figura 8.10 muestran la diferente distribución de esfuerzo con un método tradicional donde el código es desarrollado a mano, frente a un método soportado en generación de código.

Mientras que el desarrollo tradicional (*véase Tabla 8.2*) las fases de codificación es la que más esfuerzo requiere, en el desarrollo basado en generación de código las tareas a las que se dedican más esfuerzo son el Análisis, Requisitos y Pruebas.

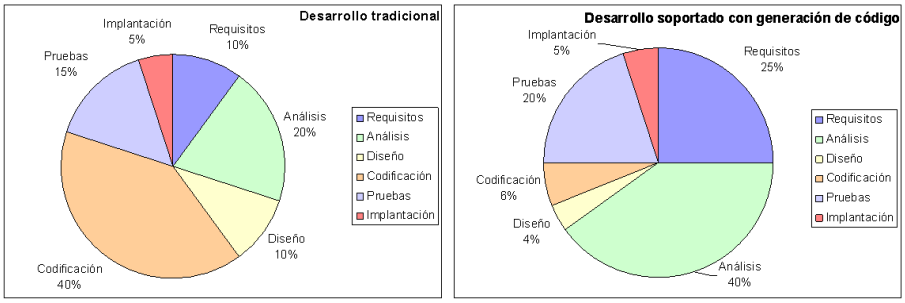


Figura 8.10: Esfuerzos dedicados a las diferentes fases.

El esfuerzo dedicado a diseño y codificación ($50\,\% = 10 + 40$) caen drásticamente al ($10\,\% = 4 + 6$) gracias al que la mayor parte del código generado automáticamente forma parte de la aplicación final tal cual, sin cambios.

El ahorro obtenido en estas dos fases, permite dedicar (porcentualmente) más tiempo a la toma de requisitos, el análisis (que forzosamente ha de ser más riguroso si cabe) y a la fase de pruebas donde los prototipos ayudan a validar la especificación desde el primer momento.

8.5. Impacto sobre la Interfaz de Usuario

El informe de la consultora Gartner [Gartner01] sobre el proceso de desarrollo de aplicaciones en CARE pone de manifiesto una mejora sustancial en todo el proceso de desarrollo de software.

Sin embargo, no es fácil desprender de dicho estudio la mejora que es consecuencia directa de la introducción del Modelado de Interfaz de Usuario basada en patrones, la especificación y generación de la interfaz de usuario sin considerar todo el proceso de desarrollo como un todo.

Según el estudio de Myers [Myers92] el desarrollo de la interfaz de usuario supone típicamente alrededor de 48-50 % de los esfuerzos de desarrollo. En nuestro proceso de desarrollo hemos comprobado que:

- En la fase de especificación y análisis, el trabajo destinado a la interfaz de usuario supone aproximadamente el 40 % del esfuerzo. Esta disminución respecto a las cifras de Myers es achacable a que la expresividad del modelo, proceso de inferencia y valores por defecto facilitan la especificación simplificándola en gran medida.
- En la fase de diseño e implementación el 90 % de los escenarios de la interfaz de usuario son usados directamente sin cambios. El 10 % de los

casos restantes necesitan ser construir o adaptar escenarios particulares para la interfaz de usuario que no pueden ser generados (quedan fuera del alcance de los patrones disponibles). Estos escenarios son construidos a mano por los diseñadores a partir del código generado e integrado con la aplicación generada. En esta fase de diseño manual el impacto de la interfaz de usuario puede suponer del orden de 50-60 %. Dicho porcentaje aunque mayor que el pronosticado por Myers, viene justificado porque gran parte de la aplicación (cerca del 90 %) ya ha sido generada y como trabajo de codificación solo resta ajustar el 10 % del código restante.

La especificación de un modelo conceptual transcurre en una espiral de ciclos de construcción y evaluación destinados a verificar que los requisitos están correctamente expresados. Estos ciclos concluyen cuando se da la especificación por validada y completada.

En los primeros ciclos, la especificación se centra en recoger y validar los requisitos funcionales. En estos ciclos, el esfuerzo de especificación de interfaz de usuario debe ser el mínimo imprescindible para alcanzar los prototipos. Aquí hemos apreciado que los procesos de inferencia reducen la especificación de UI en un 85 % respecto a una especificación sin inferencia.

Conforme se va alcanzando la especificación final, el analista completa la especificación con los requisitos del usuario. Dicho esfuerzo de especificación sigue siendo menor (respecto a especificación sin inferencia) debido al uso de valores por defecto y el principio de beneficiar el caso frecuente. Aquí hemos apreciado una mejora del 50 % respecto a una especificación sin inferencia.

8.6. Conclusiones del capítulo

La teoría y el modelo presentado en esta tesis han sido corroborados por medio de la implementación de las herramientas de soporte y del uso de éstas en un amplio abanico de proyectos reales desarrollados para la empresa privada.

Gartner Group [Gartner01] ha certificado la mejora del proceso de producción de aplicaciones en este mismo entorno industrial con unos resultados espectaculares. Consecuentemente, la distribución de esfuerzos durante el ciclo de vida del desarrollo del software se ha visto también alterado.

Aunque el Paradigma de Programación Automática [Balzer83] sea considerado como una utopía y tal vez no pueda ser nunca alcanzado al cien por cien (del mismo modo que no puede ser alcanzada una asíntota en una función matemática), hemos comprobado empíricamente como la generación automática a partir de modelos basados en patrones ha permitido solventar más

del 90 % de los escenarios de aplicaciones complejas. Y todo ello ha permitido, al mismo tiempo, un aumento de la productividad, mejora en los tiempos de entrega y calidad de la aplicación final con unos resultados sobresalientes que avalan la utilidad industrial del método propuesto.

Desde el punto de vista industrial, en los próximos años se espera una tendencia creciente hacia los productos que han venido a denominarse como encuadrados en *«Model Execution»* [Weber01].[2] El presente trabajo puede ser encuadrado como una muestra de esta nueva ola de herramientas para la producción automática de software.

[2] Ejecución de Modelos.

Capítulo 9

Conclusiones

«La verdadera investigación consiste en buscar a oscuras el interruptor de la luz. Cuando la luz se enciende, todo el mundo lo ve muy claro.»

— Anónimo

9.1. Conclusiones

Como conclusión final, este capítulo presenta la relación de aportaciones, publicaciones científicas, patentes, aplicaciones y trabajos derivados de la presente tesis. Así mismo, una serie de trabajos futuros son comentados como líneas abiertas de trabajo en el área.

9.2. Resumen de aportaciones

Las contribuciones más relevantes de ésta tesis son las siguientes:

1. Se han explorado **los patrones de interfaz de usuario** y en particular su aplicación a **la especificación de interfaces de usuario** a nivel conceptual.

2. Se ha definido el concepto de **Patron Conceptual de Interfaz de Usuario** [Molina02b, Molina02c] como concepto útil para la especificación de interfaces de usuario.

3. Se ha propuesto un **Lenguaje de Patrones** conceptuales de interfaz de usuario para la especificación precisa de interfaces de usuario.

4. Se ha definido el concepto de **Unidad de Interacción** como extensión a la Unidad de Presentación [Bodart95c].

5. El lenguaje de patrones ha sido empleado como **conceptos de especificación** en una herramienta de modelado conceptual.

6. Se han construido especificaciones de sistemas reales para **validar la adecuación del modelo** a los requisitos impuestos por analistas y usuarios. El lenguaje de patrones ha sido refinado teniendo en cuenta la retroalimentación recibida de este proceso.

7. Se ha definido una **notación gráfica** sencilla para permitir a analistas y usuarios comprender las especificaciones y validar conjuntamente los requisitos.

8. El lenguaje de especificación de interfaz de usuario basado en patrones ha sido definido en términos de meta-modelado. Y es **aplicable como extensión a cualquier modelo de análisis orientado a objetos**.

9. Las especificaciones construidas han demostrado ser **independientes de plataforma** y aptas para la generación automática a diferentes entornos finales.

10. Se han **construido traductores** y generadores de código a diversas plataformas que producen una interfaz de usuario adaptada a las características de cada plataforma y arquitectura.

11. Se han **desarrollado proyectos reales** usando las especificaciones según el modelo propuesto y las correspondientes herramientas de modelado y de traducción.

12. El proceso de inferencia propuesto da soporte a procesos de **prototipación rápida** que aceleran notablemente los ciclos de toma de requisitos, análisis, codificación de prototipo, pruebas y evaluación. Y que por tanto, permite validar no solo los requisitos de interfaz de usuario sino también los requisitos funcionales gracias al prototipado rápido del sistema.

13. El proceso de desarrollo de interfaces de usuario propuesto basado en especificación y generación de código ha demostrado **ser escalable** y capaz de **mejorar notablemente la productividad, calidad y tiempo de entrega** de los proyectos desarrollados.

9.2.1. Publicaciones relacionadas

Esta tesis está fundamentada sobre una serie de publicaciones de investigación. A raíz de la asistencia a conferencias, congresos, jornadas y toma de

contactos con otros investigadores en la materia, se ha obtenido retroalimentación de la comunidad científica que ha sido determinante para orientar y mejorar la línea de trabajo en curso. Los trabajos involucrados son los siguientes:

1. [Molina98] Pedro J. Molina. **«Especificación de Interfaz de Usuario En OO-Method»**, Proyecto Final de Carrera, Facultad de Informática, Universidad Politécnica de Valencia, Valencia, España, Septiembre, 1998.

2. [Pastor00b] Oscar Pastor, Pedro J. Molina y Alberto Aparicio. **«Specifying Interface Properties in Object Oriented Conceptual Models»** En *Working Conference on Advanced Visual Interfaces, AVI 2000*, Palermo, Italia, ACM Press, páginas 302-304, ISBN 1-58113-252-2, Mayo, 2000.

3. [Insfrán01] Emilio Insfrán, Pedro J. Molina, Sofía Martí y Vicente Pelechano. **«Ingeniería de Requisitos aplicada al modelado conceptual de interfaz de usuario»**, En *Actas del IV Workshop Iberoamericano de Ingeniería de Ambientes de Software, IDEAS'2001*, Santo Domingo, Heredia, Costa Rica, CIT, páginas 181-192, Abril, 2001.

4. [Molina01] Pedro J. Molina, Oscar Pastor, Sofía Martí, Juan J. Fons y Emilio Insfrán. **«Specifying Conceptual Interface Patterns in an Object-Oriented Method with Code Generation»**, En *Proceedings of User Interfaces for Data Intensive Systems, UIDIS '2001*, IEEE Computer Society, páginas 72-79, ISBN 0-7695-0834-0, Zurich, Suiza, Mayo, 2001.

5. [Molina02a] Pedro J. Molina, Sofía Martí y Oscar Pastor. **«Prototipado rápido de interfaces de usuario»**, En *Actas del V Workshop Iberoamericano de Ingeniería de Ambientes Software, IDEAS'2002*, Miguel Katrib et al. (editores) La Habana, Cuba, páginas 78-90, ISBN 959-7160-14-5, Abril, 2002.

6. [Molina02e] Pedro J. Molina, Ismael Torres y Oscar Pastor. **«Patrones de interfaz de usuario para la navegación orientada a objetos»**, En *Actas del III Congreso Interacción Persona Ordenador, IPO'2002*, Ignacio Aedo, Paloma Díaz y Camino Fernández (editores), AIPO, páginas 27-36, ISBN 84-607-4501-5, Leganes, España, Mayo, 2002.

7. [Molina02d] Pedro J. Molina, Santiago Meliá y Oscar Pastor. **«Just-UI: A User Interface Specification Model»** En *Computer-Aided Design of User Interfaces III, Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002*, Ch. Kolski y J. Vanderdonckt (editores), Kluwer Academics Publisher, Dordrecht, páginas 63-74, ISBN 1-4020-0643-8 Valenciennes, Francia, Mayo, 2002.

8. [Molina02b] Pedro J. Molina, Santiago Meliá y Oscar Pastor. **«User Interface Conceptual Patterns»**, En *Proceedings of the 4th International Workshop on Design Specification & Verification of Information Systems DSV-IS'2002*,Peter Forbrig, Quentin Limbourg, Bodo Urban y Jean Vanderdonckt (editores), páginas 201-214, Rostock, Alemania, Junio, 2002.

9. [Molina02c] Pedro J. Molina, Santiago Meliá y Oscar Pastor. **«User Interface Conceptual Patterns»**, En *Lecture Notes of Computer Science* Peter Forbrig, Quentin Limbourg, Bodo Urban y Jean Vanderdonckt (editores), volumen 2545, ISBN 3-540-00266-9, Springer Verlag, Berlin, Alemania, Diciembre, 2002.

10. [Molina03a] Pedro J. Molina, Santiago Meliá y Oscar Pastor. **«The Just-UI approach: Conceptual modelling of device independent interfaces»** *Journal of Human-Computer Interaction (RIHM)* Special Issue on Computer-Aided Design of User, Christophe Kolski y Jean Vanderdonckt (editores), Interfaces vol. 3 (1), 2003 (En prensa).

11. [Molina03c] Pedro J. Molina, Ismael Torres y Oscar Pastor. **«Patrones de interfaz de usuario para la navegación orientada a objetos»**, *Novatica*, ATI, Madrid, España, 2003 (En prensa).

12. [Molina03d] Pedro J. Molina, Ismael Torres y Oscar Pastor. **«User Interface Patterns for Object-Oriented Navigation»**, *Upgrade*, Disponible en: `http://www.upgrade-cepis.org`, 2003, (En prensa).

El siguiente libro, aunque no enfocado a la investigación, también divulga el uso interfaces de usuario para entornos Web:

- [Sánchez01a] José Ignacio Sánchez, Gustavo Santos y Pedro J. Molina. **«HTML 4. Iniciación y Referencia»**, McGraw Hill Iberoamericana, (2ª edición), ISBN 84-481-3168-1, Madrid, España, Septiembre, 2001.

9.2.2. Patentes industriales

Porciones de este trabajo están protegidas por patentes industriales presentadas en los Estados Unidos de America propiedad de SOSY Inc.:

1. [Iborra00] José Iborra y Óscar Pastor. **«Automatic Software Production System»** Patente USPTO 09/543.085, SOSY Inc. EE.UU, Abril, 2000.

2. [Iborra02a] José Iborra y Óscar Pastor. **«Automatic Software Production System»** Patente USPTO 20020062475, SOSY Inc. EE.UU, Mayo, 2002.

3. [Iborra02b] José Iborra y Óscar Pastor. **«Automatic Software Production System»** Patente USPTO 20020100014, SOSY Inc. EE.UU, Julio, 2002.

4. [Molina03b] Pedro J. Molina, Óscar Pastor, Juan C. Molina y José M. Barberá. **«Method and Apparatus for Automatic Generation of Information System User Interfaces»** Patente USPTO 10/356.250, SOSY Inc. EE.UU, Enero, 2003.

9.2.3. Productos software comerciales

Diversas herramientas y productos comerciales propiedad de CARE Technologies S.A. incorporan ideas presentadas en esta tesis:

1. **Herramienta de modelado Oliva Nova Modeler®.** Implementa una herramienta CASE con soporte al Modelo de Presentación. Dicho modelo permite la especificación completa de interfaces de usuario en base a los patrones presentados en esta tesis.

2. **ON Transformation Engine TCVB.** Traductor Cliente para Visual Basic v. 3.0. que genera una interfaz de usuario completamente funcional sobre plataforma Windows.

3. **ON Transformation Engine TCJava.** Traductor Cliente para Java/Swing que genera una interfaz de usuario completamente funcional sobre plataforma Java usando la librería gráfica Swing.

4. **ON Transformation Engine TCWebJSP.** Traductor Cliente Web para JSP que genera una interfaz de usuario completamente funcional para entornos web implementada bajo tecnología Java Server Pages.

5. **ON Transformation Engine TCWebCF.** Traductor Cliente Web para Cold Fusion que genera una interfaz de usuario completamente funcional para entornos web implementada bajo tecnología ColdFusion Server 4.5.

9.3. Trabajos futuros

Como trabajos futuros se presentan los siguientes campos de actuación:

1. Estudio a fondo del problema del *Round-Trip* para abordar soluciones semi-guiadas soportadas por sistemas expertos que soporten la evolución de aplicaciones conforme cambian los requisitos.

2. Estudio de modelos de diseño lógico (independientes de plataforma) y físico (dependientes de plataforma) como refinamientos al modelo conceptual, en los cuales podrían aplicarse la gran cantidad de patrones de diseño de interfaz de usuario existentes [Welie00a, Welie00b, Welie00c, Trætteberg00] y sacar provecho de las características especificas del dispositivo considerado. Esto es especialmente importante para dispositivos móviles donde los recursos son limitados [Belenguer02]. Estos modelos de diseño permitirían salvar mejor el *gap* entre el análisis y la implementación introduciendo una fase de diseño donde los diseñadores podrían afinar multitud de aspectos de la generación automática, incluyendo la selección de patrones de diseño.

3. Mejora de la parametrización y la facilidad de mantenimiento de los traductores para permitir al diseñador tener más control sobre la generación automática sin perder la corrección y completitud de un generador cerrado.

4. Estudio de vias de extensibilidad al lenguaje de patrones propuesto. Tanto en la vertiente de especificación, como en la vertiente de generación automática.

5. Identificación y adicción al lenguaje de patrones de nuevos patrones útiles para la especificación de interfaces de usuario abstractas.

6. Estudio de técnicas y procedimientos para facilitar la separación de responsabilidades (*separate of concerns*) entre ingenieros y programadores frente a diseñadores y artistas gráficos para permitir un trabajo realmente colaborativo evitando colisiones entre funcionalidad (responsabilidad de los primeros) y belleza y harmonia estética (responsabilidad de los segundos).

El objetivo finalmente perseguido no ha sido otro sino el de intentar hacer valer el lema de la Universidad Politécnica de Valencia:

«EX TECHNICA PROGRESSIO».[1]

[1] Traducción del latín: *«El progreso proviene de la técnica.»*

Apéndice A

Extensión de *OASIS* 3.0

A.1. Sintaxis propuesta para *OASIS* 3.0

A continuación se describirán las extensiones propuestas para enriquecer el lenguaje *OASIS* versión 3.0 [Letelier98] y permitir la captura de la información descrita en el capítulo previo.

