UNIVERSIDAD AUTÓNOMA DE MADRID

Un Método para la Aplicación de Documentación Inteligente en la Instanciación de *Frameworks* Orientados a Objetos

TESIS DOCTORAL

Autor:

Alvaro Manuel Ortigosa

Directores:

Roberto Moriyón Salomón

Marcelo Ricardo Campo

Escuela Técnica Superior de Informática

Febrero de 2000

Memoria presentada para optar al grado de Doctor en Ingeniería Informática

Agradecimientos

Creo sinceramente que esta tesis merece una lista de agradecimientos tan larga como la tesis misma. Ha sido tanto el tiempo que he sufrido de *tesitis* aguda, muchas veces en estado de metástasis, que no es exagerado decir que sus efectos se han hecho sentir a ambos lados del Atlántico.

Humberto Eco sostiene que no corresponde agradecer a los directores. Pero en mi caso me encuentro con el dilema de que los directores hay sido mucho más amigos que otra cosa. Entonces es en este papel de amigos que les quiero dar mi más profundo agradecimiento.

Por eso empezaré por dar las gracias a Marcelo y Analía (que no me ha dirigido, pero me ha *sufrido* de igual forma) por... por todo. Y a Roberto, quien soy plenamente consciente me ha dedicado más esfuerzo y tiempo del que era posible. A los tres, muchas, muchas gracias por su paciencia.

Pero como en los últimos tres años, además de haber hecho la tesis, he vivido, y muy bien, en Madrid, hay mucha gente responsable de esto.

Así es que tengo que agradecer a las autoridades de la Facultad de Ciencias Exactas, y a las de la UNICEN en general, por haberme dado la oportunidad de realizar mi doctorado sin otra preocupación más que el doctorado mismo. En especial quiero dar las gracias a Roberto Gratton y Laura Elisondo, quienes una vez más me dieron su confianza y apoyaron mis estudios en el exterior. De la misma forma, quiero agradecer a Jane Pryor, quien dejó semanas (meses?) de su vida frente a una computadora, preparando el proyecto FOMEC que permitió mi estancia en Madrid. A los tres, muchas gracias.

Como he dicho, estos tres años en Madrid han sido una época muy buena de mi vida, y esto ha sido responsabilidad de las excelentes personas que me he encontrado en el grupo GHIA en particular (antiguos y nuevos integrantes) y en la escuela de Ingeniería Informática en general. Sin ningún orden en particular, muchas gracias a Pilar, Javier, Pablo, Paco, Manolo, Estrella, Fede, Juan, Alfonso, Miguel Angel, Ruth, Germán, Enrique, Pablo y todos los demás. Hay dos personas, integrantes morales del grupo, a las que también quiero agradecer. Una de ellas es Juana, por haber demostrado mucha paciencia y una excelente predisposición para ayudar en cualquier gestión burocrática; qué sería de la escuela sin ti? Y la segunda es Espe, quien siempre me abrió las puertas de su casa y a quien nunca pude resistir un segundo y un tercer plato de sus exquisitos manjares. A todos, gracias por ser verdaderos amigos y permitir que me sintiera (casi) un español más.

También son muy responsables los que tuvieron que hacerse cargo de mis cosas en Tandil durante estos tres años. Y aquí la palma se la lleva Claudia. Gracias, Pitty. Como así también gracias a todo el grupo GOV.

Uno que realmente ha sufrido de cerca mis buenos y malos momentos durante el doctorado, ha sido Jony. Gracias por no intentar nunca tirarme por el balcón, por alejarme de los pistachos y por grabar *Fraser* todas las semanas. Y no por más breve, menos sufrido, Marcelin; a ti también muchas gracias.

Finalmente quisiera agradecer aquello que no se puede. Tal vez podría nombrar la paciencia para entender que un domingo (todos ellos?) es un día perfectamente normal para trabajar en una tesis de doctorado, o que un 1 de enero no es razón suficiente para dejar de trabajar en aquel *paper*. O tal vez podría hablar del constante estímulo, apoyo y optimismo para seguir adelante. O del esfuerzo para que yo diera dos pasos seguidos al ritmo de la música. O... podría hablar de mil cosas más. Pero todo se resume en: gracias Rosemary por estar.

Agradecimientos

El espacio (y el tiempo) es muy corto para nombrar a todos los que quisiera, y poder decirle a cada uno todo lo que quisiera. Si alguien no ha sido nombrado y cree que debería estar aquí, pues que sepa que seguramente ha sido por olvido de mi parte y que, por favor, se sienta acreedor de mi gratitud. Para no extenderme indefinidamente, muchas gracias a tod@s l@s que, de una u otra manera, habéis contribuido para que esta tesis llegue a su fin.

Tabla de Contenidos

Tabla de Contenidos

Índice de Figuras

Resumen

Disponer de documentación de calidad es un requisito fundamental para el éxito del proceso de desarrollo de software. Los documentos fluyen a través de este proceso, comunicando decisiones tomadas durante las distintas etapas que lo componen. Más aún, la documentación desempeña un papel central en la reutilización de software, donde es necesario que cada componente de software a ser reutilizada sea bien comprendida.

Este aspecto cobra especial importancia en el caso de los *frameworks* de aplicaciones orientados a objetos. Estos *frameworks* constituyen un gran avance en la reutilización de software porque, más que la reutilización de código de componentes individuales, permiten reutilizar el diseño de sistemas o subsistemas. Sin embargo, dependiendo de la complejidad del *framework*, el desarrollo de nuevas aplicaciones es, generalmente, una tarea difícil y trabajosa, principalmente en el caso de usuarios inexpertos. Esta dificultad constituye el mayor obstáculo a una utilización más generalizada de esta tecnología.

Por este motivo, durante los últimos 10 años se ha invertido un considerable esfuerzo en la creación de técnicas de documentación más poderosas. Las técnicas tradicionales de documentación de diseño y código orientado a objetos no son suficientes para describir el complejo diseño de un *framework*, especialmente si se tienen en cuenta los distintos tipos de usuarios que pueden necesitar la documentación del mismo. Cuatro tipo de usuarios de *frameworks* han sido identificados: programadores de aplicaciones, encargados de mantenimiento del *framework*, diseñadores de otros *frameworks* y verificadores. Considerando esta variedad de tipos de usuarios, distintos métodos de documentación han sido específicamente propuestos para documentar *frameworks*. Algunos de ellos son informales y prescriptivos, es decir, describen cómo el *framework* debe ser utilizado. Otros métodos, en cambio, son más formales y descriptivos: describen el diseño del *framework* y el usuario debe deducir cómo utilizarlo. Cada método está orientado a un determinado tipo de usuario y, si bien algunos son capaces de describir adecuadamente algunos aspectos del *framework*, ninguno consigue satisfacer con éxito todos los requisitos de la documentación de *frameworks*. Esto es especialmente cierto desde el punto de vista de los programadores de aplicaciones.

Estas limitaciones llevan a pensar que resulta necesario complementar la documentación con herramientas que ayuden de forma efectiva al usuario en la instanciación de un *framework* determinado. En la comunidad de Ingeniería de Software cuenta cada día con mayor aceptación la idea de que es esencial la utilización de herramientas más inteligentes para conseguir una mayor eficiencia en el desarrollo de software, especialmente en términos de reducir los costes de desarrollo de software confiable. En esta dirección, entre las diferentes propuestas que se han realizado, destacan los llamados *libros de recetas interactivas (active cookbooks)*. Estos libros proporcionan al usuario una interfaz que le permite acceder a recetas que le proveen ayuda semiautomatizada para el proceso de instanciación. Las recetas no explican los motivos subyacentes a las soluciones que proponen, sino únicamente la forma en que el problema planteado se puede resolver mediante la utilización del *framework*. Este tipo de asistencia simplifica la instanciación de funcionalidad prevista *a priori*, ya que para una persona resulta sencillo seguir una lista de instrucciones detalladas paso a paso. Sin embargo, es precisamente su secuencialidad estricta el aspecto que representa una de las desventajas fundamentales de esta forma de abordar el problema. El usuario de un manual de recetas interactivas tiene que limitarse a seguir cada receta del principio al final hasta el último detalle, sin ningún grado de flexibilidad, o resignarse a no utilizar la herramienta.

En este sentido, la herramienta de asistencia ideal sería aquella que permita al usuario del *framework* describir la funcionalidad requerida y, en función de eso, genere automáticamente una aplicación que provea esa funcionalidad. Aunque actualmente no es posible ofrecer este tipo

de herramientas, sí es posible construir herramientas inteligentes que le indiquen al usuario qué debe hacer para implementar una determinada funcionalidad. De esta forma se potencia al máximo tanto la capacidad del ordenador para planificar y realizar actividades repetitivas y rutinarias como la capacidad de la persona para tomar decisiones en función de análisis más profundos y matizados.

En esta tesis se presenta *SmartBooks*, un método para documentar y asistir la instanciación de *frameworks*. El objetivo de *SmartBooks* es permitir la construcción de sistemas inteligentes de documentación y seguimiento del proceso de instanciación, que puedan guiar activamente el proceso de instanciación, pero que al mismo tiempo sean lo suficientemente flexibles para adaptarse a las distintas necesidades de los usuarios.

En consecuencia, *SmartBooks* se basa en la idea de ofrecer explicaciones procedurales sensibles al contexto, de acuerdo con la funcionalidad requerida para la nueva aplicación. Para lograr flexibilidad y adaptabilidad, *SmartBooks* propone la utilización de explicaciones generadas dinámicamente utilizando técnicas de planificación, en particular planificación con minimización de compromisos. Las explicaciones generadas, denominadas planes de instanciación, son estructuradas como una secuencia de actividades que el usuario debería ejecutar para implementar la aplicación. Esas actividades son representadas por tareas de instanciación, un concepto basado en los modelos de tareas de usuario.

El método *SmartBooks* propone la utilización de reglas de instanciación para documentar los *frameworks*, de forma tal de poder general planes de instanciación. Este trabajo describe la especificación de las reglas de instanciación, utilizando tanto notaciones gráficas como textuales. También presenta un algoritmo de planificación, denominado *PIT*, el cual ha sido desarrollado para satisfacer los requisitos del dominio de instanciación de *frameworks*. Además, se describe el diseño de módulos destinados a administrar y asistir al usuario en la ejecución de las tareas de instanciación. Finalmente, se presenta un prototipo de entorno de instanciación, desarrollado para probar el enfoque propuesto.

Palabras claves: Reutilización de *Frameworks* Orientados a Objetos, Planificación, Herramientas de Apoyo al Proceso de Desarrollo de Software.

Abstract

Good quality documentation is an essential requirement for a successful accomplishment of all the tasks involved in software development. Documents flow through the development process, and they communicate decisions involved in the different stages of the development. Moreover, documentation plays a central role in software reuse, where a developer must comprehend a piece of software to be reused in order to build a new application.

This aspect gets a critical relevance in the case of object-oriented application frameworks. Object-oriented application frameworks constitute a great improvement in software reuse because they promote the reuse not only of single building blocks, but also of the design of systems or subsystems. However, depending on its complexity, the development of new applications reusing a given framework usually is a hard and time-consuming task for novice users. This aspect represents one of the most limiting factors of the technology.

Due to this reason, much effort has been devoted during the last ten years to create more powerful documentation techniques. Traditional design and code documentation techniques are not enough to describe the complexity of a framework, specially if the different kinds of users that may need to access framework documentation are considered. Four kinds of framework (re)users have been identified: application developer, framework maintainer, developer of another framework and verifier. Taking into account this variety, different documentation methods have been specifically proposed for frameworks documentation. Some of them are informal and prescriptive, that is, they describe how the framework should be used. Some other methods are more formal and descriptive: they describe the framework design, and the user has to deduce how to use it. Every technique is oriented to a given kind of framework user, and although some of these approaches are able to provide good descriptions of some framework aspects, none of them can successfully satisfy all the framework documentation requirements. This is specially true in the case of application developers.

This limitation naturally leads to think on tools that effectively help the user in the instantiation of a given framework as an important complement. In this sense, the idea that more intelligent tools are essential to achieve higher efficiency in software development, especially regarding the reduction of costs for developing reliable software, has every day stronger support within the software engineering community. Among the different proposals existing in the literature, the so-called active cookbooks represent one of the most prominent examples of tools that provide semi-automated assistance to the framework instantiation process. Active cookbooks are able to enact recipe descriptions, providing the user an interactive interface that guides him/her through the instantiation process. Recipes do not explain the design rationale, they just explain how the problem can be solved using the framework. This kind of *help* facilitates the instantiation of predicted functionality because humans are good at following step-by-step directions, but, paradoxically, its little flexibility represents one of the fundamental drawbacks of the approach. When dealing with an active cookbook the user usually has to follow the embedded recipes up to the last detail, or resign to not using the tool at all.

Considering object oriented frameworks, the ideal supporting tool would allow the framework user to describe the required functionality and, following this description, the tool would automatically generate an application implementing that functionality. Although current technology does not allow to provide that kind of tools, it is possible to build intelligent tools that show the user what has to be done in order to implement a given functionality. In this way, the use of both the computer ability to build plans and to do repetitive tasks and the human ability for making decisions according to deeper analysis is optimized.

In this thesis *SmartBooks*, a method for documentation that is able to give assistance in the instantiation process of object oriented application frameworks, is introduced. The goal of

SmartBooks is to enable the construction of intelligent tools, which can actively guide the framework instantiation, being flexible enough to adapt themselves to the different requirements of the framework users.

SmartBooks is based on the idea of providing procedural explanations, according to the functionality required for the new application. In order to achieve flexibility and adaptability, *SmartBooks* proposes the dynamic generation of the explanations, using planning techniques, specially least commitment planning. The generated explanations, called instantiation plans, are structured as a sequence of activities that should be carried out by the user in order to implement the application. These activities are represented by instantiation tasks, a concept derived from the user task models.

The *SmartBooks* method proposes the use of instantiation rules to document the framework, in order to enable the generation of instantiation plans. This work describes the specification of instantiation rules, using graphical an textual notations. It also presents a planning algorithm, called *PIT*, developed for fulfill the requirements of the framework instantiation domain. Besides, the design of modules oriented to support and help on the execution of instantiation tasks is described. Finally, a prototype of instantiation environment, developed to test the approach, is presented.

Keywords: Application Framework Reuse, Planning, Software Development Support Tools

I Introducción

1. Motivaciones

Contar con documentación de buena calidad es un requisito esencial para poder llevar a cabo exitosamente todas las tareas involucradas en el desarrollo de software. Los documentos fluyen a través del proceso de desarrollo para comunicar decisiones tomadas en las diferentes etapas del proceso. En particular, la documentación desempeña un papel central en la reutilización de software, en la cual el usuario[1] debe comprender una componente de software a ser reutilizada para construir una nueva aplicación. Este aspecto toma relevancia crítica en el caso de los *frameworks* de aplicaciones orientados a objetos.

Los *frameworks* de aplicaciones orientados a objetos [Deu99, FS97, FSJ99, JF88, WBJ90] constituyen un notable avance en la reutilización de software, porque permiten representar y reutilizar diseños e implementaciones ya probados, ayudando a reducir el costo y mejorar la calidad del software [FSJ99]. Desde el punto de vista de su objetivo, un *framework* es una aplicación semicompleta, reutilizable, que puede ser especializada para construir aplicaciones que satisfagan requisitos específicos [JF88]. Esta especialización es denominada **instanciación** del *framework*.

Considerando la perspectiva estructural, un *framework* puede ser definido como el diseño reutilizable de un sistema, que describe cómo el sistema es descompuesto en un conjunto de componentes que interaccionan [Joh97]. Aquí el sistema puede ser tanto una aplicación completa como un subsistema. El *framework* describe cómo las responsabilidades del sistema son divididas entre sus objetos [JF88, WBJ90]. Esta división de responsabilidades es, según Deutsch [Deu99], el aporte más importante de un *framework*. Además de describir los objetos componentes, un *framework* también describe cómo estos objetos interaccionan, es decir, la interfaz de cada objeto y el flujo de control entre ellos.

Los *frameworks* también permiten la reutilización de implementación, pero esto es menos importante que la reutilización de las interfaces internas de un sistema y de la forma en que sus funciones están distribuidas entre las componentes. Este diseño de alto nivel es el principal contenido intelectual de un sistema de software, y los *frameworks* proveen una forma de reutilizarlo [FSJ99].

Una de las características más importantes de los *frameworks* es la *inversión de control* [FS97]. Cuando un programador reutiliza componentes de una biblioteca, debe implementar un programa principal, el cual invoca las componentes reutilizadas siempre que sea necesario. El programador decide cuándo invocar las componentes y es responsable de la estructura general y el flujo de control del programa. Al utilizar un *framework*, en cambio, el programa principal puede ser reutilizado y el programador decide qué componentes son ensambladas con este programa principal, pudiendo incluso desarrollar nuevas componentes para ensamblar. El código escrito por el programador es invocado por el código del *framework*, siendo el *framework* el que determina la estructura general y el flujo de control del programa.

En consecuencia, el principal beneficio derivado de la utilización de *frameworks* orientados a objetos es que permiten al programador de una aplicación concentrarse sólo en aquellos aspectos particulares de su aplicación. Instanciar un *framework* significa crear una especialización del mismo, en base a una descripción de esos aspectos específicos.

---

[1] Debe notarse que, en general, el *usuario* de un componente de software reutilizable es un programador que utiliza ese componente para crear una aplicación y no el usuario final de la aplicación. Excepto que se especifique explícitamente, a lo largo de todo este trabajo, éste será el significado atribuido al término.

La forma en que esta especialización tiene lugar depende de cada *framework*. Los *frameworks* son generalmente implementados como un conjunto de clases abstractas. Una clase abstracta es una clase que no tiene instancias, sino que es utilizada como un molde para la creación de subclases [WBJ90]. Los *frameworks* utilizan las clases abstractas porque definen la interfaz de sus componentes y además proveen un esqueleto que puede ser extendido para implementar las componentes específicas.