A.1.1. Notación empleada

Las extensiones propuestas son descritas usando la siguiente notación BNF:

- **Símbolo terminal** o si se trata de caracteres, entre comillas `' '`
- `<Símbolo no-terminal>`
- Símbolos opcionales, aparecen entre corchetes `[ ]`
- Símbolos alternativos, aparecen separados por una barra vertical `|`

A.1.2. Extensión sintáctica

En este apartado se va proporcionar una descripción de la sintaxis para expresar cada concepto o elemento del Modelo de Presentación en el lenguaje *OASIS* .

Patrón de Introducción

El Patrón de Introducción contiene una serie de propiedades que son completadas a modo de plantilla. Cada patrón dispone de un nombre único en el

modelo, lo que permite referenciar el patrón desde otros lugares de la especificación.

bloque_intropat	::=	**introductionpattern** <name> **alias** <alias> **editmask** <editmask> **defaultvalue** <defaultvalue> **lowerlimit** <lowerlimit> **upperlimit** <upperlimit> **maxchars** <maxchars> **allownull** {**True** | **False**} **errormsg** <errorMsg> **helpmsg** <helpmsg> **end_introductionpattern**

Patrón de Selección Definida

El Patrón de Selección Definida posee una estructuración similar al Patrón de Introducción con la diferencia de que ha de proporcionarse una enumeración de valores (clave, etiqueta) para describir los valores válidos.

bloque_selecciondefinida	::=	**definedselectionpattern** <name> **alias** <alias> **defaultvalue** <defaultValue> **minselectable** <nat> **maxselectable** <nat> **errormsg** <errorMsg> **helpmsg** <helpMsg> { <def_item_sel_def>} **end_definedselectionpattern**
def_item_sel_def	::=	<code>, <alias>;

Información Complementaria

El Patrón de Información Complementaria queda especificado indicando una referencia a un Conjunto de Visualización.

def_oidfeedback	::=	**oidfeedback** <class_name>:<displayset_name>

Dependencia

Las formulas de dependencia están basadas en reglas ECA (Evento, Condición, Acciones). En lógica dinámica podríamos expresar del siguiente modo una regla ECA:

$$< event > \wedge < condition > \quad [\neg actions] \quad false \tag{A.1}$$

def_dependencia	::=	**dependency** { <event>**:** {<condition>[¬ actions] **false**} } **end_dependency**

Agrupación de Argumentos

La Agrupación de Argumentos queda especificada por medio de una estructuración arborescente de agrupadores y argumentos.

def_arg_agrup	::=	**groupargnode** **alias** [**argument:** <argname> | **childs:** {<groupargnode>}] **end_groupargnode**

Filtro

Los filtros quedan definidos por medio de un nombre, una lista de variables de filtro y la fórmula de filtro propiamente dicha.

def_filtro	::=	**filter** <name> **alias** <alias> **helpmsg** <helpMsg> **filter_variables** {<var>**:** <Domain>**;**} **end_filter_variables** **formula** <formula> **end_filter**

Criterio de Ordenación

El Criterio de Ordenación es una secuencia ordenada de expresiones de ordenación donde para cada uno se expresa un sentido de ordenación.

def_criterio_ordenacion	::=	**ordercriterium** <name> **alias** <alias> {**orderitem** <rolepath>[**ASC** | **DES**]} **end_ordercriterium**

Conjunto de Visualización

Los Conjuntos de Visualización son una lista ordenada de expresiones de atributo uni-valuadas alcanzables desde la clase donde se define el conjunto de visualización.

def_cjto_visualización	::=	**displayset** <name> **alias** <alias> {**displayitem** <rolepath><alias>} **end_displayset**

Acciones

Las Acciones son una lista ordenada de Unidades de Interacción que pueden ser alcanzadas desde una Unidad de Interacción origen.

def_acciones	:==	**actions** <name> **alias** <alias> {**actionitem** <targetUI><alias>} **end_actions**

Navegación

El Patrón de Navegación es una lista ordenada de tuplas (expresión de rol, Unidad de Interacción, alias) que pueden ser alcanzadas desde una Unidad de Interacción origen.

def_navegación	::=	**navigation** <name> **alias** <alias> {**navigationitem** <formula><targetUI><alias>} **end_navigation**

Unidad de Interacción de Servicio

Una UI de Servicio puede tener aplicada (opcionalmente) un Patrón de Agrupación de Argumentos.

def_ui_servicio	::=	**iuservice** <name> **alias** <alias> [<def_arg_agrup>] **end_iuservice**

Unidad de Interacción de Instancia

Cada UI de Instancia contiene una referencia a un Conjunto de Visualización y opcionalmente, referencias a un Patrón de Navegación y Acciones.

def_ui_instancia	::=	**iuinstance** <name> **alias** <alias> **displayset** <class>**:**<displayset> [**actions** <class>**:**<actions>] [**navigation** <class>**:**<navigation>] **end_iuinstance**

Unidad de Interacción de Población

Del mismo modo que el patrón previo, la UI de Población queda definida mediante referencias a los elementos usados dentro del patrón.

def_ui_población	::=	**iupopulation** <name> **alias** <alias> [{**filter** <class>**:**<filter>}] [{**ordercriterium** <class>**:**<ordercriterium>}] {**displayset** <class>**:**<displayset>} [**actions** <class>**:**<actions>] [**navigation** <class>**:**<navigation>] **end_iupopulation**

Unidad de Interacción de Maestro/Detalle

Las UI de Maestro/Detalle quedan definidas mediante referencias a la UI maestro y tuplas detalle de la forma: (UI de Detalle, fórmula de navegación, alias del detalle).

def_ui_maestro_detalle	::=	**iumasterdetail** <name> **alias** <alias> **master** <class>**:**<iumaster> {**detail** <class>**:**<uidetail> **formula** <formula>**alias** <alias>} **end_iumasterdetail**

Vistas

Una vista es una lista de interfaces que permiten definir un subsistema desde el punto de vista de la interfaz de usuario. Un Árbol de Jerarquía de Acciones puede ser definido por Vista.

def_vista	::=	**view** <name> **alias** <alias> **interfaces:** {<interface_name>} **hat:** <hatname> **end_view**

Árbol de Jerarquía de Acciones

Cada AJA queda definido como un árbol de agrupadores donde las nodos hojas del árbol apuntan a Unidades de Interacción.

def_aja	::=	**hat** <name> **alias** <alias> <def_hatnode> **end_hat**
def_hatnode	::=	**hatnode** **alias** <alias> {**targetiu:** <iuname> | **childs:** { <def_hatnode>} } **end_hatnode**

Apéndice B

Representación en XML

B.1. Representación del modelo Just-UI en XML

En esta sección se proporciona un DTD (Documento de Declaración de Tipos)[1]que permite validar documentos XML correspondientes a especificaciones Just-UI.

Intencionadamente, se ha pretendido realizar un meta-modelado mínimo imprescindible del núcleo de común de los conceptos clave en las metodologías orientadas a objetos como son: Clase, Atributo, Servicio (método) y argumento (del servicio o método). De este modo, el modelo para la especificación de interfaz de usuario propuesto es aplicable sin problemas no solo a especificaciones OO-Method clásicas, sino que también es adaptable con poco esfuerzo a otros modelos orientados a objetos como puedan ser UML, OPEN u OMT, por citar algunos.

La especificación comienza con el elemento raíz JustUISpec. La especificación contiene clases, patrones de Introducción y Selección Definida (ambos definidos a nivel de modelo para su porterior reuso) y vistas (*véase Figura B.1*).

```
<?xml version="1.0" encoding=\"UTF-8"?>
<!-- Just-UI DTD Meta-model v. 1.0   05.12.2002 ==== -->
<!-- (c) Pedro J. Molina Moreno, 2002.  ============ -->
<!-- ============================================== -->

<!-- Minimal OO Core-Kernel: Class, Attribute, ===== -->
<!-- Service (Method), Argument ==================== -->
<!ELEMENT JustUISpec (Class*,Introduction*,DefinedSelection*, View*)>
<!ATTLIST JustUISpec
       Name    CDATA #REQUIRED
```

[1]DTD. *Document Type Definition.* Conforme a la especificación XML del W3C [W3C00].

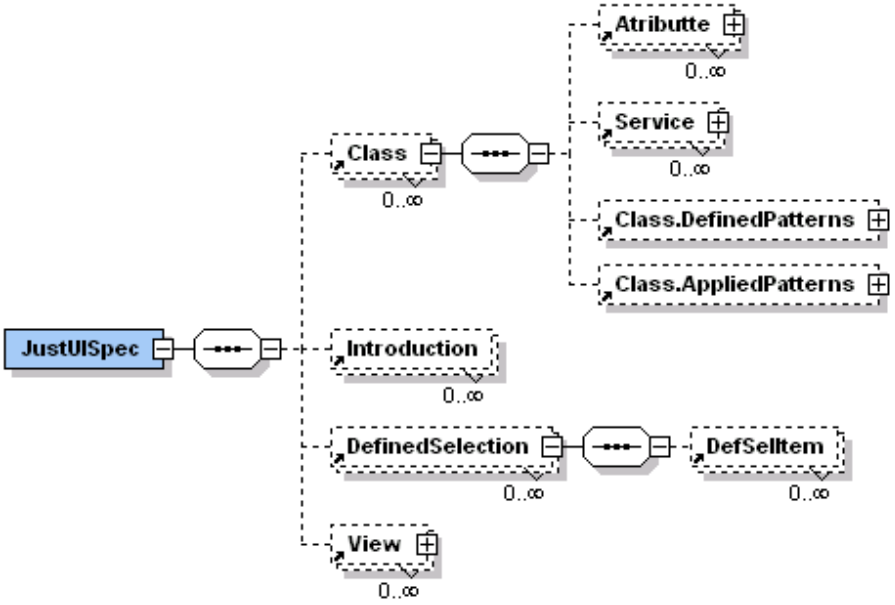


Figura B.1: Estructura de los elementos XML de una especificación Just-UI.

```
        Version CDATA #REQUIRED>
```

El elemento Class contiene una lista de atributos, y una lista de servicios (o métodos) (*véase Figura B.2*). Adicionalmente, puede tener también patrones definidos y patrones aplicados. Como propiedades relevantes a efectos del modelo de especificación de interfaz de usuario, consideraremos solo el nombre y el alias.

```
<!ELEMENT Class (Atributte*,Service*, Class.DefinedPatterns?,
                 Class.AppliedPatterns?)>
<!ATRIB Class
        Name    CDATA #REQUIRED
        Alias   CDATA #IMPLIED>
```

El elemento Atributte representa a los atributos de una clase. En nuestro modelo de interfaz de usuario las características relevantes son el dominio (Domain), el alias (Alias) y los patrones aplicados (Atributte.AppliedPatterns).

```
<!ELEMENT Atributte (Atributte.AppliedPatterns?)>
<!ATRIB Atributte
        Name    CDATA #REQUIRED
        Domain  CDATA #REQUIRED
        Alias   CDATA #IMPLIED>
```

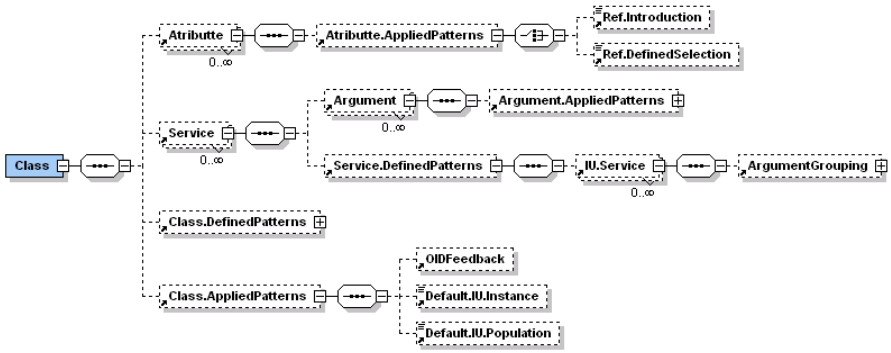


Figura B.2: Estructura del elemento XML `<Class>`.

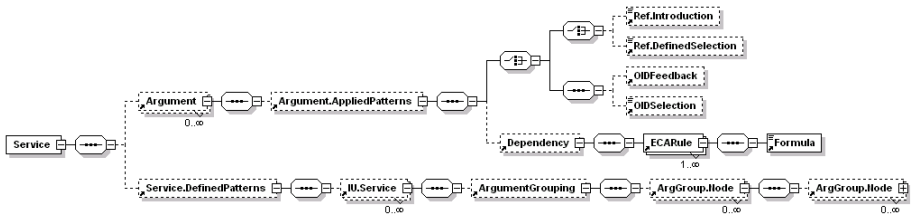


Figura B.3: Estructura del elemento XML `<Service>`.

El servicio (Service) tiene una lista de argumentos y, opcionalmente, patrones aplicados (*véase Figura B.3*) y como propiedades relevantes: nombre y alias.

```
<!ELEMENT Service (Argument*, Service.DefinedPatterns?)>
<!ATRIB Service
       Name    CDATA #REQUIRED
       Alias   CDATA #IMPLIED>
```

El argumento (Argument) puede tener también patrones aplicados. Como propiedades relevantes se consideran el nombre, el tipo (o dominio) y el alias.

```
<!ELEMENT Argument (Argument.AppliedPatterns?)>
<!ATRIB Argument
       Name    CDATA #REQUIRED
       Domain  CDATA #REQUIRED
       Alias   CDATA #IMPLIED>
```

La fórmula es un elemento auxiliar que permite almacenar fórmulas bien formadas en el lenguaje *OASIS* .

```
<!-- Auxiliar Formula Element ================== -->
<!ELEMENT Formula (#PCDATA)>
<!-- ========================================== -->
```

El Patrón de Introducción presenta una lista de propiedades en su definición, la mayoría de ellas, son opcionales.

```
<!-- L3.1 Introduction Pattern ================= -->
<!ELEMENT Introduction EMPTY>
<!ATTLIST Introduction
       Name            CDATA #REQUIRED
       EditMask        CDATA #IMPLIED
       DefaultValue    CDATA #IMPLIED
       LowerBound      CDATA #IMPLIED
       UpperBound      CDATA #IMPLIED
       MaxChars        CDATA #IMPLIED
       AllowsNulls     CDATA #IMPLIED
       ErrorMsg        CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>
```

El Patrón de Selección Definida (DefinedSelection) contiene una lista de propiedades propia. El elemento DefSelItem define cada uno de los elementos de selección.

```
<!-- L3.2 Defined Selection Pattern ============= -->
<!ELEMENT DefinedSelection (DefSelItem*)>
<!ATTLIST Introduction
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED
       DefaultValue    CDATA #IMPLIED
       MinSelectable   CDATA #IMPLIED
       MaxSelectable   CDATA #IMPLIED
       AllowsNulls     CDATA #IMPLIED
       ErrorMsg        CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>

<!ELEMENT DefSelItem EMPTY>
<!ATTLIST DefSelItem
       Code            CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
```

El Patrón de Información Complementaria (OIDFeedback) queda expresado indicando un nombre de clase y conjunto de información complementaria.

```
<!-- L3.3 OID Feedback  ========================== -->
<!ELEMENT OIDFeedback EMPTY>
<!ATTLIST OIDFeedback
       Class            CDATA #REQUIRED
       DisplaySet       CDATA #REQUIRED>
```

El Patrón de Dependencia se compone de una o más reglas ECA. A su vez, cada regla ECA es una fórmula bien formada según se describe en el apéndice D.

```
<!-- L3.4 Dependency Pattern  ===================== -->
<!ELEMENT Dependency (ECARule+)>
<!ELEMENT ECARule (Formula)>
```

Por su naturaleza implícita, el Patrón de Recuperación de estado no requiere de especificación.

```
<!-- L3.5 Status Recovery ======================== -->
<!-- Implicit behavior. Not specified. -->
```

El Patrón de Agrupación de Argumentos se representa como el elemento ArgumentGrouping. Este elemento contiene una lista de nodos de agrupamiento (ArgGroup.Node), donde cada nodo, o bien es un agrupador y contiene Alias y una lista de subnodos, o bien es un nombre de argumento y no contiene subnodos.

```
<!-- L3.6 Argument Grouping  ========================== -->
<!ELEMENT ArgumentGrouping (ArgGroup.Node*)>
<!ELEMENT ArgGroup.Node (ArgGroup.Node*)>
<!ATTLIST ArgGroup.Node
       Alias           CDATA #IMPLIED
       Argument.Name   CDATA #IMPLIED>
```

Un Filtro queda determinado por una fórmula bien formada (*véase apéndice C*) y una lista de variables.

```
<!-- L3.7 Filter  ========================== -->
<!ELEMENT Filter (Formula, FilterVariable*)>
<!ATTLIST Filter
       Name            CDATA #REQUIRED
```

```
       Alias           CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>

<!ELEMENT FilterVariable (FilterVar.AppliedPatterns?)>
<!ATTLIST FilterVariable
       Name            CDATA #REQUIRED
       Domain          CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
```

Un Criterio de Ordenación (OrderCriterium) se define como una lista de elementos (OrderCriteriumItem) donde cada elemento es una fórmula bien formada con una secuencia de roles hasta alcanzar una atributo visible y un sentido de ordenación: ascendente (ASC) o desdencente (DES).

```
<!-- 3.8 Order Criterium ========================== -->
<!ELEMENT OrderCriterium (OrderCriteriumItem+)>
<!ATTLIST OrderCriterium
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>
<!ELEMENT OrderCriteriumItem (Formula)>
<!ATTLIST OrderCriteriumItem
       Direction       (ASC | DES) #REQUIRED>
```

Un Conjunto de Visualización está compuesto por una lista elementos (DisplaySetItem) donde se expresa una fórmula para designar un atributo visible y un alias a dicho atributo.

```
<!-- L3.9 Display Set ============================ -->
<!ELEMENT DisplaySet (DisplaySetItem+)>
<!ATTLIST DisplaySet
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED>

<!ELEMENT DisplaySetItem (Formula)>
<!ATTLIST DisplaySetItem
       Alias           CDATA #IMPLIED>
```

El Patrón de Acciones se expresa como una lista de elementos (ActionItem) donde cada elemento contiene una Unidad de Interacción destino y un alias.

```
<!-- L3.10 Actions ============================== -->
<!ELEMENT Actions (ActionItem+)>
<!ATTLIST Actions
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
```

```
<!ELEMENT ActionItem EMPTY>
<!ATTLIST ActionItem
       TargetIU        CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
```