En consecuencia, en ciertos *frameworks* orientados a objetos crear una aplicación significa especializar clases abstractas para especificar los aspectos particulares de la aplicación. Otras veces, el *framework* provee también la implementación de las componentes concretas necesarias. Con estos *frameworks*, implementar una aplicación consiste en especificar qué componentes serán utilizadas y cómo serán conectadas. En el caso general, la actividad de especializar un *framework* para crear una aplicación puede ser vista como una combinación: definir nuevos componentes y reutilizar existentes, especificando la forma en que todas estas componentes son conectadas entre si.

2. El problema

La técnica ideal de reutilización es una componente que satisface exactamente las necesidades del desarrollador, y que puede ser reutilizada sin necesidad de realizar adaptaciones o aprender cómo utilizarla [FSJ99]. Sin embargo, una componente que satisface exactamente los requisitos de una aplicación, muy probablemente deba ser adaptada cuando los requisitos de la aplicación evolucionen o para ser utilizada en aplicaciones con requisitos ligeramente distintos. Cuando más adaptable sea una componente, más probable es que pueda ser utilizada para una situación en particular, pero también implica más trabajo usarla y aprender cómo usarla.

Este es el caso de los *frameworks*, que implementan estructuras adaptables a distintas necesidades: estructuras de diseño muy flexibles resultan en diseños altamente complejos y, por lo tanto, difíciles de comprender.

Un factor que aumenta aún más la dificultad de utilizar un *framework* es la inversión de control. La ventaja de poder reutilizar la estructura de control provista por el *framework*, tiene como contrapartida la necesidad de entender esta estructura de control, de forma de poder determinar el comportamiento esperado de los componentes especializados.

Por estos motivos, a pesar de las ventajas que ofrece el desarrollo de aplicaciones basadas en *frameworks* orientados a objetos, dependiendo de la complejidad del *framework*, a menudo la reutilización del mismo resulta compleja y excesivamente trabajosa para los usuarios sin experiencia [BD99, FS97, HHG90, Joh92, Sou99].

Por ejemplo, alcanzar un alto grado de productividad utilizando un *framework* para interfaces de usuario usualmente implica un tiempo que oscila entre los 6 y 12 meses [FS97]. Generalmente, esto significa que, en el caso de usuarios novatos, el desarrollo de una aplicación utilizando un *framework* toma más tiempo que el desarrollo de la misma aplicación partiendo desde cero. Esta larga curva de aprendizaje representa uno de las mayores deficiencias de la tecnología de *frameworks* orientados a objetos.

Debido a esto, durante los últimos 10 años se ha invertido mucho esfuerzo en la creación de técnicas de documentación más potentes, que permitan reducir el esfuerzo necesario para comprender un *framework*. Entre otras cosas, se observó que las técnicas tradicionales de documentación de diseño y código no son suficientes para describir la complejidad de un *framework*, especialmente si se considera los diferentes tipos de usuario que pueden necesitar acceder a la documentación [Joh92].

En este sentido, Butler y Dénommée [BD99] describen cuatro tipos de usuarios de *frameworks*: desarrolladores de aplicaciones, encargados del mantenimiento del *framework*, desarrolladores de otros *frameworks* y verificadores. Teniendo en cuenta esta variedad de

usuarios, diferentes métodos de documentación ha sido propuestos específicamente para documentar *frameworks*. Algunos de ellos son informales y prescriptivos, es decir, describen cómo el *framework* debe ser usado. Ejemplos de este tipo de documentación son los patrones propuestos por Johnson [Joh92] y los libros de recetas [Ado85, Pre95b]. Algunos otros métodos son más formales y descriptivos: describen el diseño y el usuario debe deducir cómo usar el *framework*. En esta categoría es posible clasificar los contratos de interfaz [Mey92], los contratos de interacción [HHG90], el método formal propuesto por Soundarajan [Sou99] para especificar el flujo de control del *framework* y la técnica para especificación formal de los patrones de diseño [LK98]. Existen otros métodos que poseen tanto características prescriptivas como descriptivas. Por ejemplo, los patrones de diseño [GHJV94] y metapatrones [Pre94] describen el diseño del *framework*, al tiempo que ofrecen una clara orientación sobre cómo está previsto que el *framework* sea especializado. Finalmente, también han sido propuestos métodos que combinan más de una técnica y que generalmente están diseñados para ser utilizados bajo el formato de libro electrónico [DHS98, GM95, LK94].

Cada técnica o método está orientado a un tipo específico de usuario del *framework* y, aunque algunos de estos enfoques son capaces de proveer buenas descripciones de algunos aspectos del *framework*, ninguno de ellos puede satisfacer con éxito todos los requisitos de la documentación de *frameworks*.

Estas limitaciones llevan a pensar que resulta necesario complementar la documentación con herramientas que ayuden de forma efectiva al usuario en la instanciación de un *framework* determinado. En esta dirección, entre las diferentes propuestas que se han realizado, destacan los llamados *libros de recetas interactivas (active cookbooks)* [SSP95, PPSS95]. Estos libros proporcionan al usuario una interfaz que le permite acceder a recetas que le proporcionan ayuda semiautomatizada para el proceso de instanciación. Las recetas no explican los motivos subyacentes a las soluciones que proponen, sino únicamente la forma en que el problema planteado se puede resolver mediante la utilización del *framework*. Este tipo de asistencia simplifica la instanciación de funcionalidad prevista a priori, ya que para una persona resulta sencillo seguir una lista de instrucciones detalladas paso a paso. Sin embargo, es precisamente su secuencialidad estricta el aspecto que representa una de las desventajas fundamentales de esta forma de abordar el problema. El usuario de un manual de recetas interactivas tiene que limitarse a seguir cada receta del principio al final hasta el último detalle, sin ningún grado de flexibilidad, o resignarse a no utilizar la herramienta.

3. La Tesis

En vista de estos problemas, es necesario considerar la utilización de herramientas más avanzadas, que permitan ofrecer una asistencia más acorde con las necesidades de los desarrolladores de software. En general, en la comunidad de Ingeniería de Software cuenta cada día con mayor aceptación la idea de que es esencial la utilización de herramientas más inteligentes para conseguir una mayor eficiencia en el desarrollo de software, especialmente en términos de reducir los costes de desarrollo de software fiable. Esta situación es especialmente relevante en lo que se refiere a las herramientas de apoyo a la reutilización de software [Bos98] y, en particular, en el caso de los *frameworks* orientados a objetos [OC98].

En este sentido, la herramienta de asistencia ideal sería aquella que permita al usuario del *framework* describir la funcionalidad requerida y, en función de eso, genere automáticamente una aplicación que provea esa funcionalidad. Aunque actualmente no es posible ofrecer este tipo herramientas, sí es posible construir herramientas inteligentes que le indiquen al usuario qué debe hacer para implementar una determinada funcionalidad. De esta forma se potencia al máximo tanto la capacidad del ordenador para planificar y realizar actividades repetitivas y rutinarias como la capacidad de la persona para tomar decisiones en función de análisis más profundos y matizados.

En esta tesis se presenta *SmartBooks*, un método para documentar y asistir la instanciación de *frameworks*. El objetivo de *SmartBooks* es permitir la construcción de sistemas inteligentes de documentación y seguimiento del proceso de instanciación, que puedan guiar activamente el proceso de instanciación, pero que al mismo tiempo sean lo suficientemente flexibles para adaptarse a las distintas necesidades de los usuarios.

En consecuencia, *SmartBooks* se basa en la idea de ofrecer explicaciones procedurales sensibles al contexto, de acuerdo con la funcionalidad requerida para la nueva aplicación [OCM99]. Para lograr flexibilidad y adaptabilidad, *SmartBooks* propone la utilización de explicaciones generadas dinámicamente utilizando técnicas de planificación, en particular planificación con minimización de compromisos [VS95, Wel99]. Las explicaciones generadas, denominadas planes de instanciación, son estructuradas como una secuencia de actividades que el usuario debería ejecutar para implementar la aplicación. Esas actividades son representadas por tareas de instanciación, un concepto basado en los modelos de tareas de usuario [JWMP93, Pat97, Pue97, SSC95].

En base a la combinación de técnicas de planificación con modelos de tareas de usuario, es posible construir una herramienta que permita al usuario del *framework* describir la funcionalidad que se requiere para las aplicaciones que serán implementadas; en función de esta funcionalidad, la herramienta puede generar los planes de instanciación, estructurados como una secuencia de las tareas de instanciación que deberían ser ejecutadas para implementar dicha funcionalidad.

Para proveer esta asistencia, es necesario que la documentación tradicional del *framework* sea extendida mediando el uso de reglas funcionales, que describan cómo las distintas funciones provistas por el *framework* pueden ser implementadas. Es decir, deben describir los pasos necesarios para especializar el *framework* de acuerdo a la funcionalidad requerida. En base a esta documentación, debe ser posible ofrecer al usuario la lista de opciones para incorporar a su aplicación, y de acuerdo a las opciones elegidas, generar el plan de instanciación correspondiente.

Para alcanzar mayor flexibilidad, la guía basada en la funcionalidad requerida es complementada con asistencia para mantener el software desarrollado consistente con los criterios de diseño utilizados en el desarrollo del *framework*. Esta información de diseño es codificada por medio de reglas de consistencia y administrada por un Administrador de Consistencia.

La Figura 1.1 resume la propuesta de *SmartBooks* para asistir en la instanciación de *frameworks*. En este modelo, el primer paso es documentar el *framework*. Para esto puede utilizarse cualquier técnica tradicional de documentación de diseños orientados a objetos, como por ejemplo *UML* [RJB99], *OMT* [RBP+91] o *OOSE* [JCJO92]. A esta documentación tradicional deben añadirse las reglas de instanciación, es decir, las reglas funcionales y de consistencia. En base a esta información, dos serán las componentes encargadas de brindar asistencia al usuario del *framework*. El Planificador, utilizando la descripción de funcionalidad requerida para la nueva aplicación y la documentación extendida, puede generar la secuencia de tareas de instanciación necesarias para implementar la aplicación. El Administrador de Consistencia, por su parte, utilizará las reglas para orientar, de acuerdo al diseño del *framework*, la ejecución de las tareas de instanciación, tanto las previstas en el plan de instanciación como las que ejecute el usuario por iniciativa propia.

Además de proponer un método de documentación y asistencia a la instanciación de *frameworks*, esta tesis realiza otros aportes en el mismo contexto. En primer lugar, se presenta un algoritmo de planificación, llamado *PIT* [OC00], el cual ha sido específicamente diseñado para ser utilizado en la generación de planes de instanciación. Este algoritmo parte de algoritmos propuestos en la literatura y los adapta a las necesidades de la instanciación de *frameworks*. Junto con el algoritmo *PIT*, se ha diseñado una representación para las reglas de

instanciación, de forma que las mismas puedan ser utilizadas por el algoritmo. Las reglas representadas de esta forma son denominadas acciones de instanciación.

Otro aporte presentado en este trabajo es un modelo de tareas de instanciación, creado con el objetivo de representar los planes de instanciación. Como se ha dicho, este modelo está basado en los modelos de tareas de usuario utilizados en las interfaces de usuario, y se complementa con un Administrador de Tareas, utilizado para administrar su ejecución.

Para facilitar la documentación del *framework*, también se ha diseñado una notación gráfica para representar las reglas de instanciación. Esta notación, denominada *TOON*, permite describir la implementación de la funcionalidad provista por el *framework* en términos de componentes de software y tareas de instanciación. Por otro lado, también permite establecer condiciones que, a modo de invariantes, deben ser mantenidas por el software desarrollado.

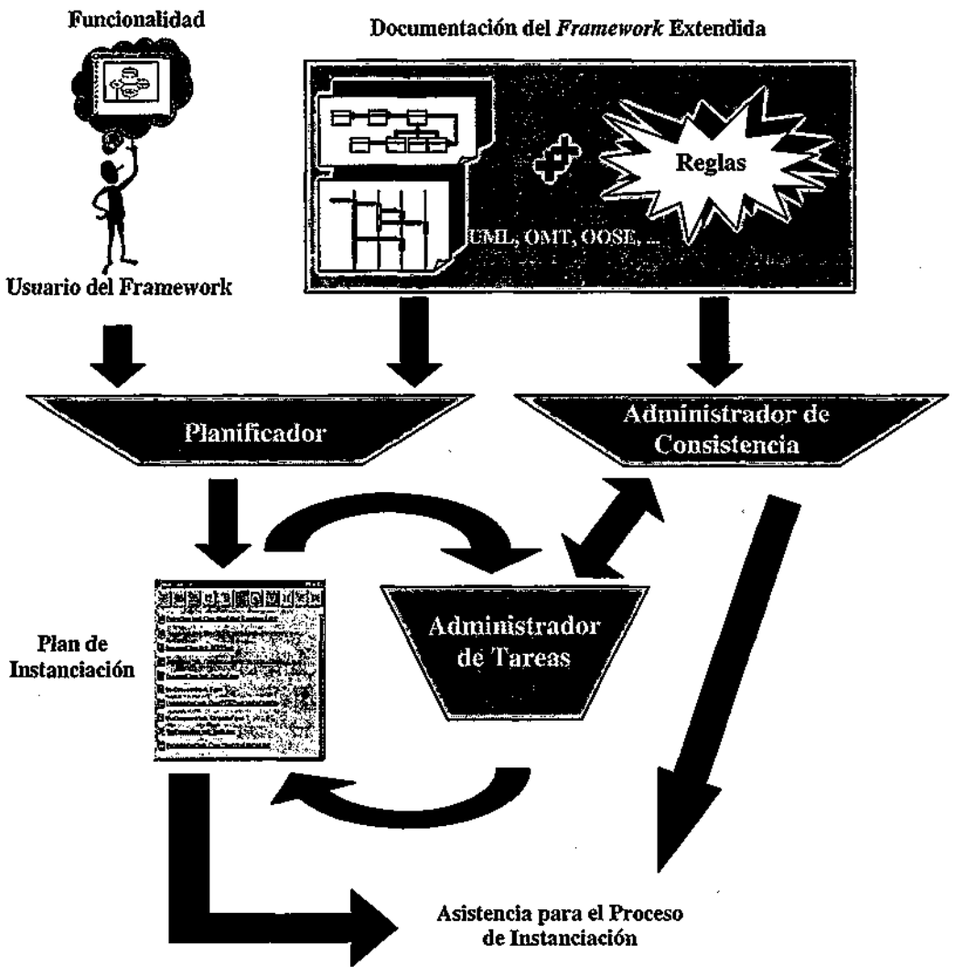


Figura 1.1 Estructura general propuesta por el método *SmartBooks*

Esta última característica es utilizada para la definición de las reglas de consistencia. Otro aporte de esta tesis es el diseño de un Administrador de Consistencia que, interaccionando con el Administrador de Tareas y en base a las reglas de consistencia, dirige la ejecución de las tareas de instanciación.

Finalmente, en este trabajo también se presenta un prototipo de entorno de documentación e instanciación de *frameworks* llamado *HiFi* [OC99], el cual ha sido diseñado y construido en base al método *SmartBooks*.

4. Organización del Trabajo

El resto de esta tesis está organizada en nueve capítulos y cuatro anexos, cuyos contenidos se describen a continuación.

El próximo capítulo introduce la reutilización de software orientado a objetos y en especial el concepto de *framework* de aplicaciones orientado a objetos. Para esto comienza con una explicación de los mecanismos en los que se basa la reutilización en la orientación a objetos y se definen los *frameworks* orientados a objetos, comparándolos con otras formas de reutilización. Luego se clasifica a los *frameworks*, mostrando ejemplos de cada tipo. Aquí se introduce el *framework* para editores gráficos *HotDraw*, el cual es utilizado como ejemplo en el resto de la tesis. Finalmente, se analizan las ventajas derivadas de la utilización de *frameworks* y los problemas relacionados con los mismos.

En el capítulo III se analiza la problemática de la comprensión de *frameworks* y las técnicas actuales para su resolución. En primer lugar, se explican los factores que complican el entendimiento de un *framework*, así como las distintas necesidades de los usuarios, derivadas de las diversas funciones que pueden desempeñar en un ciclo de desarrollo basado en *frameworks*. A continuación, se realiza una descripción de los principales enfoques, basados en documentación, propuestos en la literatura para posibilitar la comprensión de *frameworks*, finalizando con un análisis de carencias de las técnicas descritas. Luego se analizan los enfoques basados en herramientas para asistir la instanciación de *frameworks*, comenzando con herramientas orientadas a *frameworks* y/o dominios específicos, y finalizando con constructores genéricos de aplicaciones. Finalmente, las limitaciones de estos enfoques son analizadas, concluyendo con un estudio de los requisitos de técnicas de documentación y herramientas destinadas a apoyar el proceso de instanciación de *frameworks*.

El capítulo IV presenta el método *SmartBooks*. Con este objetivo, el capítulo comienza con un ejemplo concreto de las necesidades de un creador de aplicaciones basadas en *frameworks* orientados a objetos, explicándose el tipo de asistencia que se pretende proveer para satisfacer esas necesidades. A continuación, se realiza un análisis de los requisitos necesarios para proveer esa asistencia, introduciéndose los conceptos básicos en los que se fundamenta la propuesta de *SmartBooks*: técnicas de planificación y modelos de tareas de usuario. Finalmente, la propuesta es desarrollada, mostrando cómo las técnicas de planificación son utilizadas para generar los planes de instanciación, y el concepto de tarea de instanciación es utilizado para estructurar dichos planes. También se explica el tipo de documentación que debe proveer el diseñador del *framework* para posibilitar este tipo de asistencia, en forma de reglas de instanciación. El capítulo finaliza con una descripción sobre la forma en que esta información puede ser utilizada en un entorno de instanciación de *frameworks*.

Los dos capítulos siguientes se centran en explicar los principales mecanismos desarrollados para aplicar el método *SmartBooks*. El capítulo V presenta los fundamentos de las técnicas de planificación, explicando algoritmos de uso general. En particular, se describe el algoritmo *UCPOP*, cuya utilización de la técnica de minimización de compromisos lo convierte en especialmente adecuado para su utilización en el dominio de instanciación de *frameworks*. A partir de este algoritmo, se analizan las extensiones necesarias para utilizar la técnica de planificación en el contexto de *SmartBooks*. En función de este análisis, se presenta *PIT*, un algoritmo diseñado en base a *UCPOP* con el objetivo de satisfacer los requisitos específicos de *SmartBooks* [OC99]. El capítulo finaliza con un ejemplo del uso de *PIT* en la generación de planes de instanciación.

El capítulo VI, por su parte, describe las técnicas diseñadas para complementar la propuesta de *SmartBooks* y facilitar su puesta en práctica. En primer lugar se explica el modelo de tareas de usuario utilizado para representar los planes de instanciación, así como el Administrador de Tareas diseñado para su ejecución. Luego se presenta la notación gráfica *TOON*, creada para la definición de las reglas de instanciación. Finalmente, se explica el diseño

de un Administrador de Consistencia basado en las reglas de consistencia presentadas en el capítulo IV.

El capítulo VII presenta un ejemplo de utilización del método *SmartBooks* para asistir en la instanciación de aplicaciones, concretamente en la creación de una aplicación a partir del *framework HotDraw*.

En el capítulo VIII se describe el diseño e implementación del prototipo de un entorno de instanciación basado en el método *SmartBooks*.

Finalmente, el capítulo IX presenta las conclusiones de este trabajo, explicando los aportes introducidos. También se realiza una descripción de las limitaciones de la propuesta, así como posibles líneas de investigación que pueden derivarse de la misma. El capítulo concluye con algunas consideraciones finales sobre la utilización de herramientas inteligentes en el proceso de desarrollo de software.

II Frameworks Orientados a Objetos

En este capítulo se presentan los *Frameworks* de Aplicaciones Orientados a Objetos. En primer término se introducen los principios de reutilización en la orientación a objetos, explicando los conceptos básicos sobre los cuales se fundamenta. Luego se presenta el concepto de *frameworks*, explicando sus tipos y características, describiendo finalmente el proceso de creación de aplicaciones a partir de *frameworks*, denominado proceso de instanciación. Sobre la base de los conceptos aquí introducidos, en los próximos capítulos se efectuará un análisis de las actuales técnicas de documentación y herramientas de apoyo a la instanciación, así como también se describirán propiedades deseables de estas herramientas. Finalmente, se presentará una propuesta para la documentación e instanciación de *frameworks*.

1. Mecanismos de Reutilización

Antes de describir los mecanismos de reutilización en la orientación a objetos, se introducirán los términos y conceptos utilizados para su implementación. Las definiciones aquí presentadas corresponden a las encontradas habitualmente en la literatura del tema, si bien la aplicación de algunos conceptos puede ser dependiente del lenguaje de programación utilizado.

Un **objeto** agrupa tanto datos como procedimientos que operan sobre esos datos. Los procedimientos son llamados generalmente operaciones: una **operación** es la definición de un servicio que puede ser solicitado a un objeto para modificar su comportamiento. Un cliente produce la ejecución de una operación cuando le envía un **mensaje** al objeto [GHJV94]. Un **método** es la implementación de una operación [JBR99].

Cada operación declarada por un objeto especifica el nombre de la operación, los objetos que toma como parámetros y el retorno de la operación. Esto se denomina la **signatura** de la operación. El conjunto de todas las signaturas definidas por las operaciones de un objeto se denomina la **interfaz** del objeto [GHJV94]. La interfaz de un objeto describe el conjunto completo de mensajes que pueden ser enviados a ese objeto. En otras palabras, describe el comportamiento de los objetos, sin especificar su implementación o estado [RJB99]. Las interfaces son un concepto fundamental en la orientación a objetos, porque los objetos son conocidos sólo a través de sus interfaces [GHJV94, WBJ90]. A esta propiedad se la denomina **encapsulamiento**.

Un **tipo** es un nombre utilizado para hacer referencia a una interfaz [GHJV94]. Por ejemplo, es posible decir que un objeto tiene el tipo *Ventana* si acepta todos los mensajes para operaciones definidas en la interfaz *Ventana*. De acuerdo a esta definición, un objeto puede tener varios tipos y objetos diferentes pueden compartir un mismo tipo. Distintas partes de la interfaz de un objeto pueden ser caracterizadas por distintos tipos y dos objetos del mismo tipo sólo necesitan compartir parte de sus interfaces. Un tipo es **subtipo** de otro si su interfaz contiene a la interfaz del otro, llamado **supertipo**. En este caso se dice que el subtipo **hereda** la interfaz de su supertipo.

Cuando un mensaje particular es enviado a un objeto, la operación concreta que se ejecuta depende tanto del mensaje como del objeto receptor. Distintos objetos que entienden idénticos mensajes pueden tener distintas implementaciones de las operaciones para responder al mensaje. La asociación en tiempo de ejecución entre un mensaje enviado a un objeto y la operación correspondiente es denominada **acoplamiento dinámico** (*dynamic binding*).

El uso de acoplamiento dinámico produce que el envío de un mensaje no se relacione con una implementación particular hasta el momento de la ejecución. Esto permite construir programas que presuman la presencia de un objeto con una interfaz particular, sabiendo que

cualquier objeto que implemente la interfaz correcta será capaz de responder los mensajes definidos en esa interfaz. Más aún, el acoplamiento dinámico permite que objetos con interfaces idénticas puedan ser intercambiados unos por otros en tiempo de ejecución, a los efectos de, por ejemplo, pasarlos como argumentos o enviarles un mensaje. Esta capacidad de sustituir objetos es conocida como polimorfismo [GHJV94] y es uno de los conceptos claves de la orientación a objetos.

Para ofrecer polimorfismo puro, es necesario que el lenguaje de programación provea la capacidad de que los elementos utilizados para hacer referencias a objetos (variables, parámetros, etc.) puedan contener, en distintos momentos, referencias a objetos de distinto tipo. Esto es lo que se conoce como referencia polimórfica, y es una propiedad que no todos los lenguajes de programación orientados a objetos poseen. En algunos casos, los lenguajes proveen formas restringidas de polimorfismo, como son la sobrecarga de operadores (es decir, distintos métodos tienen el mismo nombre, y el método efectivamente utilizado para responder un mensaje depende del tipo del receptor y de los argumentos) y los *templates* de C++. En otros casos, como el lenguaje *Smalltalk* [Gol83] por ejemplo, *sí* proveen polimorfismo puro.

Por ejemplo, supóngase una estructura de datos que implementa una lista y que posee un método para ordenar sus componentes. El código de ese método podría tener la siguiente estructura:

ordenar

repetir hasta que todo esté ordenado:
para cada elemento en la posición i, comparar con el i+1.
Si elemento[i] > elemento [i+1], invertir posiciones.

El polimorfismo puro se obtiene cuando el mismo código (escrito en un lenguaje como C++ o *Smalltalk*) funciona con más de un tipo de objeto almacenado en la lista. La comparación entre los elementos almacenados se realiza, generalmente, enviando un mensaje a uno de los elementos. Este mensaje sí se implementa por sobrecarga, porque el método utilizado para responderlo depende del tipo de los objetos.

El polimorfismo permite que un cliente sólo necesite suponer que los otros objetos implementan una interfaz específica, posibilitando una simplificación en la definición de clientes, que los objetos se desacoplen unos de otros y variar las relaciones de unos con otros en tiempo de ejecución. En el ejemplo anterior, el método *ordenar* no depende de los tipos de objetos que pueda contener la lista, en tanto todos entiendan el mensaje de comparación.

La implementación de un objeto es definida por su **clase**. La clase especifica los datos internos del objeto y su representación; también especifica las operaciones que el objeto puede ejecutar, a través de la definición de métodos. Desde un punto de vista más abstracto, una clase es la descripción de un conjunto de objetos que comparten los mismos atributos (o estructura), comportamiento, relaciones y semántica [JBR99].

Clases nuevas pueden ser definidas en función de clases existentes utilizando **herencia** de clases. Herencia es un mecanismo a través del cual elementos más específicos incorporan la estructura y el comportamiento de elementos más generales [JBR99]. Cuando una **subclase** hereda de una clase padre o **superclase**, incluye las definiciones de todos los datos y operaciones definidos por la superclase. Transitivamente, la subclase también hereda los datos y operaciones heredados por la superclase. De esta forma, los datos contenidos por una instancia de la subclase y los mensajes a los que pueda responder serán aquellos definidos en la subclase, en la superclase o heredados por la superclase.

Debe notarse que utilizando herencia de clases, la subclase hereda tanto la interfaz de la superclase como su implementación. Esto determina que una subclase sea un subtipo del tipo definido por la superclase y por lo tanto las instancias de la subclase pueden ser utilizadas en cualquier lugar donde se asuma la presencia de una instancia de la superclase.

Del mismo modo, a través del mecanismo de herencia es posible adaptar selectivamente el comportamiento de algunos servicios, redefiniendo su implementación en subclases. Esto hace posible reutilizar y adaptar la implementación de un componente, manteniendo la interfaz invariable.

Dependiendo del lenguaje de programación, los conceptos de tipo y clase varían. Algunos lenguajes, como por ejemplo *Java*, proveen mecanismos para diferenciar explícitamente entre interfaz y clase, lo cual permite expresar en forma independiente el tipo del objeto y su implementación. En lenguajes con verificación de tipos, como por ejemplo *C++* o *Eiffel*, la clase define tanto el tipo como la implementación del objeto. En lenguajes sin tipos, como *Smalltalk*, la clase define sólo la implementación de un objeto. En estas condiciones un objeto puede pertenecer a varios tipos (conceptuales) diferentes, así como un objeto puede ser substituido por otro cualquiera siempre que la clase de este último implemente la misma interfaz.

1.1. Reutilización por composición y por herencia de clases

Los mecanismos de reutilización de interfaz e implementación discutidos en la sección precedente habilitan dos formas alternativas de reutilización en sistemas orientados a objetos: herencia de clases y composición de instancias.

El mecanismo de herencia habilita la construcción de nuevos componentes a través de subclases que heredan la implementación de la estructura y operaciones definidas en las superclases. Las clases derivadas representan entidades cada vez más especializadas, basadas en clases más generales o abstractas. Este tipo de reutilización es también denominado ***caja blanca,*** pues las subclases tienen acceso a la implementación heredada.

El polimorfismo y el acoplamiento dinámico, por otro lado, permiten la reutilización por composición de instancias. Este tipo de reutilización coloca el énfasis en la distribución de funcionalidad básica entre diferentes objetos, los cuales son combinados o compuestos para obtener diferentes comportamientos. Este tipo de reutilización es denominado ***caja negra,*** pues los diferentes objetos pueden ser reutilizados a través de su interfaz, es decir, no es necesario que los clientes externos conozcan la estructura y la implementación de los servicios de cada componente.

Cada forma de reutilización posee características propias, que la hacen más adecuada para utilizar en determinadas situaciones. La reutilización basada en herencia de implementación permite construir con poco esfuerzo nuevos componentes y agregar nueva funcionalidad a través de subclases, así como modificar la funcionalidad implementada previamente. Estos componentes comparten la mayor parte de la funcionalidad, lo cual permite propagar automáticamente a las subclases los cambios realizados en las superclases. Esta ventaja, no obstante, también representa una desventaja. En el caso general, las subclases heredan la estructura interna de las superclases, lo que implica la necesidad de modificar manualmente las subclases cuando un cambio en la estructura de la superclase es realizado. Otra restricción que presentan estas implementaciones es la adaptación de comportamiento en tiempo de ejecución. En la reutilización basada exclusivamente en herencia, los métodos a ser aplicados son determinados estáticamente, inhibiendo la posibilidad de utilizar implementaciones alternativas de un mismo servicio.

La reutilización basada en composición, por su parte, ofrece la ventaja de soportar la adaptación dinámica de aplicaciones. Bajo esta estrategia, las clases implementan funciones muy específicas, las cuales son utilizadas a través de una interfaz bien definida. Comportamientos complejos son creados a través de la composición de diferentes objetos, en este caso cada uno de ellos mantiene referencias a uno o varios objetos. A través de estas referencias un objeto puede delegar la responsabilidad de ejecutar un servicio dado a otro

objeto. El objeto que efectivamente realizará la función depende de la configuración de instancias creada. Así, por ejemplo, diferentes objetos pueden implementar diferentes algoritmos de ordenamiento, los cuales pueden ser pasados como parámetros a los objetos que serán sus clientes.

La reutilización a través de la composición de objetos es más deseable que la reutilización basada en herencia, principalmente porque es más sencilla [FSJ99]. Aquí no es necesario modificar el software existente o crear nuevas subclases. El programador sólo debe saber cómo los objetos pueden ser conectados entre si, sin tener que conocer las especificaciones exactas de esos objetos. En el caso general, sin embargo, el conjunto de componentes desarrollado no es suficiente para implementar la funcionalidad requerida por una aplicación. En estos casos, es necesario desarrollar nuevos componentes que provean operaciones no disponibles.

Por ejemplo, en una aplicación de gestión, la persistencia puede ser implementada a través de componentes específicos [Cah99]. De esta forma, cambiando estos componentes, es posible modificar el método de almacenamiento, sin que sea necesario ningún cambio en el resto de la aplicación. Sin embargo, es posible que en algún momento sea necesario utilizar un sistema de ficheros que no ha sido previsto en los componentes existentes y por lo tanto un nuevo componente deba ser programado. Para esto, la herencia sirve como el complemento que permite la extensión de funcionalidad para la construcción de nuevos componentes, aprovechando la funcionalidad provista por los componentes existentes. En este ejemplo, sería posible especializar algunos de los métodos de almacenamiento existentes para implementar la nueva funcionalidad requerida. De este modo, composición y herencia son técnicas complementarias que, cuando son utilizadas apropiadamente, contribuyen para el desarrollo de sistemas altamente flexibles y reutilizables.

1.2. Clases Abstractas

Independientemente de las capacidades provistas por un lenguaje de programación, el concepto de herencia de interfaz puede ser representado a través de clases abstractas. Una **clase abstracta** es una clase en la cual por lo menos un método no está implementado. Por esta razón, una clase abstracta nunca tendrá instancias y será utilizada sólo como una superclase. Las clases que no son abstractas son denominadas **concretas**.

Una clase abstracta es diseñada para ser utilizada como un *molde* para especificar subclases, mientras que una clase concreta es diseñada para especificar objetos. Una clase abstracta define el conjunto de servicios o *protocolo* al cual responderán todos los objetos pertenecientes a sus subclases, a través de **métodos abstractos**.

Un método abstracto define la signatura de una operación, pero no provee una implementación. Deben ser redefinidos en subclases para implementar variantes específicas del comportamiento esperado del método. Los lenguajes de programación proveen diferentes sintaxis para definir estos métodos. Por ejemplo, *Eiffel* provee la calificación ***deferred*** para diferenciarlos. *C++* permite la definición de un método como ***pure virtual*** para indicar que es un método no implementado por la clase, mientras que en *Java* el mismo efecto se obtiene declarando el método como ***abstract***. *Smalltalk*, por su parte, no provee ninguna notación especial, siendo estos métodos definidos por el mensaje ***self subclassResponsibility***, el cual indica que la implementación del método tiene que ser provista por una subclase. En este caso, si alguna subclase no redefine alguno de estos métodos y un cliente lo invoca, se producirá un error de ejecución informando que ese método no está implementado.

Cuando todos los métodos definidos por una clase son abstractos, esa clase define efectivamente un *tipo* o *interfaz*. La clase no necesariamente determina la representación de la estructura interna de los objetos, sino sólo el comportamiento definido por los servicios

representados por los métodos abstractos. Así, diferentes subclases pueden definir diferentes implementaciones tanto para la representación de la abstracción cuanto para la implementación de los servicios. El polimorfismo permite que los clientes definan variables que hacen referencia a este tipo de objetos como del tipo definido por la clase abstracta, evitando la dependencia con subclases concretas específicas. Esto permite aumentar la reutilización de los clientes, como también la flexibilidad de la solución, pues variables definidas de un tipo *A* podrán referenciar objetos de cualquier subtipo de *A*.

Un ejemplo de clase abstracta es la clase *Figure* del *framework HotDraw*[1] [Joh92], que define el protocolo común de todos los objetos que serán manipulados en un editor; parte del código de esta clase es mostrado en la Figura 2.1. En este ejemplo, la clase *Figure* posee, entre otros, cuatro métodos abstractos, *origin*; *extent*; *displayOn*: y *translatedBy*:. El método *origin* retorna un punto representando el extremo superior izquierdo del área de dibujo de la figura. El método *extent* retorna un punto que representa el ancho y alto del área de dibujo de la figura. El método *displayOn:* muestra la figura en la pantalla. Finalmente, el método *translatedBy*: mueve la figura una distancia especificada por el argumento.

```
VisualPart subclass: #Figure

    extent
            ^self subclassResponsibility

    origin
            ^self subclassResponsibility

    displayOn: aGraphicsContext
            ^self subclassResponsibility

    translateBy: aPoint
            ^self subclassResponsibility
```

Figura 2.1 Definición de los métodos abstractos de la clase *Figure*.

La implementación de los cuatro métodos depende de las subclases específicas de *Figure*. Esta clase es la superclase de *CompositeFigure*, *PolylineFigure* y *TextFigure* (Figura 2.2). Por ejemplo, una *PolylineFigure* podría calcular el *origin* y el *extent* basándose en una colección de puntos que definen los vértices de una figura poligonal; una *TextFigure*, por su parte, podría calcularlos basándose en el tipo de la fuente utilizado y la cantidad de caracteres involucrados.