El Patrón de Navegación se expresa como una lista de elementos (NavigationItem) donde cada elemento contiene una Unidad de Interacción destino, un alias y una fórmula bien formada expresando un camino de roles hasta la clase de la Unidad de Interacción destino.

```
<!-- L3.11 Navigation ============================ -->
<!ELEMENT Navigation (NavigationItem+)>
<!ATTLIST Navigation
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
<!ELEMENT NavigationItem (Formula)>
<!ATTLIST NavigationItem
       TargetIU        CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
```

Una Unidad de Interacción de Servicio (UI.Service) tiene como propiedades emergentes el nombre, un alias y un mensaje de ayuda. El patrón de agrupación de argumentos puede (opcionalmente) ser aplicado.

```
<!-- L2.1IU Service ============================== -->
<!ELEMENT IU.Service (ArgumentGrouping?)>
<!ATTLIST IU.Service
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>
```

La Unidad de Interacción de Instancia (UI.Instance) contiene una referencia a un Cjto. de Visualización (obligatorio) y referencias opcionales a Acciones y Navegación.

```
<!-- L2.2 IU Instance ============================ -->
<!ELEMENT IU.Instance (Ref.DisplaySet, Ref.Actions?,
                       Ref.Navigation?)>
<!ATTLIST IU.Instance
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>
```

La Unidad de Interacción de Población (UI.Population) contiene listas de referencias a filtros, criterios de ordenación, cjtos. de visualización (al menos un elemento) así como referencias opcionales a Acciones y Navegación.

```
<!-- L2.3 IU Population =========================== -->
<!ELEMENT IU.Population (Ref.Filter*, Ref.OrderCriterium*, Ref.DisplaySet+,
                         Ref.Actions?, Ref.Navigation?)>
<!ATTLIST IU.Population
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED>
```

La Unidad de Interacción de Maestro/Detalle (IU.MasterDetail) contiene una referencia a una Unidad de Interacción que actúe como maestro (MasterIU) y una lista de detalles (DetailIU). Cada detalle se especifica con una fórmula bien formada (secuencia de roles atravesados desde la clase maestro a la clase detalle), un alias y el nombre de una Unidad de Interacción que actuará como Detalle (DetailTargetIU).

```
<!-- L2.4 IU Master/Detail ============================ -->
<!ELEMENT IU.MasterDetail (DetailIU*)>
<!ATTLIST IU.MasterDetail
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED
       HelpMsg         CDATA #IMPLIED
       MasterIU        CDATA #REQUIRED>

<!ELEMENT DetailIU (Formula)>
<!ATTLIST IU.MasterDetail
       Alias           CDATA #IMPLIED
       DetailTargetIU  CDATA #REQUIRED>
```

Las referencias a patrones son elementos XML creados para expresar el reuso existente en el modelo. Con su uso, se expresa la idea de que se referencia (se usa) un elemento definido previamente. Como enlace para el campo de referencia se usa el nombre del patrón que se pretende reusar.

```
<!-- Auxiliar components -->
<!ELEMENT Ref.Introduction (#PCDATA)>
<!ELEMENT Ref.DefinedSelection (#PCDATA)>
<!ELEMENT Ref.Filter (#PCDATA)>
<!ELEMENT Ref.OrderCriterium (#PCDATA)>
<!ELEMENT Ref.DisplaySet (#PCDATA)>
<!ELEMENT Ref.Actions (#PCDATA)>
<!ELEMENT Ref.Navigation (#PCDATA)>
```

El Árbol de Jerarquía de Acciones (HAT) esta compuesto por un nodo inicial o raíz. Cada nodo (HAT.Node) puede tener, a su vez, una lista de nodos hijos para la composición del árbol. Los nodos hoja no tiene subnodos y contienen una referencia a una Unidad de Interacción (TargerIU).

```
<!-- L1. Hierarchical Action Tree ============= -->
<!ELEMENT HAT (HAT.Node)>
<!ATTLIST HAT
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
<!ELEMENT HAT.Node (HAT.Node*)>
<!ATTLIST HAT.Node
       Alias           CDATA #IMPLIED
       TargetIU        CDATA #IMPLIED>
```

La lista de patrones que son definidos en el ámbito de una clase se presentan a continuación: UI de Instancia, UI de Población, UI de Maestro/Detalle, Filtros, Criterios de Ordenación, Conjuntos de Visualización, Acciones y Navegación.

```
<!-- Defined patterns for Classes ============= -->
<!ELEMENT Class.DefinedPatterns (IU.Instance*, IU.Population*,
                IU.MasterDetail*, Filter*, OrderCriterium*,
                DisplaySet*, Actions*, Navigation*)>
```

La lista de patrones aplicables a una clase consta de los siguientes elementos: Un patrón de información complementaria por defecto (OIDFeedback), una UI de Instancia y de Población para realizar consultas sobre la población de la clase.

```
<!-- Applied patterns for Classes ============= -->
<!ELEMENT Class.AppliedPatterns (OIDFeedback?, Default.IU.Instance?,
                                 Default.IU.Population?)>
<!ELEMENT Default.IU.Instance (#PCDATA)>
<!ELEMENT Default.IU.Population (#PCDATA)>
```

Los atributos pueden ser decorados con el Patrón de Introducción o, alternativamente con el Patrón de Selección Definida.

```
<!-- Applied patterns for Atributtes ============= -->
<!ELEMENT Atributte.AppliedPatterns (Ref.Introduction? |
                                     Ref.DefinedSelection?)>
```

En el ámbito de los servicios pueden ser definidas Unidades de Interacción de Servicios.

```
<!-- Defined patterns for Services ============= -->
<!ELEMENT Service.DefinedPatterns (IU.Instance*)>
```

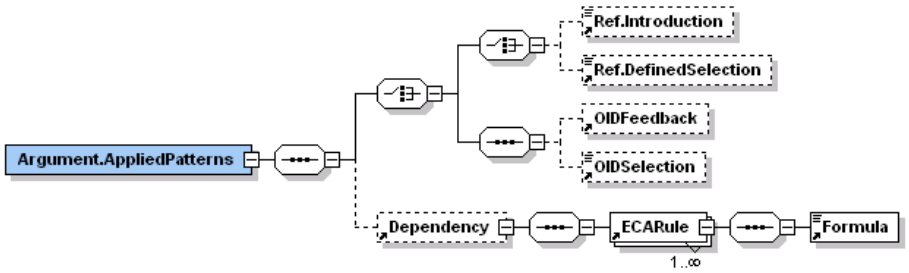


Figura B.4: Estructura del elemento XML <Argument.AppliedPatterns>.

Los argumentos pueden usar el Patrón de Introducción o el de Patrón de Selección Definida si su dominio es dato-valuado o bien tener asociado un Patrón de Información Complementaria y una Unidad de Interacción de Población para la Selección de Objetos si su dominio es objeto-valuado. Cada argumento puede tener un Patrón de Dependencia aplicado (*véase Figura B.4*).

```
<!-- Applied patterns for Arguments  ============= -->
<!ELEMENT Argument.AppliedPatterns (((Ref.Introduction? |
              Ref.DefinedSelection?) |
              (OIDFeedback?, OIDSelection?)), Dependency?)>
```

Las variables de filtro pueden usar el Patrón de Introducción o el de Patrón de Selección Definida si su dominio es dato-valuado o bien tener asociado un Patrón de Información Complementaria y una Unidad de Interacción de Población para la Selección de Objetos si su dominio es objeto-valuado.

```
<!-- Applied patterns for Filter Variables ============= -->
<!ELEMENT FilterVar.AppliedPatterns ((Ref.Introduction? |
              Ref.DefinedSelection?) |,
             (OIDFeedback?, OIDSelection?))>
```

Los argumentos y variables de filtros cuyo dominio es objeto-valuado pueden designar una Unidad de Interacción de Población que usará para facilitar la selección de las instancias. De no existir, se usará el definido por defecto en la clase.

```
<!-- OID Selection ============= -->
<!ELEMENT OIDSelection (#PCDATA)>
```

Una vista se define como una colección de interfaces. La vista puede definir un Árbol de Jerarquía de Acciones (HAT) (*véase Figura B.5*).

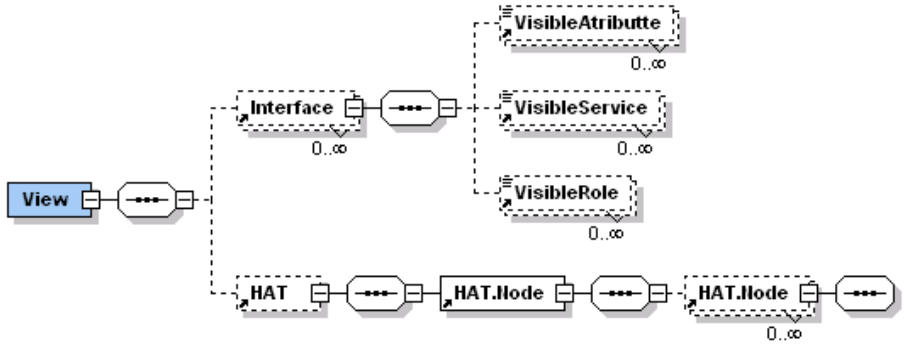


Figura B.5: Estructura del elemento XML `<View>`.

```
<!-- View (Subset of the model to derive a user interface)=== -->
<!ELEMENT View (Interface*,HAT?)>
<!ATTLIST View
       Name            CDATA #REQUIRED
       Alias           CDATA #IMPLIED>
```

Cada interfaz se compone de una lista de atributos visibles, servicios ejecutables y roles atravesables para un par de clases (ServerClass y ClientClass). La clase cliente (ServerClass) juega el rol de observadora, agente o actora, mientras que la clase servidora (ClientClass) juega el rol de clase observada.

```
<!-- Interface (for visiblity permisions) ============= -->
<!ELEMENT Interface (VisibleAtributte*,VisibleService*,VisibleRole*)>
<!ATTLIST Interface
       ServerClass     CDATA #REQUIRED
       ClientClass     CDATA #REQUIRED>
<!ELEMENT VisibleAtributte (#PCDATA)>
<!ELEMENT VisibleService (#PCDATA)>
<!ELEMENT VisibleRole (#PCDATA)>
```

Apéndice C

Formalización de las fórmulas de filtros

Las fórmulas de filtro están destinadas a permitir la consulta de las poblaciones de objetos existentes en la sociedad de objetos correspondiente a un esquema conceptual.

De manera análoga a como una sentencia `Select` de SQL permite recuperar información de una base de datos de modo declarativo, las fórmulas de filtro [Molina98] permiten consultar la población de una clase expresando la consulta en términos del espacio del problema.

Un filtro es una fórmula bien formada semejante a las fórmulas de precondición[1] de $\mathcal{OASIS}$: ambas se evalúan sobre un objeto a un valor lógico.

C.1. Sintaxis de las fórmulas

Las fórmulas de filtro se construyen de manera inductiva tal y como sigue:

- **Términos**

  1. Cada constante de un *sort* (género o tipo) es un término de dicho *sort*.
  2. Cada variable de filtro es un término en un *sort* dado.
  3. Si $f : s_1 \times s_2 \times \ldots \times s_n \rightarrow s$ es una función, y $t_1, t_2, \ldots, t_n$ son términos donde cada t_i pertenece al *sort* s_i, entonces $f(t_1, t_2, \ldots, t_n)$ es un término del *sort* s.

[1] Donde las variables del filtro juegan el mismo papel que los argumentos del servicio en una precondición.

4. Todo atributo es un término de un *sort* dado.
5. Toda secuencia $rol_1.rol_2. \ \ldots \ .rol_n.atributo$ donde el conjunto de rol_i constituyen una secuencia de roles de agregación univaluada y $atributo$ es un atributo válido en la clase destino, es un término de un *sort* dado.
6. Solo son términos los construidos siguiendo estas reglas sintácticas.

- **Átomos**

1. Las constantes `true` y `false` son átomos.
2. Si A y B son términos de un mismo tipo **C**, y op es un operador relacional (como p.e.: $=$, $<$, $>$, $\leq$, $\geq$, $=$, $\neq$, $LIKE$) definido para **C**, entonces $A\,op\,B$ es un átomo.
3. Solo son átomos los construidos de la forma 1 y 2.

- **Fórmula bien formada**

1. Un átomo es una fórmula bien formada.
2. Si V y W son fbf's entonces

$$\begin{cases} V \cup W & \text{(o lógico),} \\ V \cap W & \text{(y lógico) y} \\ \neg V & \text{(negación lógica)} \end{cases}$$

   son fbf's también.
3. Solo son fbf's las construidas de forma 1 y 2.

Desde el punto de vista categórico[2], un filtro puede verse como un morfismo que tiene como dominio un conjunto de objetos y como codominio un subconjunto de objetos del mismo conjunto de partida, esto es:

Sea f un filtro definido en una clase C_A y **A** un conjunto de objetos pertenecientes a una clase compatible[3] con C_A, entonces se cumple que al aplicar el filtro a la población de objetos **A** se obtiene un conjunto de objetos **B** que son subconjunto de **A**:

$$f(v_1, v_2, \ldots, v_n) : \mathbf{A} \rightarrow \mathbf{B}, \qquad \text{donde: } \mathbf{B} \subset \mathbf{A} \tag{C.1}$$

Las variables de filtro (v_i) definidas dentro de un filtro f dado permiten parametrizar la función de filtrado, actuando de modo análogo a como los parámetros determinan el comportamiento de una función.

Entrada		**Proceso**		**Salida**
A	$\rightarrow$	$f(v_1, v_2, \ldots, v_n)$	$\rightarrow$	B

(C.2)

[2]Desde el enfoque de la Teoría de Categorías.

[3]Compatible: Misma clase o especializada a través de relación o relaciones de herencia.

C.2. Semántica asociada

Para describir la semántica asociada a un filtro, comenzaremos introduciremos la semántica de un filtro aplicado a un objeto para después extender la definición a la aplicación sobre conjuntos de objetos.

C.2.1. Semántica de filtro aplicado a un objeto

Las fórmulas de filtro son fórmulas bien formadas que se resuelven a un valor lógico en el contexto de un objeto. La evaluación a cierto de un filtro sobre un objeto expresa que el objeto cumple la condición impuesta por el filtro.

Definición C.2.1 *Función de evaluación de un filtro para un objeto.*

Sea: $f_{ins}(t_1, t_2, \ldots, t_n)$ *un filtro definido en la clase* $\mathbf{C}$ *instanciado con los valores* (t_i)*.*

Sea x *un objeto de la clase* $\mathbf{C}$*.*

Definimos la función de evaluación de un filtro como:

$$eval(f_{ins}(t_1, t_2, \ldots, t_n), x) \rightarrow Boolean$$

o más abreviadamente:

$$eval(f_{ins}, x) \rightarrow Boolean$$

donde la función se evalúa como sigue:

$$eval(f_{ins}, x) = \begin{cases} true & \Leftrightarrow \text{ el objeto } x \text{ satisface la condición } f_{ins} \\ false & \Leftrightarrow \text{ el objeto } x \text{ no satisface la condicion } f_{ins} \end{cases}$$

Un objeto satisface la condición de filtro si al evaluar la condición instanciada de filtro sobre el objeto el resultado el cierto. Y recíprocamente, no la satisface si la condición se evalúa a falso.

C.2.2. Semántica de un filtro aplicado a un conjunto de objetos

La semántica asociada a un filtro $f(v_1, v_2, \ldots v_n)$ aplicado sobre un conjunto de objetos $\mathbf{A}$ es la siguiente:

$$\begin{aligned} & f(v_1, v_2, \ldots v_n) : \mathbf{A} \quad \rightarrow \quad \mathbf{B} \\ & \mathbf{A} \quad \rightarrow \quad f(v_1, v_2, \ldots v_n) \quad \rightarrow \quad \mathbf{B} \\ & \textit{Donde se cumple que:} \quad \mathbf{B} \subset \mathbf{A} \end{aligned}$$

A continuación se detalla como se obtiene dicho conjunto $\mathbf{B}$.

1. Las variables de filtro (v_i) toman valor en el transcurso de una consulta y son sustituidas (o instanciadas) por los valores introducidos por el usuario (t_i):

$$f(v_1, v_2, \ldots v_n) \quad \longrightarrow_{sustitución} \quad f_{ins}(t_1, t_2, \ldots, t_n) \tag{C.3}$$

   De este modo la fórmula f_{ins} es cerrada[4].

2. El filtro con los valores de las variables instanciados f_{ins} se evalúa a un valor lógico (cierto o falso) para cada uno de los objetos del conjunto $\mathbf{A}$.

$$\forall\, obj \in \mathbf{A}: \quad eval(f_{ins}(t_1, t_2, \ldots, t_n), obj) \rightarrow Bool \tag{C.4}$$

3. El conjunto $\mathbf{B}$ está compuesto solamente por aquellos objetos obj incluidos en $\mathbf{A}$ que satisfacen f_{ins}, es decir, aquellos que satisfacen la fórmula de filtro.

$$\forall\, obj \in \mathbf{A}: \quad obj \in \mathbf{B} \quad \Leftrightarrow \quad eval(f_{ins}(t_1, t_2, \ldots, t_n), obj) = true \tag{C.5}$$

C.3. Composición de filtros

$$\mathbf{A} = Pob(C_A) \quad \rightarrow \quad f(v_1, v_2, \ldots v_n) \quad \rightarrow \quad \mathbf{B}$$

$$\textit{Donde se cumple que:} \quad \mathbf{B} \subset Pob(C_A)$$

Cuando se toma la población completa de una clase como conjunto origen ($\mathbf{A} = Pob(C_A)$) para la aplicación de un filtro f, el filtro devuelve aquellos *objetos de la población completa* que satisfacen la condición de filtro ($\mathbf{B}$).

Sin embargo, no hay restricción para tomar siempre $\mathbf{A}$ como la población total de una clase, sino que los objetos perteneciente al conjunto $\mathbf{A}$ puede ser un subconjunto arbitrario de objetos con la única condición de que sean instancias de una clase compatible respecto a la clase de definición del filtro f.

En particular, esta libertad, nos permite componer filtros sencillos, para construir filtros más complejos.