Los métodos *origin, extent, displayOn:* y *translatedBy:* se definen como métodos abstractos de la clase *Figure*. Esto significa que la implementación de tales métodos es responsabilidad de las subclases.

En algunos casos, una clase abstracta puede proveer directamente la implementación de un método, el cual no debería ser redefinido por subclases porque su comportamiento no admite variantes. Este tipo de métodos es denominado métodos ***base***, pues proveen una implementación completa de un comportamiento común a cualquier aplicación del dominio de

[1] La mayor parte de los ejemplos mostrados a lo largo de todo el trabajo están basados en el *framework HotDraw*, para la creación de editores gráficos. Así mismo, el lenguaje utilizado para representar los ejemplos será *Smalltalk* [Gol83] por ser el lenguaje que se utilizó para implementar el prototipo de la herramienta descrita en este trabajo.
Si bien el concepto de framework todavía no ha sido introducido, para analizar el siguiente ejemplo es suficiente considerar que *HotDraw* es una biblioteca de clases que pueden ser utilizadas para crear editores gráficos.

aplicación. En forma más general, son denominados **bases** todos aquellos métodos cuyo comportamiento no depende de otros métodos y que no está previsto que sean especializados, pudiendo estar definidos tanto en clases abstractas como concretas.

1.2.1. Reutilización de algoritmos: Métodos *Template*

Las clases abstractas representan un concepto de gran importancia desde el punto de vista de la reutilización de componentes. Respecto de la semántica del dominio de aplicación, una clase abstracta representa una abstracción que define el comportamiento genérico de un tipo de componente de ese dominio. Bajo esta interpretación una clase abstracta define el comportamiento común de un conjunto de componentes, los cuales se diferencian en el comportamiento específico implementado para los métodos abstractos, utilizando la misma interfaz.

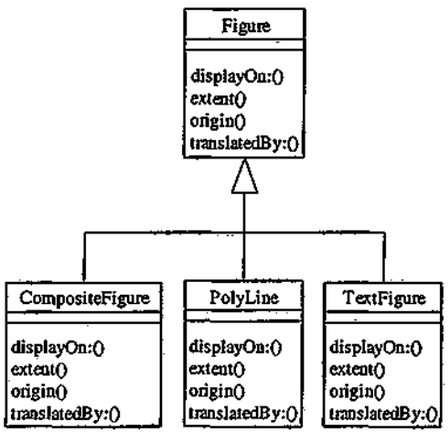


Figura 2.2 Ejemplo de una Jerarquía de Herencia

Volviendo al ejemplo de *HotDraw*, como se ha indicado más arriba, la clase *Figure* define, en primer lugar, las operaciones aplicables a cualquier objeto a ser visualizado y editado, pero no su representación específica. Esta representación dependerá del tipo específico de figura, como por ejemplo, *CompositeFigure*, *PolylineFigure* o *TextFigure*. También la clase provee la implementación de aquellos algoritmos que son independientes de la representación, pero que pueden depender de la implementación específica de los métodos abstractos. Estos algoritmos son denominados **métodos *template***, pues definen una estructura de control que invoca operaciones cuya implementación es provista por subclases u otros objetos. El comportamiento final de un método *template* varía en función de la implementación de los métodos abstractos que este invoca, de acuerdo con la clase de la instancia que recibe un mensaje que activa ese método *template*[2].

Por ejemplo, *Figure* provee la implementación de las operaciones para manipular un objeto gráfico, que pueden ser definidas en términos de las operaciones abstractas, *extent*; *origin*; *displayOn:* y *translatedBy:*.

El método *displayBox*, que retorna un área rectangular representando la región de la pantalla en la cual es mostrada la figura, puede implementarse en la clase *Figure* utilizando los puntos retornados por *origin* y *extent*. Así, *displayBox* se define como un método *template*. Este método especifica un comportamiento genérico que se reutiliza en todas las subclases de *Figure* como consecuencia de la herencia. Pueden agregarse nuevas subclases, como por

[2] Una definición más general de los métodos *template* es presentada al introducir los métodos *hooks* en la próxima sección

ejemplo elipses, sin provocar cambios en el sistema, siempre que estas nuevas subclases provean implementaciones para *origin* y *extent*. De la misma forma, los métodos *boundingBox* y *translateTo:* basan su implementación en métodos abstractos. Estos métodos son mostrados en la Figura 2.3.

```
displayBox
  ^self origin extent: self extent

boundingBox
  ^0 @ 0 extent: self extent

translateTo: aPoint
  self translateBy: aPoint - self origin
```

Figura 2.3 Definiciones de métodos *template*

Un ejemplo más complejo de un algoritmo codificado a través de un método *template*, es el método *displayOn:* de la clase *ContainerFigure*, mostrado en la Figura 2.4. Este tipo de figura está diseñada para contener en su interior el conjunto de figuras que se está editando en un momento dado. En consecuencia, mostrar una *ContainerFigure* implica mostrar todas las figuras que estén dentro del área visible de la *ContainerFigure*. El algoritmo, en este caso, calcula qué figuras deben ser visualizadas, independientemente del cálculo de posición de cada figura (a través del método *displayBox*) y de cómo se visualiza cada figura (método *displayOn:*).

```
displayOn: aGraphicsContext
  | clipRect i |
  figures isEmpty ifTrue: [^self].
  clipRect := aGraphicsContext clippingBounds.
  i := figures size.
  [i > 0] whileTrue:
            [(clipRect intersects: (figures at: i) displayBox)
                  ifTrue: [(figures at: i)
                                    displayOn: aGraphicsContext].
            i := i - 1]
```

Figura 2.4 Definición del método *template DisplayOn:*

Desde el punto de vista de la codificación del comportamiento de un dominio de aplicación, los métodos *template* representan el contenido clave de una clase abstracta. A través de métodos *template* es posible implementar el comportamiento genérico de las entidades de un dominio, el cual puede ser reutilizado, sin cambios, en diferentes clases concretas. Este comportamiento es adaptado a través de la redefinición de los métodos abstractos que el algoritmo invoca. De este modo, ellos ofrecen un mecanismo estructurado para codificar conocimiento acerca del comportamiento de un dominio, permitiendo la reutilización simultánea de interfaz e implementación.

1.2.2. Extensión de funcionalidad y reutilización de implementación

Ya se vio que aquellos métodos que poseen implementación y que no deberían ser redefinidos son representados cómo métodos base. En otras circunstancias, una clase puede proveer una implementación por omisión, como por ejemplo, el método de *Figure* que verifica si la figura contiene un determinado punto. La implementación por omisión se limita a verificar si el punto está contenido dentro de la región definida por el *displayBox* de la figura. Este

comportamiento puede ser refinado por subclases que, por sus características específicas, pueden modificar parcialmente la semántica del método u ofrecer implementaciones más apropiadas. La clase *PolylineFigure*, por ejemplo, calcula la inclusión de acuerdo a si la figura es cerrada o no, mientras que la clase *EllipseFigure* provee una implementación más adecuada para su forma. Este tipo de métodos son denominados **métodos *hook*** debido a que proveen una implementación de un comportamiento de utilidad que, en ciertos casos, puede ser extendido por alguna subclase [FHLS99].

Esto es común en la secuencia de inicialización de componentes, donde una clase puede inicializar los atributos definidos por ella, utilizando el método heredado para inicializar aquellos atributos definidos en sus superclases. La Figura 2.5 muestra la estructura general de este mecanismo de inicialización; en este ejemplo se supone que la clase *A* tiene definidos dos atributos, *a1* y *a2*, y la clase *B*, subclase de *A*, tiene definidos otros dos atributos, *b1* y *b2*.

```
Código de clase A
        initialize
                a1 := initValue1.
                a2 := initValue2.

Código de clase B, subclase de A
        initialize
                super initialize.
                b1 := initValue3.
                b2 := initValue4.
```

Figura 2.5 Estructura común de los métodos de inicialización

La definición de métodos *template* debes ser extendida para considerar también los métodos *hook*. Entonces un método *template* será aquel cuyo comportamiento dependa de los métodos abstractos y/o *hook* que invoque.

Los métodos *hook* son otra herramienta esencial provista por las clases abstractas, para extender o adaptar, de una manera estructurada, la funcionalidad de una abstracción. A través de la definición adecuada de métodos *hook* y *template*, es posible implementar una estructura adaptable que respete protocolos establecidos y permita reutilizar tanto implementación como interfaz a través de herencia o composición de objetos.

Debido a que, generalmente, los métodos *template* son diseñados con el objetivo de no ser especializados, Pree [Pre95, Pree99] los denomina *puntos congelados* (***frozen spots***). En contraste, los métodos abstractos y *hooks* son llamados *puntos calientes* (***hot-spots***), o *puntos de flexibilización*.

Debe notarse que las categorías de esta clasificación de métodos (abstractos, *template*, *hook* y bases) representan decisiones de diseño mutuamente excluyentes, es decir, un método dado no puedo pertenecer a dos categorías distintas. Por ejemplo, tanto los métodos *templates* como los bases son diseñados para no ser especializados. Sin embargo, un método base no puede invocar otro método que pueda ser especializado (*hook* o abstracto), mientras que, por definición, un *template* debe hacerlo.

2. *Frameworks* Orientados a Objetos

Si bien las clases abstractas proveen una forma de expresar el diseño de una clase, las clases tienen un nivel de granularidad demasiado pequeño [WBJ90]. Una aplicación orientada a objetos es construida a través de clases que describen la colaboración de un conjunto de

instancias para realizar las tareas del sistema. La distribución de responsabilidades entre esas clases, y los patrones de colaboración entre ellas, constituyen el **diseño** de esa aplicación.

A través de los mecanismos provistos por el paradigma es posible reutilizar una aplicación tanto como una "caja negra" (a través de una interfaz que permita acceder los servicios que implementa) cuanto como una "caja blanca", redefiniendo el comportamiento de algunas subclases. Así, se pueden obtener diferentes aplicaciones utilizando como base una aplicación existente, reutilizando tanto el código como el diseño general de esa aplicación. La cantidad de comportamiento redefinido dependerá, evidentemente, del grado de semejanza entre las aplicaciones. Si las aplicaciones fueran semejantes, probablemente sólo algunos métodos deban ser redefinidos y pocas clases especializadas, obteniendo, en consecuencia, una gran reutilización. Al contrario, si las aplicaciones difieren mucho, probablemente muchas clases necesiten ser redefinidas, disminuyendo la ganancia de reutilización.

Analizando las especializaciones requeridas para diferentes aplicaciones, es posible detectar que existen patrones de comportamiento comunes, que pueden ser generalizados y reutilizados por todas las aplicaciones. Dadas dos clases, por ejemplo, se puede crear una nueva clase, superclase de las primeras, que contenga todo el código común a ambas, mientras las originales implementan la porción específica de comportamiento. De esta forma, a partir del análisis de un conjunto de casos concretos, es posible obtener una clase abstracta, que defina el comportamiento genérico de esa familia de casos. Después de este proceso, la creación de una nueva aplicación, probablemente, requerirá la redefinición de mucho menos comportamiento que en la situación anterior [CHSV97].

Las clases abstractas representan conceptos genéricos relativos a una familia de objetos relacionados. Cada tipo de objeto representa un caso particular de la abstracción, que será representado por una subclase concreta, subclase de la abstracta. Esta subclase proveerá una variante específica del comportamiento abstracto definido en la clase abstracta. Como ya fue expuesto, las clases abstractas funcionan como un molde para sus subclases. De la misma forma, un diseño constituido por clases abstractas funciona como un *molde para aplicaciones*. Un diseño constituido por clases abstractas es denominado un *framework de aplicación orientado a objetos* o simplemente *framework* [BD99, FS97, HHG90, Joh92, Joh97, Sou99]. El *framework HotDraw*, por ejemplo, está constituido principalmente por clases abstractas que representan las figuras a ser editadas (*Figure*) y las herramientas (*Tool*) y manipuladores (*Handler*) utilizadas para transformarlas.

Informalmente un *framework* puede ser considerado como una infraestructura de clases que proveen el comportamiento necesario para implementar aplicaciones dentro de un dominio de aplicación, a través de los mecanismos de especialización y composición de objetos, típicos de los lenguajes orientados a objetos. Más formalmente, un *framework* es definido, desde el punto de vista de su objetivo, como una aplicación semicompleta, reutilizable, que puede ser especializada para construir aplicaciones que satisfagan requisitos específicos [JF88]. En cambio, considerando la perspectiva estructural, un *framework* puede ser definido como el diseño reutilizable de un sistema, que describe cómo el sistema es descompuesto en un conjunto de componentes que interaccionan [Joh97].

Una de las características más importantes de los *frameworks* es la *inversión de control* [FS97]. Una aplicación construida con un *framework* puede ser conceptualizada como compuesta de dos niveles (Figura 2.6). Un nivel superior que provee la estructura de control de la aplicación, constituido por las clases del *framework*, y un nivel inferior, constituido por subclases concretas implementadas por el usuario. Este nivel provee la implementación de operaciones específicas cuya activación es modelada en el nivel superior. En el caso general, estas operaciones pueden especializar la estructura de control de acuerdo con los requisitos de la aplicación específica, así como llamar a operaciones definidas en el nivel superior.

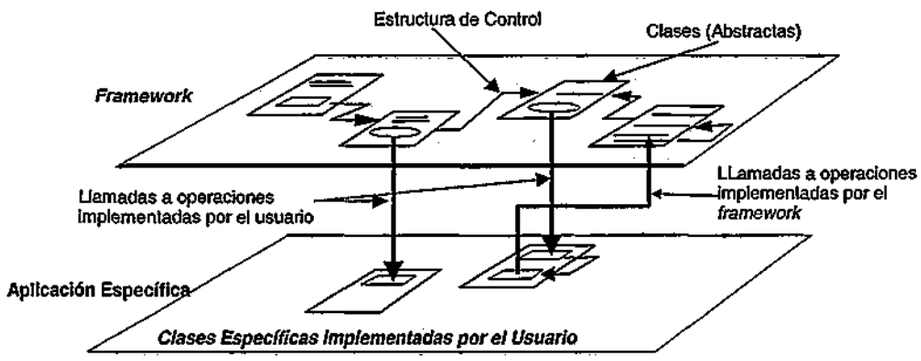


Figura 2.6 Visión conceptual de la estructura de un *framework* [Cam97]

Esta división de niveles puede ser vista en el ejemplo de la clase *Figure* presentado en la sección anterior, específicamente en la definición del método de la Figura 2.4. En este ejemplo, el método *displayOn:* está en el nivel superior, siendo éste el método que define la estructura de control del proceso de dibujo de figuras. A su vez, el método invoca código específico de cada aplicación (nivel inferior), a través del método *displayBox.*

2.1. *Frameworks* versus Bibliotecas de Clases

Si bien un *framework* podría ser considerado similar a una biblioteca (o jerarquía) de clases, en el sentido que ambos proveen un conjunto de clases para ser reutilizadas, existen importantes diferencias conceptuales. Es importante destacar estas diferencias, pues no sólo representan gran parte de las ventajas ofrecidas por el uso de *frameworks*, sino que también resultan factores que dificultan la tarea de entender un *framework.*

Existen tres diferencias principales entre las bibliotecas de clases y los *frameworks*:

- Comportamiento versus protocolo: Las bibliotecas de clases son esencialmente colecciones de comportamientos que el usuario puede invocar cuando quiere incorporar esos comportamientos individuales a su programa. Por ejemplo, si necesita representar la edad de una persona, puede utilizar la clase *Integer*; esta clase no hace ninguna presunción sobre el contexto donde la clase será utilizada. Los *frameworks*, por su parte, no sólo proveen comportamiento, sino que también proveen el protocolo o conjunto de reglas que gobiernan las formas en que los distintos comportamientos pueden ser combinados, incluyendo reglas que describen qué debe proveer el programador en oposición a lo que provee el *framework.* Por ejemplo, si el usuario decide crear un editor utilizando *HotDraw* y especializa la clase *Figure*, debe conocer qué métodos se supone que deben ser redefinidos.
- *No nos llame, nosotros lo llamaremos*: Con una biblioteca de clases, el código que el programador escribe crea instancias de clases e invoca a sus métodos. En el ejemplo de la edad de una persona, creará una instancia de la clase *Integer* y luego puede enviarle mensajes tales como +, >=, etc. Con los *frameworks*, también es posible crear instancias e invocarlas de la misma manera, es decir, utilizar los *frameworks* como si fuesen bibliotecas de clases. Sin embargo, para aprovechar completamente el diseño reutilizable de un *framework*, el programador escribe código que reescribe partes del *framework* y es llamado por éste. Es decir, el *framework* administra el flujo de control entre sus objetos. Esto es lo que sucede, por ejemplo, cuando se especializa la clases Figure y se reescriben los métodos abstractos. Escribir un programa implica repartir las responsabilidades entre las diferentes componentes de software que son invocadas por el *framework*, en vez de especificar cómo las diferentes componentes deben trabajar juntas. Esta relación es expresada por el

principio *No nos llame, nosotros lo llamaremos*, también conocido como el principio de *Hollywood* [FS97].

- Implementación versus diseño: Con las bibliotecas de clases los programadores reutilizan sólo las implementaciones, mientras que con los *frameworks* lo que se reutiliza es el diseño. Un *framework* comprende una familia de programas relacionados; representa una solución genérica de diseño que puede ser adaptada a una variedad de problemas específicos en un dominio dado. Por ejemplo, un *framework* puede establecer la forma en que funciona la interfaz de usuario, aún si dos interfaces diferentes creadas con el mismo *framework* pueden resolver problemas de interfaz muy diferentes.

Un *framework* incluye conocimiento específico de un dominio. El código de la aplicación adapta al *framework* de forma de crear una aplicación particular dentro del domino general de problemas para cuya resolución el *framework* fue diseñado. Por ejemplo, en el caso de un *framework* desarrollado por un banco, el conocimiento de dominio involucrado en el *framework* puede ser cómo funcionan las cuentas de clientes o cierto tipo de transacciones financieras. Aquí el programador puede adaptar el *framework* para crear tipos específicos de cuentas o de instrumentos financieros.