Definición C.3.1 *Composición de filtros.*

$$\textit{Sea: } f_1(v_1, v_2, \ldots v_n) \textit{ un filtro definido en la clase } \mathbf{C}: \quad \mathbf{A} \rightarrow f_1 \rightarrow \mathbf{B}$$

$$\textit{Sea: } f_2(u_1, u_2, \ldots v_k) \textit{ un filtro definido en la clase } \mathbf{C}: \quad \mathbf{A} \rightarrow f_2 \rightarrow \mathbf{B}'$$

[4]No quedan variables libres del tipo v_i pendientes de tomar un valor.

definimos $g = f_2 \circ f_1$ *del siguiente modo:*

$$\mathbf{A} \rightarrow \underbrace{f_1() \rightarrow \mathbf{B} \rightarrow f_2()}_{g()} \rightarrow \mathbf{Z}$$

$$\mathbf{A} \quad \longrightarrow \quad g() \quad \longrightarrow \quad \mathbf{Z}$$

y donde $g()$ *tiene el siguiente perfil:*

$$g(\underbrace{v_1, v_2, \ldots v_n}_{n\ veces}, \underbrace{u_1, u_2, \ldots v_k}_{k\ veces}) : \mathbf{A} \quad \rightarrow \quad \mathbf{Z}$$

Y se cumple que: $\mathbf{Z} = \mathbf{B} \cap \mathbf{B}'$

La composición definida tiene semántica de *intersección* de conjuntos. Por tanto, podemos emplear en siguiente azúcar sintáctico:

$$f_1 \circ f_2 \qquad \equiv \qquad f_2 \cap f_1$$

Donde la intersección cumple las propiedades: idempotencia (C.6), conmutativa (C.7) y asociativa (C.8).

$$f_1 \cap f_1 = f_1 \tag{C.6}$$

$$f_1 \cap f_2 = f_2 \cap f_1 \tag{C.7}$$

$$(f_1 \cap f_2) \cap f_3 = f_1 \cap (f_2 \cap f_3) \tag{C.8}$$

C.4. Ejemplos

A continuación veremos un ejemplo de uso de la intersección de filtros:

Sea f_1 un filtro para buscar coches por color:	$f_1(v_{color})$
cuya fbf es:	$v_{color} = color$

donde v_{color} es una variable de filtro y $color$ es un atributo de la clase $coche$.

Sea f_2 un filtro para buscar coches por marca:	$f_2(v_{marca})$
cuya fbf es:	$v_{marca} = marca$

donde v_{marca} es una variable de filtro y $marca$ es un atributo de la clase $coche$.

Instanciando el color a `'rojo'` y la marca a `'Lamborghini'`, al aplicar los filtros a la población completa de la clase nos quedaría que:

$Pob(coches)$	$\rightarrow$	$f_1(`rojo')$	$\rightarrow$ coches de color rojo
$Pob(coches)$	$\rightarrow$	$f_2(`Lamborghini')$	$\rightarrow$ coches Lamborghini

Aplicando ambos filtros consecutivamente, obtendríamos:

$$f_1(v_{color}) \cap f_2(v_{marca}), \quad \textit{es decir:}$$

Pob(coches) $\rightarrow f_1(`rojo') \rightarrow f_2(`Lamborghini')$ *→ coches Lamborghini y rojos*

C.5. Conclusiones

Se ha formalizado la sintaxis de las fórmulas de filtros mediante la declaración de construcción de las fórmulas bien formadas. La semántica de los filtros ha quedado definida empleando Teoría de Categorías.

La composición (o intersección) de filtros permite construir filtros complejos a partir de la intersección de dos filtros más sencillos. Esta técnica favorece la reutilización de filtros y reduce la sobre-especificación de fórmulas de filtro.

Por otro lado, las propiedades conmutativa C.7 y transitiva C.8 de la intersección permiten a las implementaciones buscar optimizaciones[5] en función del coste de búsqueda de cada uno de los filtros simples.

[5]Optimizaciones equivalentes al problema de recuperar información de bases de datos relacionales a partir de sentencias *select* anidadas.

Apéndice D

Formación de reglas de dependencia

Para facilitar la lectura se proporciona un ejemplo:

En una encuesta de población activa (EPA) se pregunta a cada sujeto entrevistado si realiza trabajo remunerado, a continuación si la respuesta es afirmativa se le pregunta por el sueldo percibido. Es claro que parece incongruente contestar "No" a la primera pregunta y "5.600.000 Pta" a la segunda.

D.1. Aplicación del concepto de AIO

La clase `AIO` *(Abstract Interaction Object)* [Bodart96] está predefinida. Como toda clase, puede contener atributos, servicios, evaluaciones y restricciones bien definidas. Cada argumento de un servicio tendrá asociado una instancia de la clase `AIO` que será la encargada de intermediar entre el argumento y el usuario [Molina98]. Es sobre estos objetos (uno por cada argumento de servicio), sobre los que se modela el patrón de dependencia en base a reglas ECA[1] (Evento-Condición-Acciones).

D.1.1. Atributos

La clase `AIO` presenta atributos que caracterizan el estado del objeto AIO. Tales atributos son:

[1] Reglas ECA similares a las empleadas en las bases de datos activas para desencadenar actividad.

- `Active`: Es un valor lógico que indica si el usuario puede interactuar con el objeto AIO o no.
- `Value`: Contiene el valor actual del objeto AIO, el valor que tomará el argumento.

D.1.2. Eventos

Como eventos relevantes para la clase `AIO` se tiene:

- `SetValue(n_valor)`: Es el evento que permite cambiar el valor del objeto AIO.
- `SetActive(activo)`: Es el evento que permite activar o desactivar el objeto AIO.

D.2. Patrón de dependencia

El analista especificará el patrón de dependencia a través de la declaración de reglas ECA sobre los asociados a los argumentos involucrados dentro de un servicio.

El ejemplo expuesto anteriormente puede quedar especificado con el siguiente par de reglas:

```
[User]:trabaja.SetValue(a) :
                    a = 'Si' : sueldo.SetActive(true)
[User]:trabaja.SetValue(a) :
                    a = 'No' : sueldo.SetActive(false);
                                 sueldo.SetValue(0)
```

Tabla D.1: Ejemplo de reglas de dependencia

Cada vez que se produce un cambio de valor en el AIO `trabaja` (y por tanto cambia el valor asociado al argumento) se evalúan las reglas asociadas ejecutándose solo aquellas que se cumplen.

D.2.1. Sintaxis empleada

Las reglas de dependencia especificadas como reglas ECA constan de tres secciones bien diferenciadas. Tienen la forma:

```
<evento> : <condición> : <acción>
```

Sección Evento

En esta primera sección de la regla se especificará un evento perteneciente a la signatura de la clase `AIO` prefijado por el agente que lanzó el servicio y el nombre de la instancia de AIO que lo disparó. Existen dos agentes predefinidos, `User` e `Internal`[2]. El primero hace referencia a un cambio provocado por el usuario, el segundo hace referencia a un cambio provocado por el sistema. El evento presenta entre paréntesis variables separadas por comas, para referenciar los argumentos del evento. Solo se permite hacer referencia a un único evento de un único AIO por cada regla.

Ejemplo: `[User]:Trabaja.SetValue(a)`

EL ejemplo hace referencia al disparo por parte del usuario del evento de cambio del valor sobre el AIO `trabaja`, donde la variable `a` se instancia al nuevo valor.

Sección Condición

Cuando se lanza el evento, se comprueba si se satisface la condición de esta segunda sección. La condición es una fórmula bien formada de *OASIS* con sintaxis similar empleada en precondiciones [Pastor95]. Dentro de la fórmula pueden aparecer operadores relacionales entre atributos de instancias de AIO y las variables que aparecen en la primera sección para realizar la sustitución de los argumentos del evento.

En esta sección se permite el acceso a los atributos de cada AIO usando como notación: `<instancia_aio>.<atributo>`

Aunque a primera vista este modo de proceder pudiera parecer una violación de la encapsulación, debe tenerse en cuenta que esta notación simplifica la lectura (frente al empleo de eventos o funciones de consulta) de las condiciones, a la vez que no presenta ningún problema si toma la asunción de que dentro del servicio actual existe visibilidad entre las instancias de AIO asociadas a este servicio.

Ejemplo: `a = 'No' AND Trabaja.valor='Si'`

La condición comprueba que la variable que contiene el nuevo valor (`a`) es igual a 'No' y comprueba que el valor antiguo (`Trabaja.valor`) tenía como valor 'Si'.

[2]Cuando no es necesario distinguir cual de los dos agentes disparo el evento, se emplea el símbolo asterisco: *.

Sección Acción

Finalmente, la última sección establece la secuencia de acciones que deben llevarse a cabo solo cuando el evento se produce y la condición se satisface. Las acciones van separadas por el operador `'.'` que indica ejecución en secuencia de acciones. Las acciones válidas serán eventos de una instancia de AIO. Los argumentos de las acciones pueden instanciarse a partir de las variables definidas en la sección de evento, o bien tomar su valor empleando eventos de consulta sobre los atributos de instancias de AIO. Las acciones disparadas por reglas de dependencia son etiquetadas con el agente `Internal`. Esto es importante para conocer si se dispararán nuevas reglas o no.

Ejemplo: `Sueldo.SetActive(false) ; Sueldo.SetValue(0)`

Esta sección de acción está compuesta por dos acciones. En primer lugar se desactiva el AIO `sueldo` y posteriormente se fija su valor a cero. Nótese que tanto una como otra acción, pueden disparar a su vez nuevas reglas de dependencia.

D.2.2. Orden de ejecución de las reglas

La ejecución de una regla puede conllevar consigo el disparo de otras. Por ello es necesario fijar la política para la ejecución de las reglas.

1. Cuando se produce un evento, se consultan las reglas disponibles **en el orden de definición** de las mismas y se procede a lanzar aquellas que encajan con el evento una tras otra.

2. Cuando se produce un encadenamiento de reglas (una regla que en su sección de acción dispara a su vez nuevas reglas) se **interrumpe la ejecución de la regla actual y se procede a lanzar la regla recién disparada**. Una vez se ha ejecutado, se prosigue con la ejecución de la regla que interrumpió su ejecución en el punto donde se dejo.

D.2.3. Ciclos en reglas

Debido a que se permite el encadenado de reglas, es posible que conjuntos de reglas puedan contener ciclos. Dichos ciclos pueden desencadenar bloqueos[3] en tiempo de ejecución.

Dentro del contexto del patrón de dependencia en interfaz de usuario no tiene mucho sentido dichas combinaciones. Por lo tanto la herramienta que

[3]Bloqueos debidos al encadenamiento de reglas de manera recursiva.

permita editar este tipo de reglas debería ayudar a la localización de los posibles ciclos en las reglas que el analista esta definiendo.

Apéndice E

Guía de estilo para aplicaciones Windows

Los principios enumerados en la sección 3.2.5. *«Principios básicos»* son muy genéricos y pueden aplicarse a cualquier entorno gráfico, sin embargo, conforme se precisa la implementación, se desciende de lo abstracto a lo particular, perdiendo generalidad.

Este es el caso de las reglas que se expondrán a continuación, que son aplicables para una familia de entornos muy concretos: Windows. Las reglas han sido extraídas de informes técnicos y recomendaciones [Europea95, Europea96a, Europea96b, Europea96c] elaboradas por al Dirección de Informática de la CEE (Comisión Económica Europea). Son consejos concisos acerca de como debe comportarse la aplicación en un entorno Windows 95. Aunque ha llovido bastante desde la aparición de Windows 95, lo cierto es que la familia de interfaces posteriores: Windows NT 4.0, Windows 98, Windows Millenium, Windows 2000, y Windows XP siguen manteniendo el líneas generales apariencia y comportamiento *(look & feel)* similar a Windows 95. Por este motivo, la guía sigue en vigor y es aplicable a toda la familia de interfaces gráficas en estos operativos.

Esta guía de Estilo para es un ejemplo muy concreto para una plataforma particular: Windows. Sin embargo, la aplicación una guía es estilo es totalmente trasladable a terceros entornos.

E.1. Reglas de estilo para Windows

A continuación se enuncian ciento setenta y una reglas de estilo de bajo nivel muy precisas, muchas de las cuales podrían ser verificadas automática-

mente sobre el código fuente de una aplicación.

El cumplimiento de la guía de estilo dota de una fuerte homogeneidad a la aplicación y de coherencia respecto a otras aplicaciones dentro de la misma plataforma.

1. El texto en el título de una barra de una ventana o caja de diálogo siempre se alinea a la izquierda.

2. Empleo de la fuente MS San Serif normal de 8 puntos para el título de la ventana.

3. Uso de la fuente MS San Serif normal de 8 puntos para menús e iconos.

4. La tecla TAB está reservada para desplazarse al siguiente control mientras que la combinación de teclas Shift+TAB se desplaza al control previo.

5. Una caja de diálogo no se puede maximizar, minimizar o alterar su tamaño a través de los bordes. No tiene icono de aplicación y la barra de titulo contiene un botón de cierre.

6. Los atajos *(shortcuts keys)* estándar son:
   - Ctrl + X : Cortar
   - Ctrl + C : Copiar
   - Ctrl + V : Pegar
   - Ctrl + Z : Deshacer *(Undo)*
   - Ctrl + A : Abrir *(versiones anglosajonas : O ⇒ open)*
   - Ctrl + P : Imprimir
   - Ctrl + G : Guardar *(versiones anglosajonas: S ⇒ save)*

7. Los atajos *(shortcuts keys)* estándar para ventanas MDI son:
   - Ctrl + F6 : Muestra la siguiente ventana de documento
   - Ctrl + F4 : Cierra la ventana del documento activo
   - Ctrl + W : Cierra la ventana del documento activo

8. Los atajos *(shortcuts keys)* estándar para Windows son:
   - F1 : Muestra la ayuda de la aplicación
   - Shift + F1 : Activa / Desactiva el modo de ayuda sensible al contexto Botón: '¿Que es esto?'
   - Shift + F10 : Presenta un menú emergente sensible al contexto, accesible también con el botón derecho del ratón
   - Alt : Activa la barra de menú
   - Alt + Tab : Presenta la siguiente ventana de aplicación
   - Alt + F4 : Cierra la aplicación o una caja de diálogo

9. Es obligatorio que todas las aplicaciones contengan como primer titulo de menú una entrada denominada Fichero de donde pende, al menos la opción de Salir.

10. El menú de Edición se emplea sólo para la gestión del portapapeles y operaciones de búsqueda y reemplazamiento.

11. Si el menú de Edición está presente, debe contener al menos, los siguientes elementos:

Deshacer	Ctrl + Z
Cortar	Ctrl + X
Copiar	Ctrl + C
Pegar	Ctrl + V

12. La barra de menú de una aplicación MDI contiene al menos los títulos de menú Fichero, Edición y Ayuda.

13. Todos los títulos y elementos de menú tienen un atajo equivalente mediante el teclado que queda presente mediante el subrayado de una de sus letras (Alt + <letra>).

14. Para cada botón de comando de en una barra de herramientas siempre debe existir un equivalente en el menú.

15. El menú de Ayuda contiene, al menos, las entradas: Temas de Ayuda y Acerca de ...

16. Si la aplicación proporciona de un modo general, menús sensibles al contexto, el menú de Ayuda contendrá un elemento Asistente para Ayuda para mostrar la ayuda sensible al contexto.

17. En una aplicación MDI, siempre existe un menú de ventana con los siguientes elementos:

Cascada
Mosaico
Organizar todo
Seguido de la lista de las ventanas (MDI hijas) abiertas.

18. En una aplicación MDI, si se abren más de 10 ventanas a la vez, el menú de lista de ventanas contiene un elemento por cada una de las nueve primeras ventanas más un elemento que da acceso al resto de ventanas abiertas.

19. Los elementos de menú que invocan a cajas de diálogo siempre van seguidos de elipsis (...).

20. Un elemento de menú no debe estar aislado entre dos barras separadoras a menos que sea el primero o el último.

21. El título de un menú en la barra de menús es siempre una única palabra.

22. El título de un menú siempre se escribe como un nombre propio, donde sólo la primera letra es mayúscula.

23. Los nombres de elementos de menús no están limitados a una palabra pero no deben exceder de cuatro (incluyendo el artículo). Sólo la primera palabra se escribe como un nombre propio.

24. La barra de menús de una aplicación SDI contiene al menos títulos de menú Fichero, Edición y Ayuda.

25. Los elementos de un menú emergente en una aplicación siempre tienen un equivalente en el menú convencional de la aplicación.

26. Si un elemento de menú se emplea para seleccionar un modo de operación con lógica de tipo si/no, entonces: el estado de habilitado debe ser visualmente retroalimentado a través de una marca a la izquierda del elemento activo.

27. Como mucho pueden aparecer hasta siete y como poco dos elementos en cada menú[1].

28. Como máximo pueden aparecer hasta siete títulos de menú en la barra de menús.

29. Los elementos del menú que tienen la misma naturaleza funcional se agrupan y se separan por líneas horizontales (barras separadoras).

30. Un menú sin elementos, que sólo contiene un titulo, lleva adjunto un signo de exclamación (!).

31. Los menús en cascada están limitados a un sólo nivel de anidamiento.

32. Los elementos de los menús emergentes están sujetos a las mismas reglas al igual que los menús convencionales (elipsis, número de elementos limitados a siete, etc.).

33. En las aplicaciones MDI, el menú de ventana contiene las entradas Cerrar y Cerrar todos que permiten cerrar el documento activo y cerrar todos los documentos abiertos respectivamente.

[1] Regla de ergonómica conocida como 7 ± 2 elementos. Tiene como base de fundamento estudios psicológicos que han determinado el número de elementos que son capaces de retener las personas en su memoria rápida.

34. Las teclas de acceso para la barra de menús son:
    Alt + A para Archivo
    Alt + E para Edición
    Alt + V para Ventana
    Alt + Y para Ayuda
    Aunque dependiendo de los títulos de menú presentes puede ser necesario variar el criterio.

35. Un elemento de menú aparece deshabilitado si la acción asociada con el elemento no está disponible.

36. Un titulo de menú debe aparecer siempre activo, incluso aunque todos los elementos que contiene no estén disponibles.

37. En una caja de diálogo con hojas de pestañas, los botones de comando asociados a todas las hojas de pestañas deben situarse fuera del área de las hojas de pestañas, mientras que los botones de comando asociados a una sola hoja de pestaña debe situarse dentro de la hoja asociada.

38. La ventana principal de una aplicación contiene una barra de título, barra de menús, un icono de aplicación, los botones de cierre, maximizar/restaurar y minimizar, además la ventana se puede mover y cambiar su tamaño.

39. Una ventana de documento MDI (ventana MDI hija) contiene una barra de título, un icono de documento, los botones de cierre, maximizar/restaurar y minimizar, además la ventana puede cambiar su tamaño y puede moverse dentro del espacio de trabajo de la ventana MDI padre.

40. Una ventana hija de un proyecto *(project workspace)* contiene una barra de título, barra de menús, un icono de documento, los botones de cierre, maximizar/restaurar y minimizar, además la ventana puede cambiar su tamaño y puede moverse sin limitaciones puesto que no está ligada a la ventana principal del proyecto. La ventana hija puede contener barras de herramientas y barra de estado.

41. Una ventana hija MDI no contiene botones de comando, barra de menús, barra de herramientas ni barra de estado.

42. Una ventana primaria SDI o MDI contiene una única barra de menús, que puede gestionarse dinámicamente de acuerdo al contexto de la aplicación.

43. La barra de herramientas debe ocupar el ancho total de la ventana independientemente del tamaño de la ventana.

44. La barra de estado debe ocupar el ancho total de la ventana independientemente del tamaño de la ventana.

45. La barra de título de una aplicación SDI contiene el nombre del objeto que está siendo tratado (documento, hoja de calculo, etc.), el nombre de la aplicación y el icono de la aplicación.

46. La barra de título de una aplicación MDI contiene el nombre de la aplicación y el icono de la aplicación.

47. La barra de título de una ventana hija MDI contiene el nombre y el icono del documento que se está tratando.

48. Debe ser posible colocar la barra de herramientas en cualquiera de los limites de la ventana: arriba debajo del menú, abajo sobre la barra de estado y a la izquierda y a la derecha. En ocasiones puede interesar barras de herramientas flotantes. En cualquier caso siempre debe ser posible ocultarla.

49. La barra de estado debe poder ocultarse.

50. Para los botones de comando se empleará la fuente MS San Serif normal de 8 puntos.

51. Para el texto modificable y controles se empleará la fuente MS San Serif normal de 8 puntos.

52. Empleo de la fuente MS San Serif normal de 8 puntos para texto estático.

53. Empleo de la fuente MS San Serif normal de 8 puntos para los campos de texto estático que describen a un control.

54. El texto que describe a un control puede incluir varias palabras, pero sólo la primera se escribe como un nombre propio (con la primera letra en mayúsculas).

55. El botón de comando Aceptar se escribe con sólo la primera letra mayúscula. Su tecla equivalente es la tecla de retorno de carro.

56. El color de fondo para cajas de diálogo es gris o blanco, con preferencia por el gris para soportar los efectos 3D.

57. Los efectos de relieve no deben emplearse para los campos de texto estático.

58. Los controles de edición, cajas de listas, cajas desplegables y cajas combinadas tienen como color de fondo el blanco, en lugar del color de fondo empleado en la caja de diálogo.

59. Un campo de texto estático no tiene borde y tiene el mismo color de fondo que la caja diálogo que lo contiene.

60. Las cajas de edición que cambian su estado de acuerdo al contexto (activas o no activas) siempre tienen borde y aparecen con un color de fondo gris cuando están desactivadas y blanco cuando están activas.

61. Un campo de texto que describe un objeto inactivo debe estar también desactivado.

62. Los botones de comando siempre tienen apariencia 3D, tres estados (soltado, pulsado e inactivo) y gris claro como color de fondo.

63. El botón de comando Cancelar se escribe con sólo la primera letra mayúscula. Su tecla equivalente es la tecla de Escape.

64. El botón de comando Ayuda se escribe con sólo la primera letra mayúscula. Su tecla equivalente es la tecla F1.

65. El tamaño de los controles debe ser proporcional al tamaño de la información que contienen.

66. Cada carácter mostrado empleando la fuente MS San Serif normal de 8 puntos ocupa como promedio:
    5 píxeles para una letra minúscula
    6 píxeles para una letra mayúscula
    5 píxeles para un dígito numérico

67. Un botón de comando en una caja de diálogo sólo debe abrir otra caja de diálogo.

68. La aplicación debe limitarse a tres niveles de profundidad (número de ventanas anidadas en un diálogo).

69. Una caja de diálogo no debería contener más de cinco botones de comando.

70. Es recomendable el empleo de hojas con etiquetas para secuenciar cajas de diálogo.

71. La barra de herramientas y reglas tienen la misma anchura.

72. Los botones de comando en una barra de herramientas están dispuestos en el mismo orden que los comandos del menú. Los botones asociados con elementos en el mismo menú se presentan contiguos y están separados del resto de grupos por 6 pixeles.

73. El contenido de la barra de herramientas puede variar al igual que el menú.

74. La barra de herramientas y reglas deben poder ser mostradas como paletas flotantes.

75. Si se emplea información constante para identificar una situación, es preferible mostrarla en una barra de contexto en lugar de repetirla en cada caja de diálogo o ventana.

76. La barra de contexto debe colocarse debajo de la barra de herramientas o debajo del menú si no existe barra de herramientas.

77. Una barra de contexto puede contener los siguientes controles: botones de verificación, listas desplegables, cajas combinadas, cajas de texto, texto estático y botones de radio gráficos.

78. La información de la barra de contexto puede mostrarse en la barra de herramientas si el espacio lo permite.

79. Los botones de radio gráficos deben obligatoriamente colocarse agrupados para indicar las diferentes posibles elecciones.

80. Todos los botones de comando de una caja de diálogo o al menos los botones de un grupo deben tener el mismo tamaño y alineación.

81. Los nombres de los botones de comando deben ir acompañados de teclas de acceso rápido para activarlos mediante la combinación de la tecla Alt + <la tecla seleccionada>.

82. Un botón de comando que abre una caja de diálogo va siempre seguido de elipsis (...).

83. Un campo de texto que describe un control cuyo contenido puede ser modificado tiene una tecla de acceso rápido.

84. Un campo de texto que describe un control cuyo contenido no puede ser modificado no tiene una tecla de acceso rápido.

85. Un campo de texto que describe un control va siempre seguido por dos puntos más un espacio (': ').

86. Un botón de comando que extiende la caja de diálogo ofreciendo más detalles va seguido de '>>'. El tamaño inicial se restablece por un botón cuyo nombre va seguido de '<<'.

87. Los botones de comando tienen como mínimo 72 píxeles de ancho y como máximo 100.

88. Los botones de comando se alinean siempre en la parte inferior derecha o en la parte superior derecha, dependiendo de la lógica del diálogo, excepto en las cajas de mensajes donde los botones aparecen centrados en la parte inferior.

89. Los botones Aceptar y Cancelar se colocan siempre los primeros en la caja de diálogo.

90. Las listas, cajas combinadas, listas desplegables y las cajas de spin siempre tienen borde.

91. Una lista debe permitir mostrar un máximo de ocho elementos y un mínimo de tres.

92. Una lista con dos o más columnas que muestran información diferente debe tener cabeceras de columna.

93. Una lista de tipo vista contiene texto e iconos de tamaño 32x32 o 16x16 píxeles.

94. Una lista desplegable siempre contiene el elemento '(ninguno)' en la primera posición si la selección es opcional.

95. Las cajas combinadas y las listas desplegables no tienen cabeceras de columna.

96. El contenido de los controles (cajas de texto, cajas combinadas, etc.) siempre se muestra alineado a la izquierda.

97. Los números se alinean a la izquierda excepto en las tablas donde se alinean a la derecha.

98. Los campos de texto asociados con el control de grupo se alinea a la izquierda.

99. Los separadores de hojas de etiquetas *(tabs dividers)* se alinean a la izquierda respecto a la ventana que los contiene y son adyacentes unos a otros.

100. Los separadores de hojas de etiquetas *(tabs dividers)* tienen un tamaño proporcional al texto que contienen o son todos del mismo tamaño.

101. El texto de los separadores de hojas de etiquetas está siempre centrado, se muestra en una línea y puede contener imágenes (en formato 16x16 pixeles) y/o texto.

102. Los separadores de hojas de etiquetas tienen efecto 3D, el mismo color de fondo que la caja de diálogo y usan la fuente MS San Serif normal de 8 puntos.

103. Un botón de comando puede contener un nombre, un icono (formato 16x16 píxeles) o ambos.

104. Un botón de comando puede desplegar un menú con una serie de acciones posibles.

105. El espacio entre un control y el limite izquierdo o derecho de la caja de diálogo es de 10 píxeles. El espacio entre un control y el limite inferior o superior es de 8 pixeles.

106. Una lista debe tener ajustado su ancho para evitar mostrar una barra de desplazamiento horizontal.

107. Es preferible usar botones de comando con imágenes sólo en la barra de herramientas pero no en las cajas de diálogo.

108. Una caja de diálogo por lo general es modal con la excepción de las cajas de búsqueda.

109. El tamaño de una caja de diálogo depende de los controles que contiene.

110. Las cajas de diálogo deben ser preferentemente rectangulares (más anchos que altos) para encajar en el espacio de pantalla que también es más ancha que alta.

111. Los controles y los campos de texto se alinean a la izquierda.

112. Los bordes de los controles están justificados y, si no lo están, están alineados a la izquierda.

113. Es preferible disponer los controles en vertical en lugar de horizontal.

114. Un campo de texto estático asociado con una caja de texto está siempre alineado horizontalmente con él. Un campo de texto estático asociado con una caja de texto rico o una lista puede alinearse verticalmente.

115. Los botones que tienen el mismo tipo de funcionalidad se agrupan (espaciados 6 píxeles verticalmente y 8 pixeles horizontalmente) y están separados de otros grupos de botones por más espacio (18 pixeles horizontalmente y 16 pixeles verticalmente).

116. El botón empleado para confirmar una acción o un conjunto de acciones siempre se etiqueta como Aceptar.

117. El formateado de los datos y las comprobaciones de filtrado se realizan siempre al abandonar el control y no sólo al abandonar la caja de diálogo.

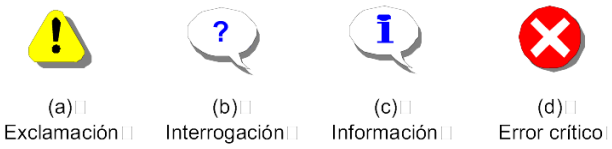


Figura E.1: Tipos de iconos para cajas de mensaje en Windows.

118. Si se detecta un error al confirmar una caja de diálogo (por ejemplo pulsando sobre el botón Aceptar), se muestra un mensaje de error y el foco vuelve al control en cuestión.

119. La pulsación sobre el botón Cancelar en una caja de diálogo que contiene errores cierra la caja de diálogo sin mostrar errores al usuario.

120. Todos los controles en una caja de diálogo deben alinearse a la izquierda y a la derecha y arriba y abajo para balancear el diálogo.

121. El espacio entre dos controles es de 8 píxeles horizontalmente y de 6 píxeles verticalmente.

122. La pulsación del botón Cancelar para cerrar una caja de diálogo lanza una caja de mensaje de confirmación si se editaron los datos de la caja de diálogo.

123. Los mensajes de las cajas de mensajes con más de una línea deben alinearse a la izquierda. Cada nueva sentencia comienza en una nueva línea.

124. El título de una caja de mensaje se alinea a la izquierda y contiene el nombre objeto sobre el cual el usuario está trabajando o el nombre de la aplicación.

125. Las cajas de mensajes son siempre modales.

126. En las cajas de mensajes deben emplearse los iconos estándares para identificar la situación.

127. Windows reconoce cuatro tipos de mensajes (*véase la Figura E.1*):

    - Mensajes de Información
    - Mensajes de Alerta
    - Mensajes de Error
    - Mensajes Críticos

128. Las cajas de mensaje se emplean para informar al usuario del resultado de sus acciones y contienen:

- El icono 'i'
- El botón Aceptar

129. Un mensaje de alerta informa al usuario que la situación requiere una decisión o que la acción solicitada puede causar una perdida de datos. Este mensaje contiene:

    - El icono '!'
    - Los botones principales Aceptar, Si / No o Si / No / Cancelar.

130. Un mensaje de error indica que el sistema detectó un error serio que previene contra la continuación de la acción. Un mensaje de error contiene:

    - El icono 'X'
    - Los botones principales Aceptar o Reintentar / Cancelar.

131. El puntero debe convertirse en un reloj de arena siempre una la acción se inicia para prevenir que otra acción se intente lanzar en paralelo.

132. El puntero debe convertirse en un puntero más reloj de arena siempre una la acción se inicia pero otra acción puede llevarse a cabo en la aplicación multihilo.

133. Debe mostrarse un indicador de progreso gráfico siempre que el tiempo requerido para procesar una acción exceda de cinco segundos.

134. Las cajas de mensajes deben tener el mismo color de fondo que las cajas de diálogo.

135. Es aconsejable emplear sólo como señal de audio el pitido del sistema (para no perder información en sistemas que no tienen capacidades de audio).

136. Una señal de audio no puede emplearse sola como retroalimentación de una acción, debe acompañarse de una respuesta visual.

137. El uso de un indicador de progreso debe acabar con un mensaje de información que indique al usuario que el proceso se ha completado de modo satisfactorio.

138. La ayuda en línea de Windows es una ventana con tres *tabs* (Contenido, Índice, Búsqueda).

139. En un indicador de progreso, el botón de comando Cancelar o Detener se emplean para interrumpir el proceso y devolver el control al usuario.

140. Los textos de los mensajes deben ser cortos y amistosos y no incluirán términos como *equivocación* o *acción errónea*.

141. El texto del mensaje debe sugerir una solución en lugar de solamente informar de un error.

142. Un mensaje debe tener dos o tres líneas a lo sumo.

143. Si un mensaje va acompañado de un número, el número debe aparecer siempre al principio del mensaje.

144. La tecla F1 queda reservada para servir de tecla de acceso rápido a la ayuda en línea.

145. La ayuda en una caja de diálogo se accede pulsando sobre el botón ¿Qué es esto? en la barra de título, pulsando en el botón de Ayuda o pulsando la tecla F1.

146. Las opciones del menú: Ayuda o Asistente para Ayuda muestran ayuda general acerca de los contenidos de una caja de diálogo o de la ventana de documento activa.

147. Windows proporciona iconos estándares para las barras de herramientas. Estos iconos solamente deben emplearse en las acciones estándar.

148. El diálogo estándar para la apertura de archivos debe emplearse para buscar un archivo navegando a través de las carpetas.

149. El diálogo estándar para la salvaguarda de archivos debe emplearse para especificar la carpeta en la cual, un archivo es salvado.

150. El diálogo estándar de búsqueda debe emplearse para buscar cadenas de caracteres en un texto.

151. El diálogo estándar de reemplazamiento debe usarse para reemplazar cadenas de caracteres en un texto.

152. El diálogo estándar de impresión se emplea para definir las propiedades de impresión a emplear.

153. El diálogo estándar de fuentes se usa para seleccionar las tipografías a emplear.

154. El diálogo estándar de selección de color se emplea para seleccionar colores.

155. Los botones de una barra de herramientas tienen dos tamaños posibles:

    - 24x22 píxeles con iconos de tamaño 16x16 píxeles o
    - 32x30 píxeles con iconos de tamaño 24x24 píxeles.

156. En una lista, las cabeceras de columna se alinean a la izquierda si la columna contiene texto y a la derecha si contiene números que se alinean a la derecha.

157. Una caja de diálogo debe contener un indicador de progreso cuando el procesamiento involucre una situación modal en la aplicación.

158. Los indicadores de progreso deben colocarse en la barra de estado cuando el procesamiento no involucra una situación modal en la aplicación.

159. El puntero de ayuda sensible al contexto se representa mediante un puntero acompañado de un símbolo de interrogación '?'.

160. La ayuda sensible al contexto es accesible por:

    - El elemento ¿Qué es esto? del menu de ayuda.
    - El botón ¿Qué es esto? de la barra de título.
    - El botón ¿Qué es esto? de la barra de herramientas.
    - Un menú emergente accesible pulsando con el botón derecho cuando el elemento ¿Qué es esto? está activo.

161. La ayuda sensible al contexto de cada menú y cada elemento debe aparecer en la barra de estado.

162. El botón de Ayuda es siempre el último de un conjunto de botones de comando.

163. Las ventanas de aplicación, las ventanas hijas MDI y las ventajas hijas de proyectos son siempre no modales.

164. Deben proporcionarse descripciones emergentes (*tooltips*) para los botones gráficos de las barras de herramientas, las imágenes de una lista o en hojas con etiquetas (*tabs*).

165. Los botones de verificación tienen un color de fondo blanco cuando pueden ser modificados y un fondo de color gris cuando no pueden ser modificados.

166. La ayuda sensible al contexto ¿Qué es esto? presenta ayuda sobre un componente de una caja de diálogo o ventana.

167. Si la aplicación maneja ayuda sensible al contexto inmediata, el menú de ayuda contiene la entrada ¿Qué es esto? para mostrar la ayuda sensible al contexto inmediata de los componentes de una caja de diálogo o una ventana.

168. La combinación de teclas Alt + F4 suele tener el mismo efecto que pulsar el botón de cancelación.

169. El botón de cierre se emplea para concluir un diálogo de consulta.

170. El mensaje mostrado en la barra de estado no puede ocupar más de una línea.

171. Un mensaje de error crítico informa al usuario que la aplicación ha causado un error que requiere el cierre de la aplicación. Este mensaje contiene:

    - El icono 'X'.
    - Los botones de Cerrar y Detalles.

Apéndice F

Abreviaturas

AIO Objeto de interfaz abstracto, *(Abstract Interface Object)*.

API Interfaz del programador de aplicaciones, *(Application Programmer Interface)*. Constituye la interfaz pública de invocación de métodos ofertada por una librería para facilitar el acceso a los servicios que encapsula.

ASP Páginas de Servidor Activas, tecnología de *script* de Internet Information Server, Microsoft Corp., *(Active Server Pages)*.