El proceso de construcción de una aplicación específica del dominio de aplicación utilizando un *framework* es denominado **instanciación** [Joh88] y consiste, generalmente, de dos etapas [WBJ90]:

- Implementación del comportamiento específico: La forma habitual de utilizar un *framework* es a través del mecanismo de herencia, creando subclases de las clases abstractas que componen el *framework*. Estas subclases deberán, básicamente, implementar el comportamiento específico de los métodos abstractos definidos en las clases abstractas, además del comportamiento adicional necesario para satisfacer los requisitos de la aplicación.
- Composición de instancias: Cuando las clases necesarias ya han sido desarrolladas, el usuario del *framework* debe describir instancias de qué clases concretas serán utilizadas y cómo estas instancias estarán conectadas. Esto se hace, generalmente, a través de un *script* o programa principal [FSJ99].

Estas dos etapas caracterizan el proceso básico de construcción de aplicaciones de *frameworks* basados en herencia. Cuando los componentes o clases necesarias ya están disponibles, una aplicación puede ser construida simplemente describiendo cómo estos componentes se combinan. No obstante, esta situación sólo es posible en dominios muy restringidos, en los cuales la cantidad de comportamientos diferentes está limitada. En dominios más amplios es necesario, en general, la especialización de clases para obtener el comportamiento específico de cada aplicación o subsistema. Una cuestión esencial es cuánto comportamiento debe ser implementado y, especialmente, qué tipo de métodos deben ser redefinidos.

En principio, cualquier diseño formado por clases abstractas podría ser considerado un *framework*, pero el hecho de que esté formado por clases abstractas no garantiza que sea totalmente reutilizable para construir aplicaciones dentro de su dominio. No necesariamente esas clases definen el conjunto adecuado de abstracciones ni una interfaz apropiada que permita, a través de la composición de objetos y la redefinición de algunos métodos, construir aplicaciones dentro del dominio.

Los *frameworks* orientados a objetos son difíciles de desarrollar. Mientras que el desarrollo de sistemas de software complejos ya es difícil, mucho más trabajoso resulta desarrollar *frameworks* reutilizables para dominios de aplicación complejos [FS97, Sch97, Sch99].

2.2. Tipos de *Frameworks*

De acuerdo con los dos estilos de instanciación descritos, los *frameworks* pueden clasificarse según el estilo predominante:

- ***Frameworks Basados en Herencia:*** Los *frameworks* de esta categoría, también denominados *frameworks* dirigidos por la arquitectura (*architecture-driven frameworks*), representan el caso más común. Estos *frameworks* proveen la estructura completa de una aplicación (o subsistema), la cual es completada con el código específico de cada aplicación particular. Ellos son utilizados a través de la especialización de sus clases para implementar comportamiento específico. Originalmente fueron denominados ***frameworks* caja blanca** debido a la necesidad de conocer al menos una parte de su funcionamiento interno para adaptarlos [Joh88].

- ***Frameworks Basados en Composición:*** Los *frameworks* de esta categoría, también denominados dirigidos por los datos (*data-driven frameworks*) o ***frameworks* caja negra** (*black box frameworks*), se caracterizan por ser utilizados a través de composición de objetos para su adaptación. Los clientes del *framework*, es decir aquellos programas u objetos que utilizan su funcionalidad, adaptan el comportamiento del *framework* utilizando diferentes combinaciones de objetos. El *framework* varia su comportamiento en función de los objetos que el cliente pasa como parámetros, pero el *framework* determina cuales son las combinaciones válidas. Este tipo de *frameworks* constituye el basamento para la construcción de *toolkits* de componentes.

Ambos tipos de *frameworks* no son independientes uno del otro. En el caso general, los *frameworks* basados en composición son el resultado de sucesivas generalizaciones de un *framework* basado en herencia [BMA97]. Desarrollar desde el inicio *frameworks* basados en composición es una tarea más difícil que el desarrollo de *frameworks* basados en herencia [Joh93]. En el ciclo normal de desarrollo, un *framework* surge como un conjunto de clases obtenidas a través de la generalización de varios ejemplos. En sucesivas tentativas de reutilización del *framework*, nuevas clases son creadas para implementar comportamientos no provistos. Así, después del desarrollo de varias aplicaciones, surgen nuevas oportunidades para refactorizar el *framework* y generalizar todavía más el comportamiento de las aplicaciones del dominio [Opd92]. De este modo, un *framework* que inicialmente fuera concebido para ser utilizado por especialización, puede evolucionar hasta convertirse en un diseño paramétrico. Este diseño paramétrico puede ser instanciado a través de la combinación de objetos básicos que representan, cada uno de ellos, diferentes funcionalidades del dominio.

No obstante, como ya se dijo, diseños de esta naturaleza sólo pueden ser obtenidos fácilmente en dominios estables y restringidos, como por ejemplo, interfaces con el usuario basadas en diálogos. A continuación se describe brevemente el *framework HotDraw* introducido más arriba, el cual combina características de ambos tipos de instanciación.

2.3. Ejemplo de *framework*: *HotDraw*

HotDraw [BJ94, BrJ94, Joh88] es un *framework* para la creación de editores gráficos para figuras bi-dimensionales. *HotDraw* puede ser utilizado para construir editores para gráficos especializados, tales como esquemas de hardware, o diagramas de proyecto de software. La Figura 2.7 ilustra la interfaz de usuario de un editor construido utilizando *HotDraw*. Las figuras se dibujan en una pantalla de edición (*drawing view*) seleccionando una de las herramientas (*tools*) de la paleta de herramientas. El conjunto inicial de herramientas para el dibujo de figuras incluye: líneas; flechas; rectángulos; círculos; y texto. Las operaciones

de manipulación de estas figuras incluyen: selección; redimensionamiento; desplazamiento; borrado y agrupamiento.[3]

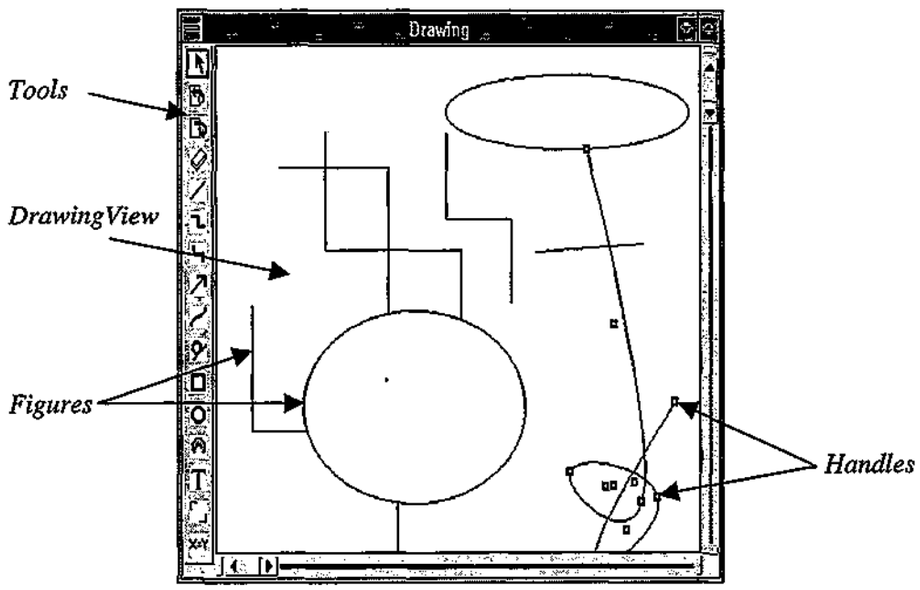


Figura 2.7 Ejemplo de un editor producido con *HotDraw*

Los usuarios de un editor construido con *HotDraw* pueden manipular figuras de diversas maneras, interaccionando directamente con ellas a través de la utilización del ratón. La interacción directa con una figura es realizada seleccionando la figura con la herramienta de selección y manipulando los puntos de control (*Handles*). Estos puntos de control permiten cambiar el tamaño y la forma de las figuras. También es posible abrir un menu de operaciones que pueden realizarse sobre la figura, como por ejemplo cambiar su color. Otra característica de *HotDraw* es la habilidad de mantener restricciones entre las figuras. Por ejemplo, un arco que liga dos figuras depende de la posición de ellas y, por lo tanto, cuando una es desplazada el arco será también desplazado para mantener la restricción.

La clase *Figure* representa los objetos que se dibujan con las herramientas; la clase *Tools* representa los objetos que aparecen en la paleta de herramientas y manipulan las figuras; la clase *Handles* representa los objetos que se utilizan como puntos de control que aparecen sobre las figuras; la clase *DrawingController* representa los objetos que administran la interacción del usuario en la pantalla de edición o área de dibujo; esta última es representada por la clase *DrawingView*.

Con la estructura provista por el *framework* es posible crear editores específicos, utilizando las clases concretas existentes (*SelectionTool* por ejemplo) y creando nuevos componentes a través de la especialización de otros, en el caso de que no existan en la biblioteca. Generalmente, cada nueva aplicación será representada por una clase encargada de crear las instancias de los componentes y vincularlos para iniciar la ejecución.

[3] Existen varias versiones disponibles del framework *HotDraw*, tanto en el lenguaje *Smalltalk* como en *Java*. Aquí se trabajará con la versión 41.4, la cual está disponible en Internet, en la dirección http://st-www.cs.uiuc.edu/users/brant/HotDraw/previous. La última versión disponible puede ser encontrada en http://st-www.cs.uiuc.edu/users/brant/HotDraw

2.3.1. Reutilización por Herencia

Por ejemplo, supóngase que se desea crear un editor gráfico que permita editar líneas, rectángulos, círculos y triángulos. Las clases que representam líneas, rectángulos y círculos ya existen en la biblioteca, pero no una que represente triángulos. El comportamiento genérico de las figuras es representado por la clase *Figure*, así que para poder editar triángulos de la misma forma que las otras figuras, deberá ser creada la clase *TriangleFigure* como una subclase de *Figure*. Para que *TriangleFigure* pueda ser utilizada, deberá proveer la implementación para los métodos abstractos *displayOn:* (para dibujar el triángulo) y *extent* (para informar del área ocupada por la figura), entre otros.

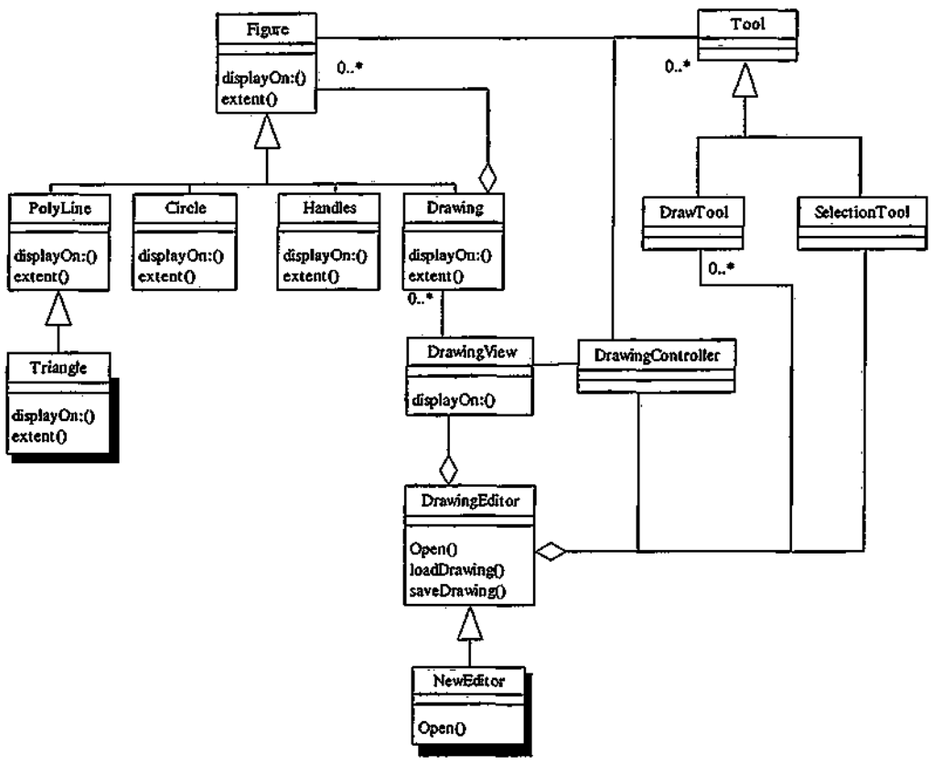


Figura 2.8 Estructura de Clases del *Framework HotDraw*

Para finalizar el editor, es necesario crear las instancias de las clases que lo componen, es decir, qué herramientas serán provistas por el editor, cuál será el área de dibujo y qué figuras serán manipuladas por el editor. Esto se hace definiendo una subclase de *DrawingEditor*. Ella debe proveer un método *open* encargado de crear una instancia de *DrawingView*, *Drawing* y *DrawingController* y conectarlas a través de los métodos que ellas proveen para definirles su modelo y su controlador. Además de eso, el método *open* deberá crear instancias de *SelectionTool* para poder seleccionar y cambiar el tamaño de las figuras y una instancia de *DrawTool* para cada tipo de figura, esto es, cada *DrawTool* recibirá como parámetro la clase de figura que va a crear. Así, un editor será creado simplemente enviando el mensaje *DrawingEditor open*[4].

[4] En realidad algunas otras operaciones deben ser definidas para obtener un editor operacional. Sin embargo, no son relevantes para los fines del ejemplo y dependen de la implementación *Smalltalk* del *framework*.

La Figura 2.8 muestra la estructura de clases del editor. Las clases sombreadas son las clases que debieron ser programadas para implementar el editor de dibujos descrito más arriba. En esta figura aparecen sólo algunos de los métodos relevantes para comprender el mecanismo básico de utilización del *framework*.

Como es simple de observar en el ejemplo, el usuario del *framework* sólo debe programar aquellos aspectos específicos a su aplicación. Esto implica la reutilización de la estructura de control global del diseño, además del código que implementa funcionalidad general. Así, instancias de *TriangleFigures*, como también otras figuras, pueden ser seleccionadas, redimensionadas, copiadas, desplazadas, giradas, borradas, almacenadas, etc. El comportamiento necesario para esto es heredado de las superclases abstractas. Básicamente, las nuevas subclases deben proveer implementaciones para los métodos abstractos definidos en las superclases, que serán invocados por otros objetos del *framework*, o por los métodos *template* de sus superclases.

2.3.2. Reutilización por Composición

La factorización de funcionalidad favorece mucho la reutilización de comportamientos ortogonales a través de la composición de objetos. El *framework HotDraw* también nos ofrece un ejemplo de reutilización por composición [BrJ94], a través de los *tools*. Existen dos formas de definir los *tools* que manipularán las figuras de un editor determinado. La primera consiste en crear nuevos *tools* especializando la clase *Tool*, es decir, reutilización por herencia. La segunda forma, y las más utilizada, consiste componer instancias de clases existentes, utilizando para ello una herramienta especialmente diseñada, el *Tool Builder*.

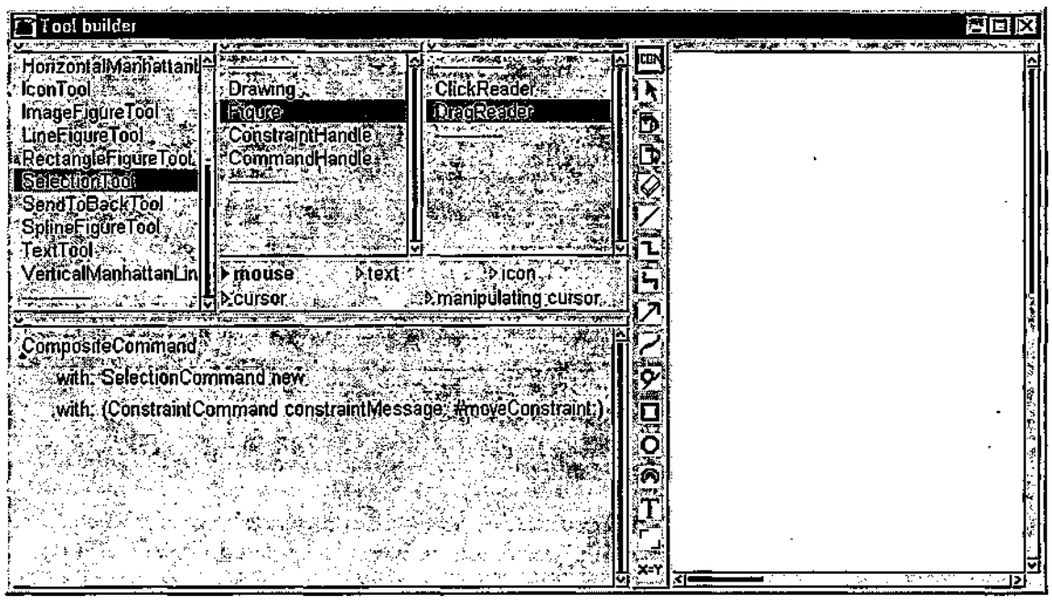


Figura 2.9 Interfaz para la definición de nuevos *Tools* por composición.

La creación de *tools* por composición en *HotDraw* implica la composición de instancias de, principalmente, dos clases: *Reader* y *Command*. Los *readers* son los encargados de interpretar las acciones del usuario, es decir los eventos de ratón y teclado, y los *commands* efectúan los cambios en el diagrama que está siendo editado. Cada tipo de *tool* tiene asociado una tabla, donde se representa las relaciones existentes entre figuras, *readers* y *commands*. De esta forma, es posible especificar como reaccionará el editor cuando el usuario efectúe una acción determinada (apretar un botón del ratón, por ejemplo) sobre un tipo dado de figura cuando esté seleccionado un determinado *tool*.

Todo esto puede ser especificado a través del *Tool Builder*, mostrado en la Figura 2.9. En esta figura puede verse como se define el *SelectionTool* asociando instancias de la clases *DragReader, CompositeCommand* y *SelectionCommand.* En concreto, la definición de la Figura 2.9 describe que si está activo una herramienta del tipo *SelectionTool* y se produce un evento de *drag* del ratón sobre una instancia de *Figure*, debe enviarse el mensaje *with:with:* a la clase *CompositeCommand*, con una instancia de *SelectionCommand* y una restricción como argumentos. Además de estas relaciones, utilizando el *Tool Builder* es posible definir los iconos y cursores asociados al *tool.*

3. Análisis: Ventajas y Problemas del desarrollo basado en *Frameworks*

Como en toda situación donde existen conflictos entre lo general y lo específico, es posible identificar ventajas y problemas. La utilización de *frameworks* orientados a objetos para el desarrollo de aplicaciones presenta las siguientes ventajas:

- Reutilización de diseño y no sólo de código
- Código ya probado y depurado
- Mejoras adicionales pueden ser hechas
- Más foco en el área de conocimiento
- Mejora en el mantenimiento

Desde el punto de vista de la creación de aplicaciones, se ha visto que preferentemente un *framework* debe ser especializado a través de la composición de instancias de las clases que lo forman. Pero esta situación sólo es posible en casos donde el dominio de aplicación del *framework* es acotado y con pocas variantes. En la mayoría de los casos el dominio será demasiado rico para permitir este tipo de especialización, siendo necesaria la especialización del *framework* a través de la creación de subclases de las clases del *framework* y redefinición de métodos.

Dentro de los *frameworks* basados en herencia, la situación ideal es aquella donde las clases definidas por el usuario sólo deben especializar los *hot-spots*, es decir, aquellos puntos previstos para especialización en el diseño del *framework*, los métodos abstractos y *hook*. Sin embargo, esto implica que el diseñador del *framework* debe ser capaz de anticipar todas las necesidades de adaptación que tendrán los usuarios del *framework*, algo que no es sencillo de alcanzar, sobre todo en dominios de aplicación complejos.

En consecuencia, la situación más realista es que un *framework* podrá ser usado para implementar una aplicación de forma más o menos sencilla en tanto que el dominio del *framework* no sea muy complejo y la aplicación se ajuste a las previsiones del diseñador. En otros casos, el usuario del *framework* deberá ser capaz de entender el diseño del mismo para adaptarlo a requisitos no previstos en ese diseño.

Si bien siempre el usuario debe tener ciertos conocimientos para ser capaz de crear una aplicación a partir del *framework*, estos conocimientos deben ser mayores a medida que nos movemos de una especialización por composición hacia una especialización que envuelva funcionalidad no prevista en el diseño del *framework*. En el caso de la especialización por composición, el usuario debe conocer las combinaciones válidas de componentes que pueden ser hechas. En el caso de especialización por refinamiento de los *hot-spots*, el usuario debe saber qué método especializar de acuerdo a los requisitos y cuál es el comportamiento esperado de los nuevos métodos. Esto, básicamente, implica conocer qué protocolos de comunicación con otros objetos deben respetar esos métodos. Ya en el caso más general de tener que realizar adaptaciones no previstas, el usuario debe entender el diseño del *framework*, las razones detrás

de ese diseño y el efecto que tendrán los cambios que pretende realizar. Esto implica conocer el diseño casi tan bien como los diseñadores del *framework*.

De esta forma, entender el *framework* se vuelve un paso fundamental en el proceso de instanciación, porque de otra forma no sería posible utilizarlo. Para que el *framework* resulte de utilidad, es muy importante que esté acompañado de documentación adecuada, que permita alcanzar el grado de entendimiento necesario para una adaptación. Más aún, es deseable la existencia de herramientas que ayuden al usuario a comprender el *framework*, facilitando el proceso de creación de aplicaciones.

Para analizar las características que deben tener la documentación y las herramientas de apoyo a la instanciación de *frameworks*, es necesario estudiar las dificultades asociadas al entendimiento de los mismos. En el próximo capítulo se analizarán los factores que influyen sobre la facilidad de comprensión de un *framework*, así como las técnicas de documentación y herramientas disponibles hoy en día para auxiliar en esta comprensión.

III Frameworks: Documentación y Apoyo a la Instanciación

En el capítulo anterior se analizaron las características de los *frameworks* orientados a objetos, así como también el proceso de creación de aplicaciones a partir de ellos. En función de esto, se concluyó que es de fundamental importancia facilitar la labor de comprensión del *framework*, tanto a través de una adecuada documentación como a través de uso de herramientas de apoyo al proceso de instanciación.

En este capítulo se analizan las técnicas existentes para apoyar la utilización de *frameworks* para crear aplicaciones. En primer lugar se hace un análisis de las necesidades de los usuarios de *frameworks* y luego se describen las técnicas utilizadas para documentar diseños orientados a objetos y más concretamente *frameworks* y las herramientas desarrolladas para asistir el proceso de creación de aplicaciones a partir de *frameworks*.

1. Comprensión de *Frameworks* Orientados a Objetos

El inconveniente habitual que los usuarios de un *framework* encuentran es que para conseguir crear una aplicación, es decir, especializar las clases abstractas y construir la aplicación a partir de esas clases, necesitan, generalmente, comprender el diseño detallado de las clases del *framework*.

Las características propias de una aplicación orientada a objetos hacen que este tipo de aplicaciones no sean fáciles de entender. El flujo de control es transferido de un objeto a otro por medio del envío de mensajes y por lo tanto se haya distribuido entre las clases que componen la aplicación. Como consecuencia, resulta difícil entender el comportamiento de una de estas aplicaciones, sobre todo si para ello sólo se dispone del código.

Cuando se trata de *frameworks*, la situación es más compleja aún. Una solución general puede tratar, sin cambios, diferentes variantes de un problema dado. Una solución flexible, por su parte, es una solución que a través de pequeñas cambios en la estructura puede ser adaptada para resolver esas diferentes variantes. Las soluciones generales son ciertamente deseables, pero en la mayoría de los casos, presentan problemas de desempeño o están limitadas a dominios muy restringidos [Par79]. Las soluciones flexibles pueden ser adaptadas para cada caso particular, permitiendo explorar al máximo aquellos aspectos que simplifican su solución en cuanto a desempeño y funcionalidad.

En el caso de los *frameworks*, estos representan un compromiso entre una solución general y una solución flexible, aunque en la mayor parte de las situaciones un *framework* no ofrece tanto soluciones generales, como sí estructuras adaptables a distintas necesidades. El problema es que, en el caso general, estructuras de diseño muy flexibles resultan en diseños altamente complejos y, por lo tanto, difíciles de comprender.

Estos problemas pueden resumirse en las dificultades descritas por Butler y otros [BKM99] para entender un *framework*, sobre todo la primera vez que se usa:

- El diseño es muy **abstracto**, de forma de poder representar los puntos comunes de las aplicaciones del dominio
- El diseño es generalmente **incompleto**, requiriendo subclases adicionales para crear una aplicación
- El diseño provee mucha **flexibilidad**, que en algunos casos es excesiva. A veces, no toda la funcionalidad es necesaria en la aplicación que se está implementando.

- Las colaboraciones y dependencias resultantes entre clases pueden ser **indirectas** y **oscuras**.

1.1. Lectores de la Documentación de *Frameworks*

Un problema adicional que se debe tener en cuenta al analizar los requisitos de la documentación de *frameworks* es la variedad de audiencia potencial de esta documentación, lo que ocasiona que deba adaptarse a diferentes necesidades. Así, en general, los siguientes tipos de usuarios de *frameworks* son distinguidos en la literatura [BKM99, Mat96]:

Encargados de elegir el *framework*: ingenieros de software que deben decidir qué *framework* utilizar y que deben ser capaces de evaluar rápidamente el ámbito de aplicación del *framework* y decidir si es adecuado para la aplicación que debe ser construida. Una evaluación rápida y acertada puede ayudar a evitar costos innecesarios.

Programadores de aplicaciones: estos usuarios necesitan saber cómo adaptar el *framework* para producir la aplicación deseada. Aquí se puede distinguir entre dos tipos de programadores:

- Aquellos que quieren utilizar el *framework* de forma normal, para desarrollar aplicaciones que se ajusten al propósito para el cual el *framework* fue diseñado. Estos usuarios necesitan documentación que describa cómo está previsto que el *framework* sea usado.
- Aquellos que quieren ir más allá del uso normal del *framework* y agregarle nuevas características. Estos usuarios necesitan un entendimiento más profundo del funcionamiento y diseño del *framework*.

Encargados del mantenimiento del *Framework*: Un diseñador y/o programador encargado del mantenimiento y evolución del *framework* debe entender su diseño. Es decir, debe conocer el diseño interno del *framework* y los motivos de ese diseño (*design rationale*), así como también el dominio de aplicación y la flexibilidad requerida para el *framework*. Por esta razón, la calidad de la documentación de un *framework* es de importancia fundamental para su mantenimiento y evolución. La documentación debe reflejar claramente todas las decisiones tomadas en cada una de las etapas del desarrollo, desde el análisis hasta la programación, de modo que los cambios necesarios puedan ser realizados sin necesidad de reconstruir el diseño partiendo del análisis del código.

Diseñadores de *Frameworks*: estos usuarios buscan ideas en los *frameworks* existentes, aún en *frameworks* de dominios diferentes. Estos diseñadores requieren, mayormente, información de alto nivel de abstracción, aunque sus necesidades son similares a las de los encargados del mantenimiento [Boo99].

Verificadores: Algunos diseñadores de *frameworks* y aplicaciones pueden estar interesados en la corrección de su sistema. Ellos necesitarán verificar ciertas propiedades del sistema, de forma tal de asegurar que se satisfagan las necesidades de los clientes.

Los diferentes tipos de usuarios de la documentación de un *framework* imponen distintos requisitos sobre esta documentación. Cada uno necesita diferentes puntos de vista del *framework*, variando sobre todo el nivel de abstracción y grado de formalización necesario en cada caso.

Para analizar los requisitos de la documentación de *frameworks*, se comenzará por describir los requisitos de la documentación de bibliotecas de clases. De acuerdo a [LHM+90] la siguiente información debe ser incluida en la documentación para entender y usar las clases de una biblioteca de clases:

- Información estructural, tal como el nombre de la clase, la superclase si existe, tipo y orden de cualquier parámetro en tiempo de instanciación, más información similar para los métodos.
- Descripciones en lenguaje natural que comenten, para cada clase, la esencia de la clase y la abstracción que representa.
- Utilización: describe si la clase está diseñada para ser instanciada de una forma particular o para no ser instanciada.
- Terminología: la terminología introducida con respecto a los conceptos que la clase captura.
- Configuración: descripción de cómo las clases se relacionan una con otras (similar a los patrones de diseño, ver §III.2.2.1) y la forma en que está previsto que las clases se instancien en ciertas configuraciones.
- Aserciones: restricciones semánticas que establezcan precondiciones y postcondiciones para los métodos e invariantes de clases
- Métodos: para cada método información estructural tal como parámetros, resultados de operaciones y los tipos correspondientes.

Todos estos aspectos son también relevantes para la documentación de *frameworks*. Además, como se ha indicado más arriba, la documentación de *frameworks* debe ser descrita en diferentes niveles de abstracción, debido a que debe atender las necesidades de programadores con diferentes niveles de experiencia, conocimientos y objetivos.

Además, según Johnson [Joh92] la documentación de un *framework* debe incluir cuatro elementos: el propósito del *framework*, cómo utilizarlo, el propósito de los ejemplos de aplicaciones y el diseño detallado del *framework*. Dos de estos elementos exigen especial atención:

Cómo utilizar el *framework*: éste es el aspecto más importante de la documentación para alcanzar una utilización óptima del *framework*. A menudo la documentación se centra en explicar cómo el *framework* funciona y está estructurado, sin explicar cómo usarlo. Es más importante intentar el enfoque inverso: primero describir cómo utilizar el *framework* y luego explicar cómo funciona.

Es muy importante destacar que si el *framework* no está documentado apropiadamente para facilitar su uso, entonces no será utilizado [Jol99, MS99]. Es posible afirmar que un *framework* sólo puede ser tan bueno como sea la comprensión sobre la forma de utilizarlo que se alcance. Mucho esfuerzo es necesario para convertir la documentación de uso en un producto de calidad. El usuario del *framework* no invertirá mucho esfuerzo en aprender sus características intrincadas. Él quiere algún tipo de guía y orientación sobre cómo utilizar el *framework*, es decir, suficiente información para que él pueda reutilizar sus partes sin tener que conocer todos los detalles. Esto es similar a la noción de instrucciones minimalistas [RC93].

El diseño del *framework*: la descripción del diseño detallado del *framework* debe contener tanto las clases y sus relaciones como las colaboraciones entre clases. Mientras la documentación de uso sirve a los fines de los usuarios ocasionales, la documentación del diseño ayuda a usuarios experimentados a utilizar el *framework* en otras aplicaciones, además de las previstas por el diseñador original.

2. Técnicas de Documentación de *Frameworks* Orientados a Objetos

El problema de la dificultad de utilización de los *frameworks* ha llevado a proponer numerosos enfoques, orientados a proveer un asistencia para diferentes tipos de usuarios. Los diferentes enfoques propuestos durante los últimos años pueden ser divididos históricamente en

cuatro categorías principales: técnicas genéricas, técnicas específicas para *frameworks*, técnicas mixtas basadas en documentación electrónica y técnicas orientadas a herramientas.

2.1. Técnicas de Documentación de Diseños Orientados a Objetos

La primera y más simple de las técnicas utilizadas para documentar *frameworks* es incluir con la distribución del *framework* el código fuente de aplicaciones que hayan sido construidas utilizando el mismo. Ésta es, frecuentemente, la única documentación disponible para los usuarios del *framework*, e idealmente debe consistir de un conjunto de ejemplos de aprendizaje que introduzcan el *framework* de forma gradual e ilustren en cada paso un solo *hot-spot*, empezando por las formas más simples y comunes de reutilización de ese *hot-spot*. De todas formas, las aplicaciones de ejemplo comúnmente no están organizadas y el usuario debe navegar a través del conjunto de clases, buscando un ejemplo con funcionalidad similar a la aplicación que quiere construir.

Además de los ejemplos de aplicaciones, los primeros *frameworks* eran documentados utilizando notaciones generales para documentar software orientado a objetos, como por ejemplo *OMT* [RBP+91] y *OOSE* [JCJO92]. Debido a que estas notaciones no fueron diseñadas específicamente para documentar *frameworks*, fallan en capturar los aspectos esenciales de los mismos, especialmente las colaboraciones entre clases abstractas. Más aún, están orientadas a documentar un diseño dado, pero no proveen la asistencia necesaria para reutilizar, es decir, crear distintas aplicaciones a partir de ese diseño.

Otra técnica genérica adaptada a los *frameworks* son los manuales de referencia. Un manual de referencia para un sistema orientado a objetos consiste en una descripción de cada clase (responsabilidad, atributos o variables de instancia y métodos), variables globales, constantes y tipos. Aplicados a *frameworks*, estas descripciones deben incluir información, por ejemplo, acerca de si la clase está prevista para ser especializada o si un método dado debe ser reescrito. Los manuales de referencia por si solos no son una forma muy útil de aprender a utilizar un *framework* [BD99].

2.2. Técnicas Específicas para *Frameworks*

Una de las primeras técnicas diseñadas específicamente para documentar *frameworks* fueron las recetas y libros de recetas (*cookbooks*) [Pre95]. Una receta describe como desarrollar un típico ejemplo de reutilización durante la creación de la aplicación, mientras que los libros de recetas son colecciones de recetas. Las recetas no explican las razones de diseño, sino que sólo explican cómo un problema puede ser resuelto utilizando el *framework*. Generalmente se provee una guía a los contenidos de las recetas, ya sea a través de una tabla de contenidos o utilizando la primera receta como un resumen general del libro. Ejemplos de este tipo de documentación son [Ado85] y [KP88].

Para dotar a los libros de recetas de una mejor estructura, Johnson [Joh92] propuso documentar *frameworks* utilizando una notación textual informal, que estructura la descripción como un conjunto de ***patrones***[1] (*patterns*) organizados jerárquicamente. Un *patrón* describe un problema que se repite en el dominio del *framework* y entonces describe como resolverlo, siguiendo la estructura siguiente:

- Una descripción del problema.
- Discusión detallada de las diferentes formas posibles para resolver el problema, con ejemplos y referencias a otras partes del *framework*.

[1] Si bien en su trabajo Johnson los llamó simplemente patrones, aquí se los llamará patrones de aplicación, siempre que el contexto lo requiera, para distinguirlos de los patrones de diseño presentados en la próxima sección.

- Resumen de la solución, seguido de referencias a otros patrones que completan la descripción

La documentación de un *framework* estará constituida por un conjunto de estos *patrones*, los cuales pueden ser fácilmente comprendidos por usuarios no expertos en el *framework*. La organización sigue un enfoque en espiral, donde las recetas para las formas más frecuentes de reutilización son presentadas en primer lugar y los conceptos y detalles son diferidos tanto como sea posible, siendo la primera receta un resumen de los conceptos del *framework* y de las otras recetas.

Un patrón explica, en un lenguaje simple, lo que un usuario debe hacer para resolver un problema utilizando el *framework*, indicando, por ejemplo, qué clase debe ser especializada y qué método se debe implementar para conseguir un efecto deseado. El siguiente ejemplo documenta una funcionalidad provista por el *framework* para editores gráficos *HotDraw* (ver §II.2.3).