CASE Ingeniería de Software asistida por ordenador, *(Computer Aided Software Engineering)*. Métodos, técnicas y herramientas informáticas destinadas a facilitar y dar soporte al desarrollo de tareas propias de la Ingeniería del Software.

CIO Objeto de interfaz concreto, *(Concrete Interface Object)*.

COM *(Common Object Model)*. Arquitectura de componentes propuesta por Microsoft Corp. para la invocación de objetos.

CORBA *(Common Object Request Broker Architecture)*.

CTT Notación de Tareas desarrollada por Paternò [Paternò00] *(Concur Task Tree)*.

CUA *(Common User Access)*. Estándar definido por IBM [IBM92] para el diseño aplicaciones en entornos gráficos que se ha convertido en un estándar industrial de facto para entornos Windows.

DSN Nombre de origen de datos en el standard ODBC, *(Data Source Name)*.

DTE Diagrama de Transición de Estados.

F.b.f. Formula bien formada que cumple las restricciones gramaticales un lenguaje dado.

GUI Interfaz gráfica de usuario, *(Graphical User Interface)*.

HLL Lenguaje alto nivel, *(High Level Language)*.

HTML Lenguaje de marcas de hipertexto, *(HyperText Markup Language)*.

IDE Entorno de desarrollo Integrado, *(Integrated Development Environment)*.

IU Unidad de interacción, *(Interaction Unit)*.

IIOP Protocolo de interconexión de ORB para Internet, *(Internet Inter-ORB Protocol)*.

JSP Páginas de Servidor de Java, *(Java Server Pages)*.

MB-UIDE Entornos de desarrollo de interfaces de usuario basadas en modelos, *(Model Based User Interface Development Environments)*.

MDI Interfaz de documentos multiple, *(Multiple Document Interface)*. Estándar para la creación de aplicaciones que permiten contener múltiples documentos abiertos de modo simultáneo.

OASIS *(Open Active Specification of Information Systems)*. Lenguaje de especificación formal orientado a objetos para modelos conceptuales.

ODBC *(Object DataBase Conectivity)*. Protocolo estandarizado que proporciona acceso a bases de datos independizando a las aplicaciones del tipo de base de datos (fabricante) y localización (local o remota).

OMG *(Object Management Group)*.

OOA Análisis orientado a objetos, *(Object–Oriented Analysis)*. Primera metodología orientada a objetos propuesta por Coad y Yourdon [Coad91].

OO-MBCG Termino acuñado por Bell [Bell98] para referir la generación de código basada en modelos orientados a objetos, *(Object–Oriented Model Based Code Generation)*.

OO-UI Interfaces de usuario orientadas a objetos, *(Object Oriented User Interfaces)*.

PDA Asistente para datos personales, *(Personal Data Assistant)*.

PU Unidades de presentación, *(Presentation Units)*.

RAD Desarrollo rápido de aplicaciones, *(Rapid Application Development)*.

RGB Rojo, verde y azul, *(Red, Green & Blue)*. Es el sistema de codificación más empleado en ordenadores para expresar tonos de color. Se basa en que cada color queda determinado por una terna de porcentajes de intensidad de color relativos a los colores primarios aditivos: rojo, verde y azul.

RMI Invocación Remota a Métodos. Mecanismo de RPC nativo en Java. *(Remote Method Invocation)*.

RPC Protocolo de invocación remota de procedimientos. *(Remote Procedure Call)*.

SDI Interfaz de documentos simple, *(Simple Document Interface)*. Estándar para la creación de aplicaciones contrapuesto a MDI donde solo es posible disponer de un documento simultáneamente.

SOAP Protocolo de acceso a objecto simple, *(Simple Object Access Protocol)*.

UIMS *(User Interface Management Systems)*. Aplicación o conjunto de herramientas para facilitar el análisis, diseño, implementación y mantenimiento de la interfaz de usuario. Constituyen la primera generación de herramientas para soportar el diseño de interfaces de usuario.

XML Lenguaje de marcas extendido, *(eXtended Markup Language)*.

WAP Protocolo para aplicaciones inalámbricas, *(Wireless Application Protocol)*.

WIMP Ventana, icono, ratón y menú, *(Window, Icon, Mouse & Menu)*. Son la esencia de los conceptos introducidos por Xerox en *Palo Alto Research Center* a comienzos de los 80s. que dan lugar a las interfaces gráficas de usuario.

WML Lenguaje de marcas del WAP, *(WAP Markup Language)*.

WYSIWYG Lo que ve es lo que obtiene, *(What You See Is What You Get)*.Técnica de interacción donde la retroalimentación al usuario y la manipulación directa toman un papel primordial.

Apéndice G

Glosario de términos

A

AIO Objeto de interfaz abstracto, *(Abstract Interface Object)* [Vanderdonckt93].

Alias Etiqueta visible para el usuario en la interfaz de usuario. Los son por tanto dependientes de idioma y pueden ser traducidos para soportar aplicaciones multi-idioma.

API Interfaz del programador de aplicaciones, *(Application Programmer Interface)*. Constituye la interfaz pública de invocación de métodos ofertada por una librería para facilitar el acceso a los servicios que encapsula.

C

CASE Ingeniería de Software asistida por ordenador, *(Computer Aided Software Engineering)*. Métodos, técnicas y herramientas informáticas destinadas a facilitar y dar soporte al desarrollo de tareas propias de la Ingeniería del Software.

CIO Objeto de interfaz concreto, *(Concrete Interface Object)* [Vanderdonckt93].

COM *(Common Object Model)*. Arquitectura de componentes propuesta por Microsoft Corp. para la invocación de objetos.

CORBA *(Common Object Request Broker Architecture)*.

CUA *(Common User Access)*. Estándar definido por IBM [IBM92] para el diseño aplicaciones en entornos gráficos que se ha convertido en un estándar industrial de facto para entornos Windows.

D

DSN Nombre de origen de datos en el standard ODBC, *(Data Source Name)*.

DTE Diagrama de Transición de Estados.

F

Evento Método que al ser lanzado provoca un cambio en la vida de uno o más objetos. Acepción del término empleado en *OASIS* y OO-Method.

Evento de interfaz Evento producido por un entorno gráfico, gestor de ventanas o sistema operativo que recoge las acciones del usuario como pueden ser la pulsación de teclas o las acciones llevadas a cabo por el ratón.

F

F.b.f. Formula bien formada que cumple las restricciones gramaticales un lenguaje dado.

G

GUI Interfaz gráfica de usuario, *(Graphical User Interface)*.

H

HTML Lenguaje de marcas de hipertexto, *(HyperText Markup Language)*.

I

IDE Entorno integrado de desarrollo, *(Integrated Development Environment)*. Son editores asociados a un lenguajes de programación que facilitan el trabajo de programación. Suelen incorporar facilidades para el diseño visual de interfaces de usuario para el lenguaje que soportan.

Interfaz Dentro de contexto de *OASIS* y OO-Method, una interfaz es una relación binaria entre dos clases (agente y servidora) donde se especifica la visibilidad sobre atributos, servicios y roles de la clase servidora desde la clase agente.

IU Unidad de interacción, *(Interaction Unit)*.

L

Layout Disposición geométrica de los controles o widgets en una ventana. Existen algoritmos de disposición de controles que producen buenos resultados de modo automático. Sin embargo, también hay excepciones que deben ser contempladas o gestionadas por un diseñador experto.

Lenguaje de Patrones Conjunto de patrones definidos dentro de un mismo contexto que utilizados de modo conjunto son útiles para resolver problemas complejos dentro del contexto en el cual se definen.

M

MB-UIDE Entornos de desarrollo de interfaces de usuario basadas en modelos, *(Model Based User Interface Development Environments)*.

MDI Interfaz de documentos multiple, *(Multiple Document Interface)*. Estándar para la creación de aplicaciones que permiten contener múltiples documentos abiertos de modo simultáneo.

Modelo Conceptual Un Modelo Conceptual es una especificación de un sistema de información que trata de recoger los requisitos del sistema en términos de conceptos en el espacio del problema.

Modelo de Presentación El Modelo de Presentación es un modelo basado en un lenguaje de patrones para recopilar los requisitos de Interfaz de Usuario y establecer su relaciones con elementos de análisis del modelo conceptual.

O

OASIS *(Open Active Specification of Information Systems)*. Lenguaje de especificación formal orientado a objetos para modelos conceptuales.

ODBC *(Object DataBase Conectivity)*. Protocolo estandarizado que proporciona acceso a bases de datos independizando a las aplicaciones del tipo de base de datos (fabricante) y localización (local o remota).

OMG *(Object Management Group)*.

OOA Análisis orientado a objetos, *(Object–Oriented Analysis)*. Primera metodología orientada a objetos propuesta por Coad y Yourdon [Coad91].

OO-MBCG Termino acuñado por Bell [Bell98] para referir la generación de código basada en modelos orientados a objetos, *(Object–Oriented Model Based Code Generation)*.

P

Patrón Un patrón es la abstracción de un problema recurrente y una solución recurrente a dicho problema. De modo que esta catalogación de problemas ayuda a identificar problemas y a resolverlos de modo más sencillo.

Patrón Conceptual Es un patrón que surge en el contexto del Modelado Conceptual.

Patrón de Diseño Es un patrón que se aparece en contextos de diseño. Como ejemplos más representativos consúltese [Gamma95].

PDA Asistente para datos personales, *(Personal Data Assistant)*.

Pixel *Picture Element.* Cada punto de la pantalla capaz de alterar su color. Las pantallas o dispositivos de representación dibujan matrices de píxeles.

R

RAD Desarrollo rápido de aplicaciones, *(Rapid Application Development)*.

RGB Rojo, verde y azul, *(Red, Green & Blue)*. Es el sistema de codificación más empleado en ordenadores para expresar tonos de color. Se basa en que cada color queda determinado por una terna de porcentajes de intensidad de color relativos a los colores primarios aditivos: rojo, verde y azul.

S

SDI Interfaz de documentos simple, *(Simple Document Interface)*. Estándar para la creación de aplicaciones contrapuesto a MDI donde solo es posible disponer de un documento simultáneamente.

SOAP Protocolo de acceso a objeto simple, *(Simple Object Access Protocol)*.

U

UIMS *(User Interface Management Systems)*. Aplicación o conjunto de herramientas para facilitar el análisis, diseño, implementación y mantenimiento de la interfaz de usuario. Constituyen la primera generación de herramientas para soportar el diseño de interfaces de usuario.

X

X11 / X Window System X11 es el protocolo de ventanas de red estándar en entornos Unix que proporciona las primitivas de paso de mensajes para dar soporte al entorno o gestor de ventanas gráfico que se ejecuta sobre éste.

XML Lenguaje de marcas extendido, *(eXtended Markup Language)*.

V

Vista Conjunto de interfaces en un modelo OO-Method que conforman la unidad de generación para interfaces de usuario.

W

WAP Protocolo para aplicaciones inalámbricas, *(Wireless Application Protocol)*.

Widget *Window Gadget*. Control u objeto gráfico que se dibuja en pantalla y que da soporte a la interacción con el usuario en un entorno gráfico.

WIMP Ventana, icono, ratón y menú, *(Window, Icon, Mouse & Menu)*. Son la esencia de los conceptos introducidos por Xerox en *Palo Alto Research Center* a comienzos de los 80s. que dan lugar a las interfaces gráficas de usuario.

WML Lenguaje de marcas del WAP, *(WAP Markup Language)*.

WYSIWYG Lo que ve es lo que obtiene, *(What You See Is What You Get)*. Técnica de interacción donde la retroalimentación al usuario y la manipulación directa toman un papel primordial.

Bibliografía

[2U02] «2U Consortium», 2002. Disponible en http://www.2uworks.org. 55

[Abrams99a] Mark Abrams, Constantin Phanouriou, A.L. Batongbacal, S. Williams y J. Shuster. «UIML: An Appliance-Independent XML User Interface Language». En A. Mendelzon (Editor), «Proceedings of 8th Int. World-Wide Web Conference WWW'8», Elsevier Science Publishers, Toronto, Canada, Mayo 1999. Disponible en http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html. 256

[Abrams99b] Mark Abrams y Constantine Phanouriou. «UIML: An XML Language for Building Device Independent User Interfaces». En A. Mendelzon (Editor), «Proceedings of XML'99», Elsevier Science Publishers, Philadelphia, EE.UU., Diciembre 1999. Disponible en http://www.harmonia.com/resources/xml99Final.pdf. 256

[Aho86] Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, EE.UU., Enero 1986. ISBN 0201100886. 252

[Alexander64] Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964. 8, 72, 79

[Alexander75] Christopher Alexander. *The Oregon Experiment*. Oxford University Press, 1975. 79

[Alexander77] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. 79, 85, 117

[Alexander79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. 79, 81

[Appleton97] Brad Appleton. «Patterns and Software: Essential Concepts and Terminology», 1997. Disponible en http://www.enteract.com/~bradapp/docs/patterns-intro.html. 80

[Arg02] «Argo UML», 2002. Disponible en http://argouml.tigris.org/. 12, 236

[ASL01] «Action Semantics for the UML», Agosto 2001. Disponible en http://www.kc.com/as_site/download/ActionSemantics.zip. 55

[Azevedo00] Pedro Azevedo, Roland Merrick y Dave Roberts. «OVID to AUIML - User-Oriented Interface Modelling». En N.Ñunes (Editor), «Proceedings of 1st Workshop 'Towards a UML Profile for Interactive Systems Development' TUPIS'00», York, Reino Unido, Octubre 2000. Disponible en http://math.uma.pt/tupis00/submissions/azevedoroberts/azevedoroberts.html. 256

[Balzer83] R. Balzer, T.E. Cheatham y C. Green. «Software Technology in the 1990s: Using a New Paradigm». *IEEE Computer*, págs. 39–45, Noviembre 1983. 2, 236, 290

[Balzert96] Helmut Balzert. «From OOA to GUIs: The JANUS System.» *Journal of Object-Oriented Programming*, tomo 8(9), págs. 43–47, Febrero 1996. 22

[Barfield93] Lon Barfield. *The User Interface. Concepts & Design*. Addison-Wesley, Reading, MA, EE.UU., 1993. 77

[Belenguer02] Jorge Belenguer. «Concretización para PocketPC de los Patrones de Interfaz». Nota interna 132, CARE Technologies S.A., Denia, España, 2002. 266, 273, 275, 298, 380

[Bell98] Rodney Bell. «Code Generation from Object Models». *Embedded Systems*, Marzo 1998. Disponible en http://www.embedded.com/98/9803fe3.htm. 102, 109, 348, 353

[Bergin02] Joseph Bergin. «Building Graphical User Interfaces with the MVC Pattern», 2002. Disponible en http://csis.pace.edu/~bergin/mvc/mvcgui.html. 31, 254

[Bergman02] Lawrence D. Bergman, Guruduth Banavar, Danny Soroker y Jeremy Sussman. «Combining Handcrafting and Automatic Generation of User-Interfaces for Pervasive Devices». En Ch. Kolski y J. Vanderdonckt (Editores), «Computer-Aided Design of User Interfaces III», págs. 155–166. Kluwer Academics Publisher, Dordrecht, Valenciennes, Francia, Mayo 2002. ISBN 1-4020-0643-8. 92, 265

[Bodart93] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, Isabelle Sacré y Jean Vanderdonckt. «Architecture Elements for Highly-Interactive Business-Oriented Applications». *Lecture Notes in Computer Science*, tomo 153, págs. 83–104, 1993. 22

[Bodart94] François Bodart y Jean Vanderdonckt. «On the Problem of Selecting Interaction Objects». En S.W. Draper y G.R.S. Weir G. Cockton (Editor), «Proceedings of BCS Conf. HCI'94 "People and Computers IX"», págs. 163–178. Cambridge University Press, Cambridge, Reino Unido, 1994. 32

[Bodart95a] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, I. Provot, B. Sacré y Jean Vanderdonckt. «Towards a Systematic Building of Software Architectures: the TRIDENT methodological guide». En Ph. Palanque y R. Bastide (Editor), «Proceedings of 2nd Eurographics Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'95», págs. 262–278. Springer-Verlag, Viena, Austria, Junio 1995. 22

[Bodart95b] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, I. Provot, Jean Vanderdonckt y G. Zucchinetti. *Critical Issues in User Interface Systems Engineering*, capítulo 7. Key Activities for a Development Methodology of Interactive Applications, págs. 109–134. Springer-Verlag, Berlin, Alemania, 1995. 22

[Bodart95c] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux y Jean Vanderdonckt. «Computer-Aided Window Identification in TRIDENT». En «Proc. of 5ª conferencia HCI INTERACT'95», págs. 331–336. Chapman & Hall, 1995. 294

[Bodart95d] François Bodart y Jean Vanderdonckt. «Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide». En «Proceedings of Design, Specification and Verification of Interactive Systems, DSV-IS'95», págs. 262–278. Springer-Verlag, Junio 1995. 22, 24, 26, 27, 28, 29, 30, 33, 34, 375, 383

[Bodart96] François Bodart y Jean Vanderdonckt. «Widget Standardisation through Abstract Interaction Objects». En A.F. Özok y G. Salvendy (Editor), «Proceedings of 1st Int. Conf. on Applied Ergonomics ICAE'96», págs. 300–305. USA Publishing, Istanboul - West Lafayete, Mayo 1996. 15, 22, 125, 325

[Boehm88] B. Boehm. «A Spiral Model of Software Development and Enhancement». *IEEE Computer Society*, tomo 2(21), 1988. 48

[Ceri00] Stefano Ceri, Piero Fraternali y Aldo Bongio. «Web Modeling Language (WebML): a modeling language for designing Web sites». *Computer Networks (Amsterdam, Netherlands: 1999)*, tomo 33(1–6), págs. 137–157, 2000. Disponible en `http://citeseer.nj.nec.com/ceri00web.html`. 50

[Cleaveland01] J. Craig Cleaveland. *Program Generators with XML and Java*. Prentice Hall, Upper Saddle River, New Jersey, EE.UU., 2001. ISBN 0-13-025878-4. 4, 91, 98, 236, 240, 241, 243, 244, 251, 376, 379, 380

[Coad91] P. Coad y E. Yourdon. *Object Oriented Analisis*. Prentice Hall, 1991. 22, 107, 348, 353

[Computer87] Apple Computer (Editor). *Human Interface Guidelines. The Apple Desktop Interface*. Addison-Wesley, Reading, MA, EE.UU., 1987. 73, 76, 77