Patrón: Diagramas animados

Restricciones, manipuladores y herramientas permiten a un diagrama reaccionar ante las acciones del usuario, pero no pueden dar al diagrama una vida propia. Efectos de animación requieren un objeto controlador para dirigir a todas las figuras en un diagrama

La funcionalidad de animación es provista en *HotDraw* creando una subclase de *Drawing* que define el método ***step***. Esta es la principal razón para crear subclases de *Drawing*. Al diagrama se le envía repetidamente el mensaje *step* siempre que *HotDraw* se está ejecutando. El propósito del método ***step*** es mover cada una de las figuras del diagrama. Por ejemplo, *MovingDrawing* simula el problema de n cuerpos asignándole a cada figura una velocidad y asumiendo que las figuras ejercen fuerzas las unas sobre las otras. Su método ***step*** es:

```
step
  animating ifFalse:[^super step].
```

"Primero calcula las nueva velocidad de cada figura, calculando la fuerza gravitacional que cada una ejerce sobre las otras."

```
  self figures do:[:fig1 | fig1 calculateForceFrom: self figures].
```

"Luego mueve cada figura"

```
  self figures do:[:fig1 | fig1 step].
```

Típicamente existe alguna forma de detener la animación, en este caso asignando el valor *false* a la variable *animating*. La forma más fácil de modificar esta variable es través de una herramienta. La herramienta debe ser una instancia de *DrawingActionTool* que es parametrizada con un bloque que envía el mensaje *startAnimation* al diagrama cuando la herramienta es seleccionada y un bloque que envía el mensaje *stopAnimation* cuando la herramienta es deseleccionada. (Ver ***Tools*** (8))

Anime un diagrama creando una subclase para él que defina el método step, el cual ejecute el próximo paso en la animación.

Un concepto similar a los libros de recetas pero con objetivos diferentes son los libros o cuadernos de diseño (*design books* o *design notebooks*). Un libro de diseño recoge información de diseño, incluyendo la teoría subyacente, información del dominio y una discusión de las decisiones de ingeniería tomadas. En estos libros también se incluye información sobre los requerimientos del sistema, especificaciones, arquitectura, componentes de diseño, historia del diseño y código. Aunque no fueron diseñados para documentar *frameworks*, los libros de diseño han sido utilizados [ASP93] para capturar las decisiones y razones de diseño de los sistemas de software, así como también sistemas que combinan software y hardware.

2.2.1. Patrones de diseño

Siguiendo la idea de documentación con patrones textuales descrita arriba, es posible documentar el diseño de un *framework* a través de su descripción con patrones de diseño (*design patterns*) [Bos98b, GHJV94]. Estos patrones tienen por objetivo sistematizar el conocimiento y experiencia de diseño. Capturan experiencia de diseño a nivel de micro-arquitecturas, especificando las relaciones entre clases y objetos involucrados en un problema de diseño en particular. Un patrón de diseño presenta una solución al problema y provee un nivel de abstracción superior al de clases y objetos. La descripción del patrón en un nivel abstracto permite reutilizarlo para el diseño de nuevas aplicaciones, así como también mejorar el diseño de aplicaciones existentes, independientemente de las implementaciones. Uno de los objetivos de los patrones de diseño es proveer meta conocimiento sobre cómo incorporar flexibilidad a un *framework*. Además, este tipo de información puede ser muy útil para documentar el diseño del *framework* a un mayor nivel de abstracción, debido a que son apropiados para la descripción de arquitecturas.

La documentación de un patrón de diseño puede ser realizada de acuerdo al siguiente esquema:

- *Contexto*: Describe el contexto donde se identifica el problema de diseño.
- *Problema*: Describe explícitamente el problema de diseño a ser resuelto.
- *Solución*: Describe la solución del problema, definiendo los objetos y clases que participan del diseño y sus responsabilidades y colaboraciones, generalmente utilizando diagramas de colaboraciones y ejemplos de aplicaciones del patrón a situaciones concretas.
- *Consecuencias*: Describe las consecuencias que la solución adoptada tiene en el diseño de la aplicación, haciendo explícitas las ventajas y restricciones.

Con este esquema, es posible documentar todos los aspectos involucrados en la solución de cada problema de diseño, especificando el patrón aplicado y las consecuencias de aplicar dicho patrón en el diseño completo. Un ejemplo conocido de patrón de diseño es el patrón *Composite*, mostrado en la Figura 3.1.

Nombre: *Composite*

Intención:

Compone objetos en estructura de árbol para representar jerarquía de partes o de contenido. Define la forma de tratar objetos simples y compuestos uniformemente.

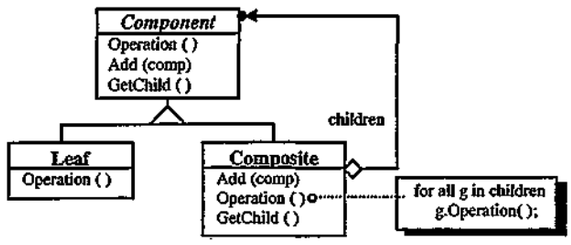
Motivación:

Los editores gráficos permiten al usuario construir figuras simples, tales como líneas o rectángulos, como así también figuras compuestas de estas figuras simples. Para evitar definir clases, representando objetos simples y compuestos, con interfaces diferentes, se usa el patrón *Composite*. Este patrón define una clase abstracta que representa tanto los objetos simples como los compuestos, en este caso sería la clase *Gráfico*. Esta clase declara operaciones específicas a todos los objetos gráficos, como por ejemplo *Dibujar*. Las subclases de *Gráfico* que representan a los objetos simples, *Línea*, *Rectángulo*, implementan el método *Dibujar* para dibujar estas figuras simples. La subclase representando el objeto compuesto, *Figura*, implementa *Dibujar* invocando esta operación sobre cada uno de los objetos que la componen.

Aplicabilidad:

Este patrón puede ser usado cuando se desea representar jerarquía de partes o de contenido y que el cliente ignore las diferencias entre los objetos simples y compuestos al aplicarles una operación determinada.

Estructura:



Participantes:

♦ *Component* (Gráfico): declara la interfaz de los objetos en la composición, implementa el comportamiento por omisión de la interfaz común a todas las clases.

♦ *Leaf* (Línea, Rectángulo): representa los objetos simples en la composición y define el comportamiento de estos objetos.

♦ *Composite* (Figura): define el comportamiento de los objetos compuestos, almacena los componentes hijos, implementa las operaciones relacionadas a los hijos.

Colaboraciones:

Los clientes usan la interfaz de la clase *Component* para interaccionar con los objetos en la estructura. Si el objeto es un *Leaf* la solicitud es manejada directamente. Si el objeto es un *Composite*, la solicitud es propagada a los componentes hijos.

Consecuencias:

Define jerarquía de clases constituidas por objetos simples y compuestos. Permite al cliente tratar los objetos simples y compuestos uniformemente, no es necesario saber con qué tipo de objeto se está trabajando simplificando el código a ser escrito. Además, permite agregar fácilmente nuevas clases de componentes.

Usos conocidos:

Ejemplos de este patrón se pueden encontrar en la arquitectura de varios *frameworks*. Por ejemplo, la clase *View* del *framework MVC* de *Smalltalk* es un *Composite*.

Patrones de Diseño Relacionados:

El patrón *Decorator* es usado frecuentemente con el *Composite*. El patrón *Chain Of Responsibility* también se relaciona.

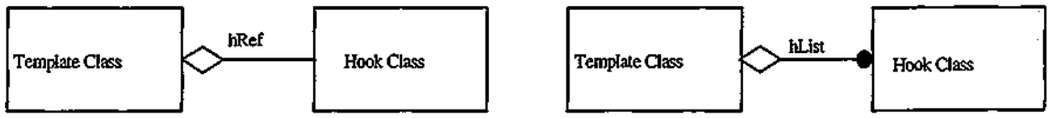
Figura 3.1 Descripción del patrón *Composite*

El *framework HotDraw* utiliza este patrón exactamente como el ejemplo descrito en la documentación del patrón de la Figura 3.1. La clase *Figure* desempeña el papel de *Component*, la clase *ContainerFigure* es el *Composite* y un conjunto de figuras se comportan como *Leaf*, *ElipseFigure* y *PolylineFigure* entre otras.

2.2.2. MetaPatrones

Pree propuso la idea de meta-patrones como un complemento para documentar la estructura esencial de un *framework* [Pre94]. Basándose en la premisa de que la estructura esencial de un *framework* es compuesta por la organización de los métodos genéricos (*template*)

y los métodos redefinibles (*hook*), Pree identifica un conjunto básico de combinaciones de estos dos conceptos en la estructura de un *framework*, los cuales determinan los metapatrones. Con esta premisa, una clase de un *framework* es considerada una clase *template* si posee un método *template*, clase *hook* si posee métodos *hook* o clase *template-hook* si posee ambos tipos de métodos.

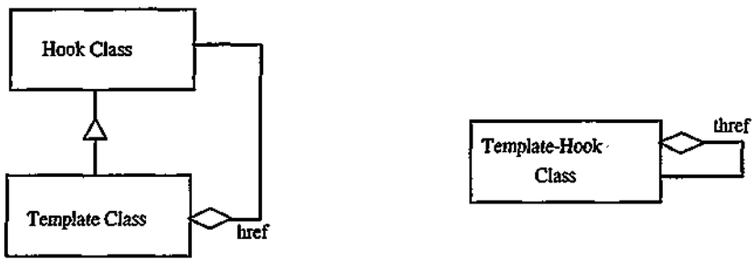


Patrón de Conexión 1:1 Patrón de Conexión 1:N

Figura 3.2 Patrones de conexión

Un método *template* es aquel que invoca otro método que puede ser redefinible, o sea, un método abstracto o *hook* (cabe aquí notar que Pree no distingue entre métodos abstractos y *hook*; él considera todos los métodos redefinibles como *hooks* porque a los efectos de la información que él quiere representar, esta distinción no es importante). Así, si el método *hook* pertenece a otra clase perteneciente a otra jerarquía, esto determina una organización denominada Patrón de Conexión. De acuerdo con la funcionalidad de la relación, este patrón puede ser 1:1 o 1:N (es decir, un método *template* envía el mismo mensaje, implementado con un método *hook*, a distintos objetos).

Una forma diferente de relación existe cuando los métodos *hook* pertenecen a una superclase en la jerarquía. Esto conduce a una conexión recursiva, en la cual una subclase invoca típicamente al mismo método definido en la superclase. El diagrama a la izquierda de la Figura 3.3 muestra la estructura de clases del Patrón de Conexión Recursiva, para el caso de una funcionalidad 1:1.



Patrón de Conexión Recursiva 1:1 Patrón de Unificación Recursivo 1:1

Figura 3.3 Patrones de Conexión Recursiva y de Unificación

Finalmente, cuando los métodos *template* y *hook*, se encuentran en la misma clase, los patrones anteriores se combinan en los Patrones de Unificación Recursiva, como muestra el diagrama de la derecha en la Figura 3.3.

Con estas estructuras básicas, es posible documentar la estructura esencial de un *framework* de varias formas y en diferentes niveles de abstracción. Una forma puede consistir simplemente en la estructura de clases *template* y *hook* de acuerdo con los meta-patrones. Alternativamente, la estructura de objetos puede ser complementada con la información de los meta-patrones en una notación de dos niveles. Como se observa en la Figura 3.4 los meta-patrones pueden ser de utilidad para explicar, en términos abstractos, la relación definida entre las clases de un *framework*, pudiendo complementar en un ambiente las notaciones convencionales ya conocidas.

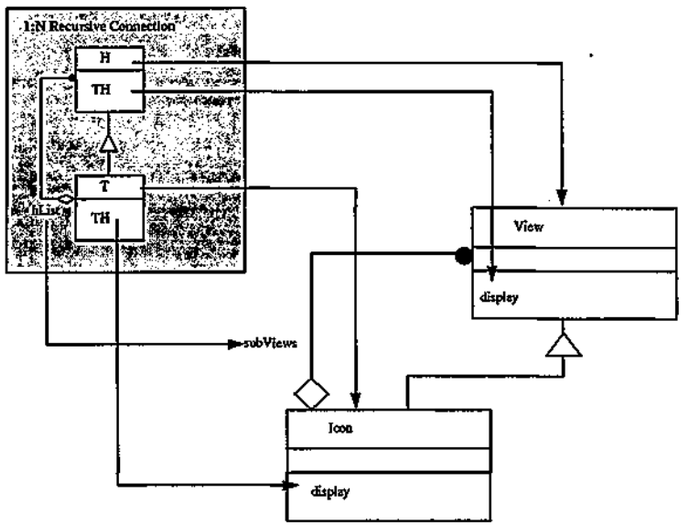


Figura 3.4 Anotación de la estructura de objetos con meta-patrones

2.2.3. Tarjetas de documentación de puntos de flexibilización (*Hot-Spot Cards - HSC*)

Las tarjetas de documentación de puntos de flexibilización [Pre96] se proponen como un mecanismo complementario para la documentación de los *hot-spots* definidos por los meta-patrones. Una *HSC* tiene por objetivo proveer la información necesaria para extender el *framework* en algún punto de flexibilización previsto por el diseñador. Las *HSC* son clasificadas en dos tipos:

- *HSC* de función: Identifica una función del *framework* que se debería mantener flexible, especificando el grado de flexibilidad: dinámica y adaptable por el usuario final. Esta indicación significa que el comportamiento del método documentado puede requerir ser variado dinámicamente en tiempo de ejecución. Esta información es utilizada para definir qué meta-patrón necesita ser utilizado para implementar la funcionalidad prevista.
- *HSC* de datos: Identifica elementos del dominio factibles de ser generalizados.

El formato típico de una *HSC* es mostrado en la Figura 3.5. En ambos casos, una *HSC* provee la definición general de la semántica del aspecto documentado y al menos dos ejemplos diferentes de implementaciones o entidades específicas.

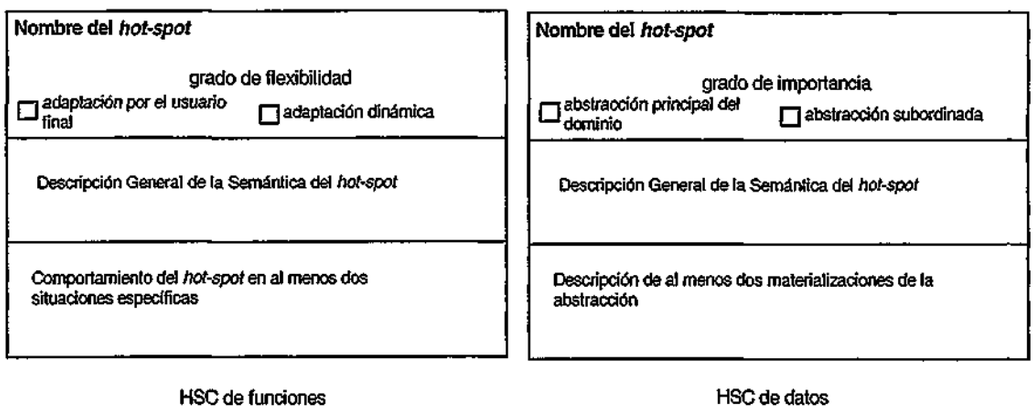


Figura 3.5 Esquema general de la tarjeta de funciones y datos

Por ejemplo, la figura siguiente muestra dos posible HSC correspondientes a un *framework* de alquileres.

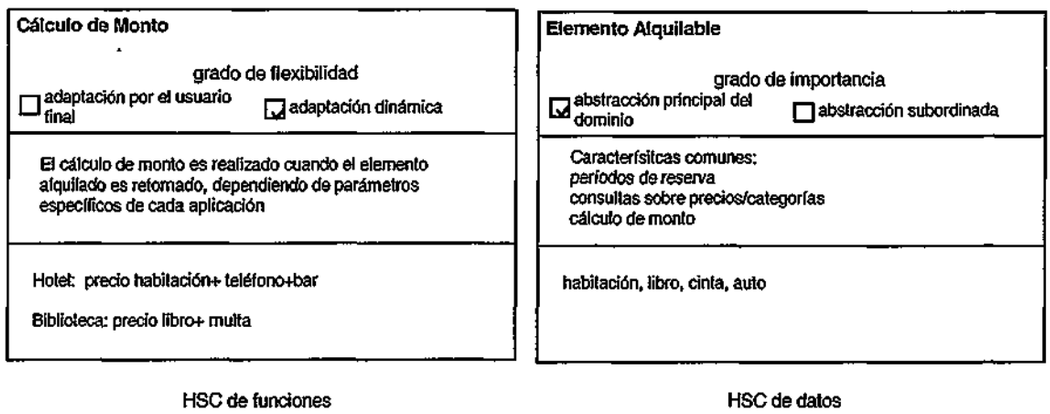


Figura 3.6 *HSCs* correspondientes a un *framework* de alquileres y préstamos de bienes

2.2.4. Técnicas formales

A medida que el tamaño y complejidad de los *frameworks* se incrementó, se hicieron necesarias técnicas más formales para representar la estructura de los *frameworks*. Un ejemplo de documentación formal son los **contratos de interfaz** [Mey92], los cuales son especificaciones de obligaciones, cada una proveyendo la especificación de la interfaz e invariante de una clase aislada; un contrato de interfaz especifica las restricciones de tipos dados por la signatura de un método y la interfaz semántica del método. De forma similar a otras técnicas centradas en clases individuales, este enfoque no puede trasladarse de forma apropiada al nivel de los *frameworks*.

En cambio, los **contratos de interacción** [HHG90] son también especificaciones de obligaciones, pero estos tratan con el comportamiento cooperativo de varios participantes que interaccionan para alcanzar un objetivo conjunto. Las colaboraciones entre clases y entre instancias son el aspecto más importante en un *framework*. Estas colaboraciones determinan el flujo de control abstracto que es impuesto por el *framework* y que representa la arquitectura de la aplicación. Las dependencias establecidas dinámicamente entre instancias e invariantes que deben ser mantenidos no pueden ser reflejados por el código aislado de las clases que componen el *framework*.