[Conallen00] Jim Conallen. *Building Web Applications with UML*. Addison-Wesley, Reading, MA, EE.UU., 2000. 55

[Constantine99] Larry Constantine y Lucy Lockwood. *Software for use. A practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, Reading, MA, EE.UU., 1999. 1, 194, 195

[Coo01] «Cool:Plex», 2001. Disponible en http://www3.ca.com/Press/PressRelease.asp?ID=1800. 14

[Coplein96] James O. Coplein. *Software Patterns*. SIGS Books & Multimedia, Nueva York, EE.UU., 1996. 81, 84

[Coplein98] James O. Coplein. «C++ Idioms». En «Proceedings of the EuroPlop'98.», 1998. Disponible en http://www.bell-labs.com/user/cope/Patterns/C++Idioms/EuroPLoP98.html. 86

[Corp.95] Microsoft Corp. (Editor). *The Windows Interface Guidelines for Software Design*. Microsoft Press, Redmond, Washinton, EE.UU., 1995. 73

[Coutaz87] Joëlle Coutaz. «PAC – An Object Oriented Method for Dialogue Design». En «Proceedings of the INTERACT'87 Conference», Elsevier Science, Stutgart, Alemania, 1987. 31, 254

[Cowart00] R. Cowart y B. Knittel. *Using Microsoft Windows 2000 Professional*. Que Corp., Indianapolis, IN., EE.UU., 2000. 65

[Czarnecki00] Krzysztof Czarnecki y Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, EE.UU., 2000. 236

[Donohoe98] Mark Donohoe. «The KDE Style Book», 1998. Disponible en http://developer.kde.org/documentation/standards/kde/style_book/style-book.html. 73

[Erikson97] Thomas Erikson y John Thomas. «CHI1997 Workshop. Putting It All Together: Towards a Pattern Language for Interaction Design», 1997. Resumen disponible en http://www.pliant.org/personal/Tom_Erickson/Patterns.WrkShpRep.html. 87

[ESDI93] ESDI (Editor). *OBLOG CASE V1.0 - User's Guide*. ESDI S.A., Lisboa, Portugal, 1993. 3, 54, 247

[Europea95] Comisión Europea. «Windows Applications Development Guide for C++». Informe técnico, Comisión Europea. Direction Informatique, 1995. 69, 70, 331

[Europea96a] Comisión Europea. «Style Guide for Windows 95 Applications. Volume 1. Basic Concepts». Informe técnico, Comisión Europea. Direction Informatique, 1996. 331

[Europea96b] Comisión Europea. «Style Guide for Windows 95 Applications. Volume 2. 200 Basic Rules». Informe técnico, Comisión Europea. Direction Informatique, 1996. 331

[Europea96c] Comisión Europea. «Style Guide for Windows 95 Applications. Volume 3. From Windows 3 to Windows 95». Informe técnico, Comisión Europea. Direction Informatique, 1996. 331

[eZw01] «eZway», 2001. Disponible en http://http://www.freewayinc.net/. 14

[Fincher03] Sally Fincher, Janet Finlay, Sharon Greene, Lauretta Jones, Paul Matchen, Pedro J. Molina y John Thomas. «CHI 2003 Workshop. Perspectives on HCI patterns: concepts and tools», 2003. Disponible en http://www.dsic.upv.es/~pjmolina/CHI2003WS/. 88

[Foley82] J.D. Foley y A. van Dam. *Fundamental of Interactive Computer Graphics*. Addison-Wesley, Reading, MA, EE.UU., 1982. 68

[Foley91] James D. Foley, Won Chul Kim, Srdjan Kovacevic y Kevin Murray. *Intelligent User Interfaces*, capítulo Chapter 15. UIDE an Intelligent User Interface Design Environment, págs. 339–384. ACM Press, Addison Wesley, Reading, MA, EE.UU., 1991. 18, 20, 375

[Foundation93] Open Software Foundation (Editor). *OSF/Motif Style Guide: Revision 1.2.*. Prentice Hall International, Englewood Cliffs, Nueva Jersey, EE.UU., 1993. 73

[Fowler96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Object-Oriented Software Engineering Series. Addison-Wesley, Reading, MA, EE.UU., Octubre 1996. ISBN 0201895420. 88

[Gamma95] Eric Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.*. Addyson Wesley, 1995. ISBN 0201633612. 14, 79, 82, 86, 88, 117, 118, 353

[Gartner01] Gartner. «CARE Technologies: Application Development. Projects Assessment Managenent Presentation». Performance benchmark, Gartner Group, April 2001. 280, 282, 283, 284, 285, 286, 287, 288, 289, 290, 380

[Genera00] Genera. «Genova 7.0», 2000. Disponible en http://www.genera.no/2052/tilkunde/09.04/default.asp. 15, 236

[Ginda00] R. Ginda. «Writing a Mozilla Application with XUL and Javascript». En «Proc. of O'Reilly Open Source Software Convention», O'Reilly, Monterey, EE.UU., Julio 2000. Disponible en http://www.mozilla.org/docs/ora-oss2000/chatzilla/overview.html. 256

[Gómez98] Jaime Gómez, Emilio Insfrán, Vicente Pelechano y Óscar Pastor. «The Execution Model: a Component-Based Arquitecture to Generate Software Components From Conceptual Models». En John Grundy (Editor), «Proceedings of International Workshop on Component-based Information Systems Engineering. 10th International Conference on Advanced Information Systems Engineering, CAiSE-98, Pisa (Italia).», págs. 87–94. 1998. ISSN 1170-487X. 61

[Gómez01] Jaime Gómez, Cristina Cachero y Óscar Pastor. «On Conceptual Modeling of Device-Independent Web Applications: Towards a Web Engineering Approach.» *IEEE Multimedia. Special Issue on Web Engineering.*, tomo 8(2), págs. 20–32, 2001. 50

[GNO02] «GNOME», 2002. Disponible en http://www.gnome.org. 73

[Goldberg83] A. Goldberg y D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, EE.UU., 1983. 31, 254

[Gomez99] Jaime Gomez, Óscar Pastor, Emilio Insfrán y Vicente Pelechano. «From Object-Oriented Conceptual Modeling to Component Based Development». En Trevor Bench-Capon, Giovanni Soda y A. Min Toa (Editores), «Database and Expert Systems Applications», págs. 332–341. Springer-Verlag, Florencia (Italia), 1999. ISBN 3-540-66448-3. 61

[Griffits00] Richard Griffits, Lyn Pemberton, Jan Borchers y Adam Stork. «CHI 2000 Workshop. Pattern Languages for Interaction Design: Building Momentum», 2000. Resumen disponible en http://www.stanford.edu/~borchers/hcipatterns/chi2000ws.html y en www.acm.org/sigchi/bulletin/2000.1/borchers.pdf. 88

[Group.02a] Apache Group. «Velocity», 2002. Disponible en http://jakarta.apache.org/velocity/. 251

[Group02b] W3C. XForms Working Group. «XForms - The Next Generation of Web Forms», 2002. Disponible en http://www.w3.org/MarkUp/Forms/. 257

[Harel87] David Harel. «Statecharts: A Visual Formalism for Complex Systems». *Science of Computer Programing*, tomo 8, págs. 231–274, 1987. 105, 252

[Hartmann94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel y J. Kusch. «Revised Version of the Modelling Language Troll (Troll version 2.0)». Informe técnico, Technische Universitat Braunschweig, Informatik-Berichte, April 1994. 2, 54

[i-L] «i-Logix». Disponible en http://www.ilogix.com. 107

[IBM92] IBM (Editor). *Object Oriented Interface Design: IBM Common User Access Guidelines*. Que, Carmel, IN., EE.UU., 1992. 73, 76, 156, 347, 351

[Iborra00] José Iborra y Óscar Pastor. «Automatic Software Production System», April 2000. Patente USPTO 09/543.085, SOSY Inc. 296

[Iborra02a] José Iborra y Óscar Pastor. «Automatic Software Production System», Mayo 2002. Patente USPTO 20020062475, SOSY Inc. 296

[Iborra02b] José Iborra y Óscar Pastor. «Automatic Software Production System», Julio 2002. Patente USPTO 20020100014, SOSY Inc. 297

[IFPUG01] IFPUG. «International Function Point User Group», 2001. International Function Point User Group, Disponible enhttp://www.ifpug.org. 280

[Inc.02] Macromedia Inc. «Macromedia», 2002. Disponible en http://www.macromedia.com. 11

[Insfrán99] Emilio Insfrán, Roel Wieringa y Óscar Pastor. «Using TRADE to improve an Object-Oriented Method». Informe técnico, University of Twente. Computer Science Department, Enschede, Holanda, Julio 1999. 121

[Insfrán00] Emilio Insfrán, Roel Wieringa y Óscar Pastor. «Requirements Engineering-Based Conceptual Modeling», 2000. Submited(Available at http://www.dsic.upv.es/~einsfran/publicaciones.html. 121

[Insfrán01] Emilio Insfrán, Pedro J. Molina, Sofía Martí y Vicente Pelechano. «Ingeniería de Requisitos aplicada al modelado conceptual de interfaz de usuario». En «Actas de IDEAS'2001», págs. 181–192. CIT, Santo Domingo, Heredia, Costa Rica, Abril 2001. 119, 120, 121, 122, 295

[IPI92] «The UIMS Workshop Tool developers: A Metamodel for the Runtime Architecture of An Interactive System.» *SIGCHI Bulletin*, tomo 1(24), págs. 32–37, 1992. 255, 380

[ISO88] ISO (Editor). *Information Process Systems - Open Systems Interconection - LOTOS - A Formal Description Based on Temporal Ordering of Observational Behaviour*. ISO Central Secretariat, 1988. ISO/IS 8807. 40, 203

[ITU96] ITU. «ITU: Recommendation Z. 120: Message Sequence Chart (MSC)», 1996. International Telecomunication Union. 50

[Jaaski95] Ari Jaaski. «Object-Oriented Specification of User Interfaces». *Software Practive & Experience*, Noviembre 1995. 50

[Javahery02] Homa Javahery y Ahmed Seffah. «A Model for Usability Pattern-Oriented Design». En Costin Pribeanu y Jean Vanderdonckt (Editores), «TAMODIA'200. 1st International Workshop on TAsk, MOdels and DIAgrams for user interface design», págs. 104–110. Academy of Economic Studies, Bucarest, Rumania, Julio 2002. 88

[Johnson89] J. Johnson, T.L. Roberts, W. Verplank, D.C. Smith, C. Irby, M. Beard y K. Mackey. «The Xerox Star. A retrospective». *IEEE Computer*, tomo 22(9), págs. 18–25, 1989. 65

[KDE99] «KDE User Interface Guidelines», 1999. Disponible en http://developer.kde.org/documentation/standards/kde/style/basics/index.html. 73

[Kennedy-Carter00] Kennedy-Carter. «Executable Modeling with the UML. The xUML method and iUML, the xUML tool set.» Informe Técnico CTN 80 v1.1, Kennedy–Carter Ltd., 2000. Disponible en http://www.kc.com. 109

[Lea99] Doug Lea. «Christopher Alexander: an introduction for Object-Oriented Designers», 1999. Disponible en http://www.enteract.com/~bradapp/docs/patterns-intro.html. 83

[Letelier98] Patricio Letelier, Isidro Ramos, Pedro Sánchez y Óscar Pastor. *OASIS Versión 3.0: Un enfoque formal para el modelado conceptual orientado a objeto*. Servicio de Publicaciones UPV, Valencia, España, 1998. ISBN 84-7721-663-0. 54, 57, 59, 174, 212, 301

[Lozano01] Mª Dolores Lozano. *Entorno Metodológico Orientado a Objetos para la especificación y Desarrollo de Interfaces de Usuario*. Tesis Doctoral, Universidad Politécnica de Valencia, Valencia, España, Diciembre 2001. 50

[Mac01] «Mac OS X», 2001. Disponible en http://www.apple.com/macosx/. 65

[Microsystems89] Sun Microsystems (Editor). *Open Look Graphical User Interface Application Style Guidelines*. Addison-Wesley, Reading, MA, EE.UU., 1989. 73

[Microsystems99] Sun Microsystems. «Java Look & Feel Design Guidelines», 1999. Disponible en http://java.sun.com. 73

[Molina98] Pedro J. Molina. *Especificación de Interfaz de Usuario En OO-Method*. Proyecto Fin de Carrera, Universidad Politécnica de Valencia, Valencia, España, Septiembre 1998. 4, 69, 70, 87, 115, 295, 319, 325

[Molina01] Pedro J. Molina, Óscar Pastor, Sofía Martí, Juan J. Fons y Emilio Insfran. «Specifying Conceptual Interface Patterns in an Object-Oriented Method with Code Generation». En «Proceedings of 2nd IEEE Workshop on User Interfaces for Data Intensive Systems UIDIS'01», págs. 72–79.

IEEE Computer Society, Zurich, Suiza, Mayo 2001. ISBN 0-7695-0834-0. 295

[Molina02a] Pedro J. Molina, Sofía Martí y Óscar Pastor. «Prototipado rápido de interfaces de usuario». En Miguel Katrib et al. (Editor), «V Workshop Iberoamericano de Ingeniería de Ambientes de Software (IDEAS 2002)», págs. 78–90. La Habana, Cuba, April 2002. ISBN 959-7160-14-5. 225, 295

[Molina02b] Pedro J. Molina, Santiago Meliá y Óscar Pastor. «User Interface Conceptual Patterns». En Peter Forbrig, Quentin Limbourg, Bodo Urban y Jean Vanderdonckt (Editores), «Design, Specification, and Verification of Interactive Systems», págs. 201–214. Université catholique de Louvaine, University of Rostock, Louvain-La-Neuve, Belgica y Rostock, Alemania, Junio 2002. 89, 90, 293, 296

[Molina02c] Pedro J. Molina, Santiago Meliá y Óscar Pastor. «User Interface Conceptual Patterns». En Peter Forbrig, Quentin Limbourg, Bodo Urban y Jean Vanderdonckt (Editores), «Proceedings of the 4th International Workshop on Design Specification & Verification of Information Systems DSV-IS'2002», Springer Verlag, Berlin, Alemania, Diciembre 2002. ISBN 3-540-00266-9. Disponible en http://www.springer.de/cgi/svcat/search_book.pl?isbn=3-540-00266-9. 89, 90, 293, 296

[Molina02d] Pedro J. Molina, Santiago Melia y Óscar Pastor. «Just-UI: A User Interface Specification Model». En Ch. Kolski y J. Vanderdonckt (Editores), «Computer-Aided Design of User Interfaces III», págs. 63–74. Kluwer Academics Publisher, Dordrecht, Valenciennes, Francia, Mayo 2002. ISBN 1-4020-0643-8. 89, 90, 115, 125, 295

[Molina02e] Pedro J. Molina, Ismael Torres y Óscar Pastor. «Patrones de interfaz de usuario para la navegación orientada a objetos». En Ignacio Aedo, Paloma Díaz y Camino Fernández (Editores), «III Congreso Interacción Persona Ordenador», págs. 27–36. AIPO, Leganes, España, Mayo 2002. ISBN 84-607-4501-5. 295

[Molina03a] Pedro J. Molina, Santiago Meliá y Óscar Pastor. «The Just-UI approach: Conceptual modelling of device independent interfaces». En Christophe Kolski y Jean Van-

derdonckt (Editor), «Journal of Human-Computer Interaction (RIHM). Special Issue on Computer-Aided Design of User Interfaces», 1. 2003. (En prensa). 296

[Molina03b] Pedro J. Molina, Óscar Pastor, Juan C. Molina y José M. Barberá. «Method and Apparatus for Automatic Generation of Information System User Interfaces», Enero 2003. Patente USPTO 10/356.250, SOSY Inc. 297

[Molina03c] Pedro J. Molina, Ismael Torres y Óscar Pastor. «Patrones de interfaz de usuario para la navegación orientada a objetos». *Novatica*, 2003. (En prensa). 296

[Molina03d] Pedro J. Molina, Ismael Torres y Óscar Pastor. «User Interface Patterns for Object-Oriented Navigation». *Upgrade*, 2003. (En prensa) Estará finalmente disponible en http://www.upgrade-cepis.org. 296

[Myers92] Brad A. Myers y Mary Beth Rosson. «Survey on User Interface Programming. In Striking a Balance». En «Proc. of CHI'92», tomo 992, págs. 195–202. ACM Press, Monterey, EE.UU., Mayo 1992. 233, 280, 289

[Nunes00a] Nuno Jardim Nunes y João Falcão e Cunha. *Object Modeling and User Interface Design*, capítulo 6. Whiteater Interactive System Development with Object Models. Object Technology Series. Addison-Wesley, Reading, MA, EE.UU., 2000. ISBN 0-201-65789-9. 48

[Nunes00b] Nuno Jardim Nunes y João Falcão e Cunha. «Wisdom: A Software Engineering Method for Small Software Development Companies». *IEEE Software*, 2000. 48, 49, 376

[Nunes01] Nuno Jardim Nunes. *Object Modeling for User-Centered Development and User-Interface Design*. Tesis Doctoral, Universidad de Madeira, Madeira, Portugal, 2001. 48

[ObjectTime] ObjectTime. «ObjectTime CASE Tool». Disponible en http://www.objectime.com. 107

[OMG97a] «UML Semantics. Version 1.1, documento OMG ad/97-08-04», Septiembre 1997. Disponible en http://www.omg.org. 200

[OMG97b] «UML Notation. Version 1.1, documento OMG ad/97-08-05», Septiembre 1997. Disponible en http://www.omg.org. 200

[OMG99] «Unified Modeling Language 1.3», 1999. Disponible en http://www.omg.org. 54

[OMG01] «Model Driven Architecture Specifications», Septiembre 2001. Disponible en http://www.omg.org/mda/specs.htm. 54, 236

[Parnas76] David L. Parnas. «On the Design and Development of Program Families». *IEEE Transactions on Software Engineering SE-2*, págs. 1–9, Marzo 1976. 93

[Pastor92a] Óscar Pastor. *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el Modelo Orientado a Objetos*. Tesis Doctoral, Universidad Politécnica de Valencia, Valencia, España, 1992. 54, 57

[Pastor92b] Óscar Pastor, F. Hayes y S. Bear. «OASIS: An OO Specification Language». *Proc. of CAiSE-92 (Conferencia), Lecture Notes in Computer Science*, tomo 593, págs. 348–363, 1992. 2, 57

[Pastor95] Óscar Pastor y Isidro Ramos. *OASIS 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*. Servicio de Publicaciones UPV, Valencia, España, 3 edición, Octubre 1995. 2, 54, 57, 59, 173, 174, 212, 327

[Pastor96] Óscar Pastor, Vicente Pelechano y B. Bonet. «An OO Methodological Approach for Making Automated Prototyping Feasible». En «Proc. of DEXA-96», págs. 18–25. Springer-Verlang, Septiembre 1996. 3

[Pastor97] Óscar Pastor, Emilio Insfrán, Vicente Pelechano, José Romero y José Merseguer. «OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods». *Proceedings of 9th International Conference, CAiSE97, Lecture Notes in Computer Science 1250*, págs. 145–159, Junio 1997. 3, 87

[Pastor00a] Óscar Pastor, Silvia Abrahão y Juan J. Fons. «An Object-Oriented Approach for Web-Solutions Modeling». En «Proceedings Media In Information Society (MEIS'00)», págs. 127–128. Ljubljana, Eslovenia, 2000. 50

[Pastor00b] Óscar Pastor, Pedro J. Molina y Alberto Aparicio. «Specifying Interface Properties in Object Oriented Conceptual

Models». En «Proceeding of ACM Working Conference on Advanced Visual Interfaces, AVI 2000», págs. 302–304. ACM Press, Mayo 2000. ISBN 1-58113-252-2. 295

[Pastor01] Óscar Pastor, Silvia Mara Abrahão, Juan Carlos Molina y Ismael Torres. «A FPA-like Measure for Object Oriented Systems from Conceptual Models». En «Current Trends in Software Measument», págs. 51–69. Shaker Verlag, Montreal, Canada, 2001. 280

[Paternò00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, Berlin, Alemania, 2000. 40, 119, 120, 122, 203, 347, 376

[Paternò02] Fabio Paternò. «CTTE. The ConcurTaskTree Environment», 2002. Disponible en http://giove.cnuce.cnr.it/ctte.html. 40, 211

[Paulk93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis y Charles V. Weber. «Capability Maturity Model for Software, Version 1.1». Informe técnico, Software Engineering Institute, Febrero 1993. CMU/SEI-93-TR-24, DTIC Number ADA263403. Displonible en: http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.024.html. 102

[Pelechano98] Vicente Pelechano, Emilio Insfrán y Óscar Pastor. «El Metamodelo OO-Method y su Repositorio Relacional». Informe técnico, Dpto. de Sistemas de Información y Computación, Universidad Politécnica de Valencia, 1998. 118, 200

[Pre02] «The Precise UML Group», 2002. Disponible en http://www.puml.org. 55

[Pree95] Wolfgang Pree. *Desing Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, MA, EE.UU., 1995. 85

[Pribeanu01] Costin Pribeanu, Jean Vanderdonckt y Quentin Limbourg. «Task Modelling for Context Sensitive User Interfaces». En Ch. Johnson (Editor), «Proceedings of 8th International Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'2001», págs. 49–68.

Springer Verlag, Berlin, Alemania, Octubre 2001. Publicado también en Lecture Notes in Computer Science, Vol. 2220. 257, 383

[Primo94] Eduardo Primo. *Introducción a la investigación científica y tecnológica*. Alianza Editorial, Madrid, España, 1994. ISBN 84-206-2789-5. 1

[Pro] «Bridge Point». Disponible en `http://www.projtech.com`. 109

[Puerta96] Angel R. Puerta. «The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development». En Jean Vanderdonckt (Editor), «Proceedings of 2nd Workshop on Computer-Aided Design of User Interfaces CADUI'1996», págs. 19–25. Presses Universitaires de Namur, Namur, Belgica, Junio 1996. 44, 212

[Puerta97a] Angel R. Puerta. «A Model-Based Interface Development Environment». *IEEE Software*, tomo 4(14), págs. 41–47, Julio/Agosto 1997. 44, 212

[Puerta97b] Angel R. Puerta y David Maulsby. «Management of interface design knowledge with MOBI-D». En «Proceedings of the 2nd international conference on Intelligent user interfaces, IUI'1997 (Orlando, Florida, EE.UU.)», págs. 249–252. ACM, ACM Press, Nueva York, EE.UU., 1997. ISBN 0-89791-839-8. 44

[Puerta97c] Angel R. Puerta y David Maulsby. «MOBI-D: A Model-Based Development Environment for User Centered Design (Atlanta, EE.UU.)». En «CHI'97», págs. 4–5. ACM SIGCHI, ACM Press, Nueva York, EE.UU., Marzo 1997. 44

[Puerta99] Angel R. Puerta y Jacob Einsenstein. «Towards a General Computational Framework for Model-Based Interface Development Systems». En «IUI'99 (Redondo Beach, California, EE.UU.)», págs. 171–178. ACM SIGCHI, ACM Press, Nueva York, EE.UU., 1999. ISBN 1-58113-098-8. 254

[Puerta02] Angel R. Puerta y Jacob Eisenstein. «XIML: A Common Representation for Interaction Data». En «Procs. of Inteligent User Interfaces 2002, IUI'2002», ACM

SIGCHI, ACM, Enero 2002. Disponible en http://www.iuiconf.org/02pdf/2002-002-0043.pdf. 257

[Rat02] «Rational Rose», 2002. Disponible en http://www.rational.com/products/rose/index.jsp. 11, 245

[Riehle96] Dirk Riehle y Heinz Zullighoven. «Understanding and Using Patterns in Software Develompment», 1996. Disponible en http://www.enteract.com/~bradapp/docs/patterns-intro.html. 80, 86

[Robbins97] J. Robbins, D. Hilbert y D. Redmiles. «ARGO: A Design Environment for Evolving Software Architectures». En «Proceedings of ICSE'97», págs. 600–601. ACM Press, Boston, MA., EE.UU., 1997. 12

[Roberts98] Dave Roberts, D. Berry, S. Isensee y J. Mullaly. *Designing for the User with OVID: Bridging User Interface Design and Software Engineering*. New Riders Publishing, Septiembre 1998. 35, 36, 375

[Rossi01] Gustavo Rossi, Daniel Schwabe y Robson Guimarães. «Designing personalized web applications». En «Proceedings of the tenth international conference on World Wide Web», págs. 275–284. SIGWEB/ACM, ACM Press, New York, EE.UU., 2001. ISBN 1-58113-348-0. 50

[Ruble97] D.A. Ruble. *Practical Analysis and Design for Client/Server and GUI Systems*. Yourdon Press Computing Series, 1997. 195

[SDL99] «ITU-T Recomendation Z.100. Languages for telecomunication applications: Specification and Description Language». Informe técnico, ITU-T, Ginebra, Suiza, 1999. 105, 252

[Selic94] B. Selic, G. Gullekson y P.T. Ward. *Real Time Object-Oriented Modeling*. John Wiley and Sons, Nueva York, EE.UU., 1994. 105, 252

[Sernadas87] A. Sernadas, C. Sernadas y H.D. Ehrich. «Object-Oriented Specification of Databases: An Algebraic Approach». En P.M.Stocker y W.Kent (Editores), «Proc VLDB 87», págs. 107–116. Morgan Kaufmann, 1987. 2

[Shaler97] S. Shaler y S.J. Mellor. «Recursive Design of an Application-Independent Architecture». *IEEE Software*, págs. 61–72, Enero 1997. 109

[Silva01] P. Pinheiro da Silva y N.W. Paton. «A UML-Based Design Environment for Interactive Applications». En «Proceedings of 2nd IEEE Workshop on User Interfaces for Data Intensive Systems UIDIS'01», págs. 60–71. IEEE Computer Society, Zurich, Suiza, Mayo 2001. 37

[Silva02] Paulo Pinheiro da Silva. «UMLi», 2002. Disponible en `http://www.cs.man.ac.uk/img/umli/index.html`. 36, 37, 38, 39, 375, 376

[Sánchez99] Juan Sánchez, Jose M. Barbera, Juan C. Molina y Pedro J. Molina. «Informe sobre interfaces y vistas». Nota interna 69, Proyecto OO-Method, CARE Technologies S.A., Junio 1999. 213, 214

[Sánchez01a] J. Ignacio Sánchez, Gustavo Santos y Pedro J. Molina. *HTML 4.*. Iniciación y Referencia. McGraw Hill Iberoamericana, Madrid, España, 2 edición, Septiembre 2001. ISBN 84-481-3168-1. 296

[Sánchez01b] Juan Sánchez, Juan J. Fons y Óscar Pastor. «From user requirements to user interfaces: a methodological approach». En «CAISE 2001», Interlaken, Suiza, Junio 2001. 50

[Software02] RedWhale Software. «XIML Forum», 2002. Disponible en `http://www.ximl.org`. 257

[Sys01] «System Architect», 2001. Disponible en `http://www.popkin.com/products/product_overview.htm`. 13, 245

[Tam98] R. Chung-Man Tam, David Maulsby y Angel R. Puerta. «UTEL: A tool for Eliciting User Task Models from Domain Experts». En «Proceedings of the 3nd international conference on Intelligent user interfaces, IUI'1998 (San Francisco, California, EE.UU.)», págs. 77–80. ACM, ACM Press, Nueva York, EE.UU., 1998. ISBN 0-89791-955-6. 44

[Tel02] «TeleLogic», 2002. Disponible en `http://www.telelogic.com`. 107

[Tidwell99] Jennifer Tidwell. «Common Ground: A Pattern Language for Human-Computer Interface Design», 1999. Disponible en http://www.mit.edu/~jtidwell/common_ground.html. 88, 117, 197

[Tog02] «Together», 2002. Disponible en http://www.togethersoft.com/. 11

[Trætteberg00] H. Trætteberg. «Model Based Design Patterns». Position paper, 2000. Workshop on User Interface Design Patterns, CHI'2000. 88, 90, 298

[Trætteberg02] H. Trætteberg. *Model-Based User Interface Design*. Tesis Doctoral, Norwegian Uiversity of Science and Technology, Trondheim, Noruega, Mayo 2002. Disponible en http://www.idi.ntnu.no/~hal/publications/thesis/thesis.pdf. 88, 90

[Vanderdonckt93] Jean Vanderdonckt y François Bodart. «Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection». En S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel y T. White (Editores), «ACM Proc. of the Conf. on Human Factors in Computing Systems INTERCHI'93», págs. 424–429. ACM, ACM Press, Amsterdam, Holanda, Abril 1993. 15, 16, 125, 254, 351

[Vanderdonckt99a] Jean Vanderdonckt. «Advice-Giving Systems for Selecting Interaction Objects». En «Proceedings of 1st IEEE Workshop on User Interfaces for Data Intensive Systems UIDIS'99», 1999. 23, 92

[Vanderdonckt99b] Jean Vanderdonckt. «Assisting Designers in Developing Interactive Bussiness Oriented Applications». En «Proc. of HCI International '99», 1999. 23, 92

[Vanderdonckt01] Jean Vanderdonckt y Cristelle Farenc (Editores). *Tools for Working with Guidelines*. Springer Verlag, Londres, Reino Unido, 2001. ISBN 1-85233-355-3. Proceedings of the Int. Workshop on Tools for Working with Guidelines, 2000 Annual Meeting of the Special Interest Group (Espace Bellevue, Biarritz, France, 7-8 octobre 2000). 73, 252

[Ver02] «Verilog», 2002. Disponible en http://verilog.com. 107

[VIS02] «VISIO», 2002. Disponible en http://www.microsoft.com/visio. 14, 195

[Vlissides96] John Vlissides. «Seven Habits of Successful Pattern Writers». *C++ Report*, Noviembre/Diciembre 1996. Disponible en http://www.research.ibm.com/designpatterns/pubs/7habits.html. 117

[W3C00] W3C. «Extensible Markup Language (XML) 1.0 (Second Edition)», Octubre 2000. W3C Recommendation. Disponible en http://www.w3.org/TR/2000/REC-xml-20001006. 242, 256, 307

[W3C01] W3C. «Extensible Stylesheet Language (XSL) Version 1.0», Octubre 2001. W3C Recommendation. Disponible en http://www.w3.org/TR/xsl/. 250

[Walrath99] Kathy Walrath y Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, Reading, MA, EE.UU., Julio 1999. ISBN 0201433214. 254

[Weber01] Alexandra Weber. «Executable Models Are Inevitable». *Software Development Magazine*, Septiembre 2001. Disponible en http://www.sdmagazine.com/documents/s=1123/sdm0109b/0109b.htm. 236, 291

[Weiss99] David M. Weiss y Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, EE.UU., 1999. ISBN 0-201-69438-7. 93, 94, 95, 97, 100, 376

[Welie00a] Martijn van Welie. «The Amsterdam Collection of Patterns in User Interface Design», 2000. Disponible en http://www.cs.vu.nl/~martijn/patterns/index.html. 88, 90, 117, 197, 298

[Welie00b] Martijn van Welie y H. Trætteberg. «Interaction patterns in user interfaces». En «7th. Pattern Languages of Programs Conference», Allerton Park Monticello, Illinois, EE.UU., Agosto 2000. 90, 298

[Welie00c] Martijn van Welie, Gerrit C. van der Veer y A. Eliëns. «Patterns as Tools for UI Design». En «International Workshop on Tools for Working with Guidelines», págs. 313–324. Biarritz, Francia, Octubre 2000. 298

[Welie02] Martijn van Welie, Kevin Mullet y Paul McInerney. «CHI2002 Patterns Workshop», 2002. Resumen disponible en http://www.welie.com/patterns/chi2002-workshop/. 88

[Winckler02] Marco Winckler, Philippe Palanque, Christelle Farenc y Marcelo S. Pimenta. «Task-Based Assessment of Web Navigation Design». En Costin Pribeanu y Jean Vanderdonckt (Editores), «TAMODIA'200. 1st International Workshop on TAsk, MOdels and DIAgrams for user interface design», págs. 161–169. Academy of Economic Studies, Bucarest, Rumania, Julio 2002. 57

[Winn02] Tiffany Winn y Paul Calder. «Is This a Pattern?» *IEEE Software*, págs. 59–65, Enero/Febrero 2002. 84

[Withey96] J. Withey. «Investment Analisys of Software Assets for Product Lines». Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnige Mellon University, Pittsburg, PA, Noviembre 1996. 101, 102

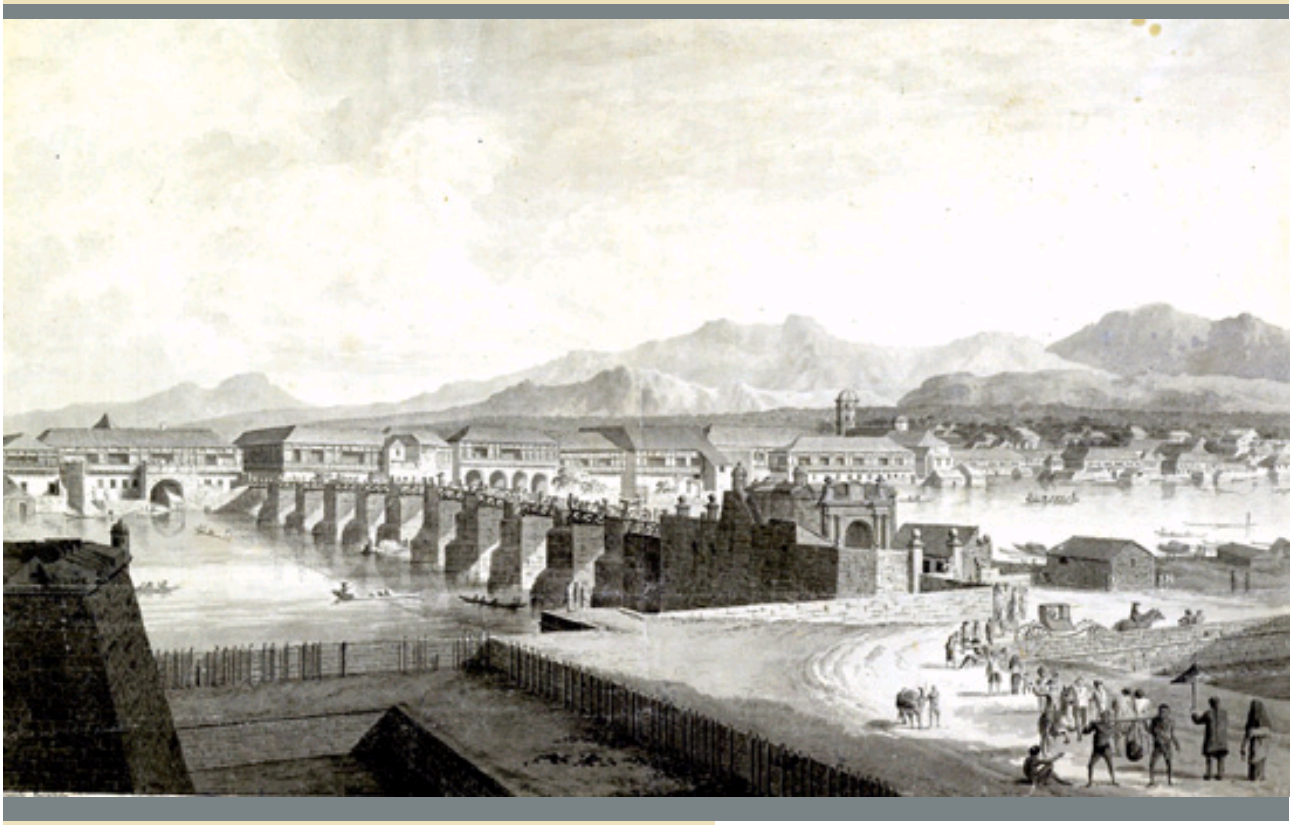
[Zloof77] M. Zloof. «Query-By-Exaple: A Data Base Language». *IBM System Journal*, tomo 4, págs. 324–343, 1977. 173

«No es preciso tener muchos libros,
sino tenerlos buenos. »

— Lucio A. Séneca, filósofo y pensador cordobés, 4 a.C. – 65.

Índice de figuras

Índice de tablas



UNIVERSIDAD
POLITECNICA
DE VALENCIA