Un ejemplo clásico de este caso lo representan las dependencias entre instancias del modelo y las vistas, en el *framework MVC (Model-View-Controller)* [Gol83]. Este *framework*, desarrollado originalmente en Smalltalk y del cual existen actualmente implementaciones en otros lenguajes como Java y C++, provee un diseño genérico para la construcción de interfaces de usuario interactivas y hace hincapié en la división de responsabilidades. Para esto define tres tipos de componentes de una interfaz:

- modelos, que son aquellos objetos que representan los datos visualizados y manipulados a través de la interfaz.
- vistas, objetos que poseen el conocimiento acerca de cómo deben ser mostrados los datos del modelo
- controladores, que se encargan de interpretar las acciones producidas por el usuario a través de los dispositivos de entrada, como por ejemplo el teclado y el ratón.

Las vistas son dinámicamente asociadas con una instancia de su modelo y colaboran con él para mantener un invariante: una vista refleja siempre el estado de su modelo. Aquí es

imposible saber anticipadamente si una instancia del modelo va a tener vistas asociadas y a qué clase pertenecerán esas vistas.

Este tipo de colaboraciones es difícil de capturar en una notación que enfatiza la descripción del diseño en términos de clases. Por esto Helm y otros proponen la idea de contratos de interacción para especificar las *relaciones de comportamiento* de un conjunto de objetos que interaccionan [HHG90]. Un contrato describe las colaboraciones que existen entre un conjunto de participantes, a través de una lenguaje semiformal.

Un contrato especifica los siguientes aspectos de una relación de comportamiento:

- Participantes en la relación y sus obligaciones contractuales
- Un invariante, en cuyo mantenimiento colaboran los participantes en el contrato. El invariante, que puede no estar presente, consiste de un predicado que los participantes colaboran para mantener; durante la ejecución el invariante puede tornarse falso, requiriendo ser satisfecho nuevamente.

Dentro de las obligaciones contractuales se especifican:

- **Obligaciones de tipo**: las variables que los participantes deben poseer y la interfaz externa que deben proveer
- **Obligaciones causales:** los participantes deben realizar una secuencia ordenada de acciones y deben instanciar ciertas condiciones como verdaderas en respuesta a los mensajes

El ejemplo de la Figura 3.7 muestra la estructura parcial de una especificación de un contrato que describe las colaboraciones entre el modelo (*Model*) y las vistas (*Views*). El contrato establece los servicios que tiene que soportar cada uno de los participantes, los cuales deberán ser métodos implementados por las clases que participarán en el contrato.

La notación utilizada consiste de las siguientes construcciones especiales:

$\in$: pertenece $\quad\quad$ $\notin$: no pertenece $\quad\quad$ // comentario

$\Rightarrow$: implica $\quad\quad$ $\rightarrow$: mensaje $\quad\quad$ $\forall$: para todo

$<\|$ *variable: predicado: acciones*$>$: aplicación de las *acciones a* todos los elementos del conjunto descrito por el *predicado*, de forma paralela.

{*condiciones*}: precondiciones o postcondiciones

El contrato de la Figura 3.7 define dos participantes *Model* y *Views*. *Views* es un conjunto de componentes del tipo *View*, cada uno de los cuales, provee los servicios *setModel* (que indica para un *View* cual es el modelo) y *refresh* que tiene como precondición que la vista tiene que reflejar el estado del modelo para ejecutar la operación *display*. El *model*, por su parte, va a indicar a cada vista asociada cuando cambió su estado, a través del mensaje *changed*.

Un contrato puede ser especializado mediante la sentencia *REFINES*. La especialización de un contrato consiste en agregar acciones a los participantes o especializar la respuesta a algunos mensajes. Las obligaciones que son redefinidas, reescriben a las anteriores, mientras que el resto de las obligaciones son heredadas.

Estos contratos pueden ser utilizados para verificar la corrección de aplicaciones existentes, aunque debido a su complejidad esto es conveniente sólo si se cuenta con interpretación mecánica de los mismos. Sin embargo, no proveen elementos que permitan utilizarlos para asistir la construcción de nuevas aplicaciones usando el *framework*.

2.3. Técnicas Mixtas Basadas en Documentos Electrónicos

Varios trabajos han propuesto combinaciones de dos o más de las técnicas descritas, utilizando hipertexto como vínculo entre los diferentes componentes de la documentación. Por ejemplo, Lajoie y Keller [LK94] extendieron los patrones de aplicación, utilizando el término *motif* para nombrarlos y así evitar confusiones con los patrones de diseño. Ellos utilizan un esqueleto para la descripción de un *motif* que tiene el nombre y la intención, una descripción de la situación de reutilización, los pasos involucrados en la adaptación y referencias cruzadas a *motifs*, patrones de diseño y contratos. Los patrones de diseño proveen información sobre la estructura interna y los contratos proveen descripciones más rigurosas de las colaboraciones relevantes al *motif*.

Un enfoque similar fue propuesto por Demeyer y otros [DHS98], con la intención de asegurar consistencia entre la implementación del *framework* y la documentación relacionada. El enfoque automatiza el mantenimiento de vínculos hipermediales entre la documentación en línea y el código fuente del *framework*, utilizando enlaces hipermediales computados, es decir, vínculos que ejecutan un cálculo para determinar el destino de la acción de navegación.

```
CONTRACT ModelView
Model SUPPORTS [
        // Conjunto de atributos
 atributes []
 setVarValue(varName,val) ⇒ atributes[varName].value;
                // post-condición: la variable queda con el valor val
                        {atributes[varName].value=val};
                // invoca al servicio changed para informar a mudanza
                        changed().

        // Secuencialmente cada View recibe el mensaje refresh
 changed() ⇒ < || v: v ∈ Views: v → refresh() >
 addView(v) ⇒ { v ∈ Views }
 removeView(v) ⇒ { v ∉ Views }
 ]

Views: SET (View) WHERE EACH View SUPPORTS [
        // Actualiza la presentación en la pantalla después de actualizar los valores
 refresh() ⇒ { View reflects Model.atributes};
                display()
 setModel(m) ⇒ { model = m }
 ]
INVARIANT
        // Cada vista  siempre muestra el estado actual de su modelo
 Model.setVarValue(varName,val)
        { ∀ v: v ∈ Views: v reflects Model.atributes}
INSTANTATION
        Model → addView(View)
        View → setModel(Model)
END CONTRACT // ModelView
```

Figura 3.7 Ejemplo de Contrato de Interacción

Finalmente Meusel y otros [MCK97] proponen un modelo para agrupar distintas técnicas de documentación en una sola estructura, basado en el principio de la pirámide: la documentación sigue una estructura *top-down*, con hipervínculos que relacionan la información del mismo nivel de abstracción o de niveles contiguos. El primer nivel está orientado a proveer

información para el encargado de elegir el *framework* adecuado para la aplicación que se quiere desarrollar; responde la pregunta "¿Sirve el *framework* para mi propósito?". El segundo nivel está enfocado al uso normal del *framework* y responde a la pregunta "¿Cómo uso el *framework*?". Finalmente, el tercer nivel se centra en el diseño detallado del *framework*, respondiendo al problema de "¿Cómo funciona el *framework*?". El nivel de uso normal está documentado a través de los patrones de aplicación. Cada patrón es acompañado de un diagrama de flujo que describe los pasos de aplicación del patrón, pudiendo tener diversos diagramas de clases y colaboración para documentarlo. Para el nivel detallado, proponen el uso de patrones de diseño, complementados tal vez con técnicas más formales y utilización de herramientas de depuración e ingeniería reversa.

Recientemente ha habido un esfuerzo creciente orientado a dotar de más formalidad a diversas técnicas de documentación. El objetivo es brindar una comunicación menos ambigua entre los expertos, así como permitir la asistencia de herramientas CASE en el proceso de diseño e instanciación. Este es el caso de la técnica para una precisa especificación visual de los patrones de diseño [LK98]. En este enfoque, las descripciones de patrones son divididas en tres modelos (roles, tipos y clases) y cada modelo es documentado utilizando *UML* [Rat97], extendido con avances recientes en notaciones visuales para alcanzar una mayor precisión.

Similarmente, Soundarajan [Sou99] presenta un enfoque basado en la traza (*trace-based*) para especificar el comportamiento de *frameworks*, en particular el flujo de control. Esta técnica formal no es propuesta para ser usada aisladamente, sino como un complemento de técnicas más informales, como por ejemplo el estudio de aplicaciones existentes (ejemplos).

2.4. Técnicas Orientadas a Herramientas

Otro enfoque para la documentación de *frameworks* está representado por la técnica denominada **ejemplares** (*exemplars*) [GM95]. Un ejemplar es un modelo visual ejecutable, consistente de instancias de clases concretas y representaciones explícitas de sus colaboraciones. Para cada clase abstracta del *framework*, por lo menos una de sus subclases concretas es instanciada en el ejemplar. Esta técnica está orientada a proveer asistencia al proceso de instanciación del *framework*. El usuario crea la nueva aplicación adaptando gradualmente los ejemplares de acuerdo a los requerimientos de la aplicación, pudiendo visualizar en cada paso el resultado de las modificaciones. No obstante la utilidad de comenzar a partir de una aplicación existente, este enfoque presenta algunos problemas. Los modelos visuales son difíciles de construir y existen grandes límites en qué puede ser hecho a través de ellos. Además, este tipo de documentación tiene problemas similares a los libros de recetas: no proveen explicaciones sobre las razones de diseño y no resultan útiles para usos del *framework* que van más allá de lo previsto por el diseñador.

2.4.1. Libros de recetas interactivas

Actualmente una de las técnicas que más asistencia proveen a los usuarios de un *framework* para la creación de nuevas aplicaciones son los libros de recetas interactivas [PPSS95], herramientas que proveen asistencia semiautomática al proceso de instanciación. Un libro de recetas interactivas es capaz de ejecutar las descripciones de las recetas, proveyendo al usuario una interfaz interactiva que lo guía a través del proceso de instanciación. Estos libros se diferencian de los libros de diseño en que se centran en los detalles de implementación de una adaptación. De todas formas, aunque los libros de recetas y los libros de diseño sean conceptualmente diferentes, están predestinados a ser integrados en una herramienta [Pre95b].

Estos libros se basan en la idea de que la especialización de un *framework* siempre tiene lugar a través de sus puntos de flexibilización (*hot-spots*). El usuario debe encontrar la receta que es apropiada para una adaptación específica; esta receta es luego utilizada simplemente siguiendo los pasos que describen cómo llevar a cabo cierta tarea de adaptación. La Figura 3.8,

por ejemplo, muestra los diversos pasos de una receta para crear un nuevo dispositivo de comunicaciones y la activación de un editor de recursos a través de esa receta.

Las recetas pueden ser utilizadas para disparar herramientas que asistan en puntos específicos del proceso de instanciación. En el ejemplo de la Figura 3.8 un editor de menús es utilizado en un paso del proceso de creación de un nuevo dispositivo.

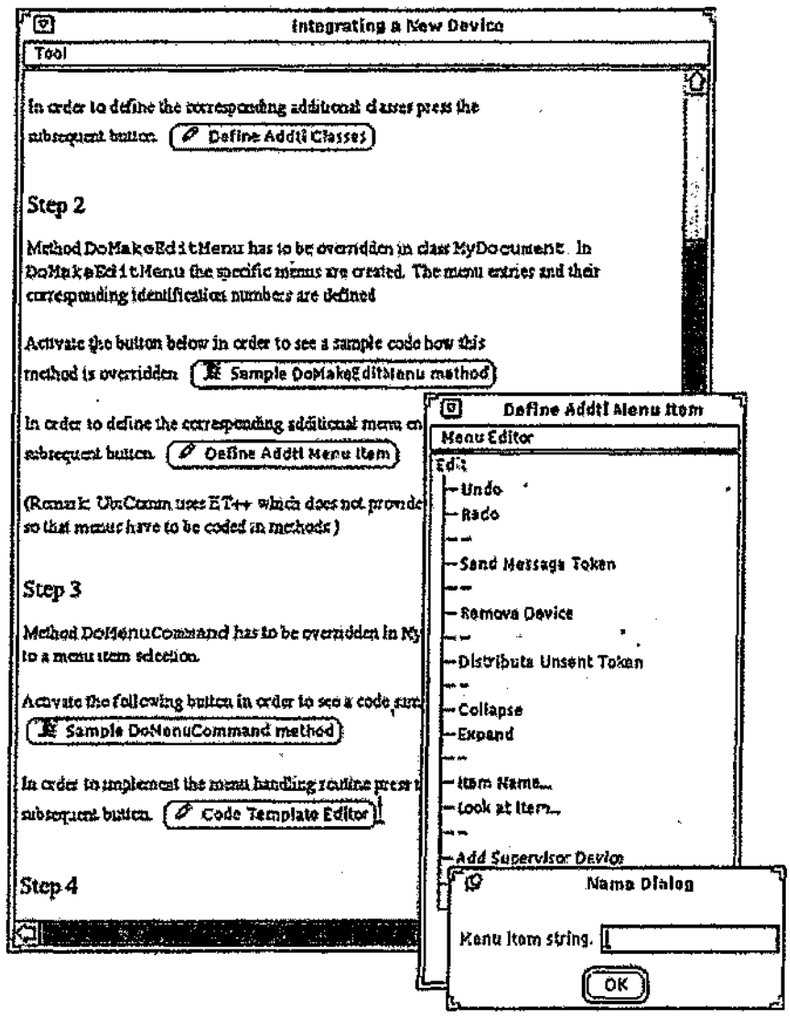


Figura 3.8 Activación de un editor de recursos utilizando una receta activa.

Los libros de recetas interactivas son un paso hacia la solución del problema de asistencia al proceso de instanciación de *frameworks*. Un aspecto importante es que buscan centrarse en la funcionalidad que se quiere obtener de la aplicación desarrollada. Esto permite al usuario concentrarse en qué debe hacer la aplicación, no prestando tanta atención, en principio, a cómo implementar esa funcionalidad. A partir de esto genera guías concretas de los pasos a seguir para obtener la funcionalidad. Este tipo de ayuda facilita la implementación de la aplicación porque las personas son buenas para seguir indicaciones paso a paso.

Los libros de recetas interactivas, sin embargo, presentan algunas limitaciones que restringen en gran medida su aplicación práctica. Si bien resultan adecuados para especificar eficientemente ciertos aspectos de la adaptación de *frameworks*, uno de los problemas fundamentales de los que adolecen es la poca flexibilidad que proveen. Así, al utilizarlos, el

usuario debe seguir todos y cada uno de los pasos de una receta, o bien optar directamente por no tener ninguna asistencia.

Este problema de flexibilidad también se manifiesta en la poca capacidad de asistir en situaciones donde el usuario quiere especializar el *framework* en alguna forma que no fue anticipada por los diseñadores o autores de la documentación. Es decir, los libros se adaptan bien a situaciones donde el usuario realizará las especializaciones a través de los *hot-spots* provistos por el *framework*, pero carecen de asistencia en los casos en que el usuario necesite un tipo de adaptación no previsto. En este sentido, Pree y otros [PPSS95] proponen resolver el problema mediante la utilización de relaciones estructurales, las cuales, conceptualmente, capturan las interacciones entre componentes del *framework*. Estas relaciones hacen explícita información sobre la comunicación entre componentes que está usualmente distribuida en el código de las clases, proveyendo puntos (*placeholders*) donde insertar nuevos componentes. De esta forma, nuevos componentes agregados al *framework* deben respetar el protocolo establecido por la relación estructural en la cual son insertados. Este tipo de mecanismo, si bien permite agregar al *framework* nuevos componentes no previstos, no se adapta realmente a situaciones nuevas. El problema sigue siendo la incapacidad de este tipo de herramientas de asistir parcialmente en el proceso de instanciación. La situación debe estar totalmente prevista o no se puede brindar asistencia alguna.

Un problema adicional de los libros de recetas interactivas es que están diseñados para ser utilizados a través de herramientas específicas para cada tipo de adaptación necesaria. Este mecanismo es poco genérico, porque no se puede aplicar a todos las adaptaciones que pueden ser necesarias en los *frameworks* en general y porque limita más aún la capacidad de adaptación de la herramienta a situaciones no previstas. Si bien en algunos casos se pueden utilizar herramientas genéricas, estas quedan limitadas a situaciones simples, como editar una jerarquía en forma de árbol o los menús de una interfaz a usuario, no siendo un mecanismo que se pueda generalizar para situaciones más complejas.

Finalmente, otro importante factor que limita la utilidad de estos libros es el alto costo de su construcción. El autor del libro debe programar el comportamiento específico para cada receta, como por ejemplo el tipo de interacción que tendrá con el usuario y la generación de código correspondiente. Esto mismo hace muy difícil la extensión de la documentación, sobre todo a partir de la funcionalidad agregada al *framework* a través de las distintas instanciaciones.

2.5. Recapitulación de las Técnicas Vistas

Analizando las técnicas descritas, puede observarse un acuerdo general en que una buena documentación para un *framework* debe proveer más de una forma para representar el diseño del mismo. Técnicas como ejemplares y libros de recetas (incluyendo las activas) son buenas para proveer asistencia, pero la carencia de información de diseño reduce su utilidad para apoyar la adaptación del *framework* en formas no previstas. Por otro lado, enfoques más pasivos, especialmente aquellos que combinan más de una técnica de documentación, proveen mayor flexibilidad, a costa de imponer una mayor carga sobre el usuario del *framework*. Este esfuerzo adicional es más evidente con usuarios novatos, quienes tienen que invertir una gran cantidad de tiempo y esfuerzo leyendo documentación antes de ser capaces de construir la aplicación. Así, es necesario proveer mayor flexibilidad para que la documentación pueda asistir a usuarios con diferentes objetivos.

Teniendo en cuenta que la audiencia de la documentación de un *framework* está compuesta principalmente por usuarios que seleccionan el *framework*, los programadores de aplicaciones típicas y los usuarios que deben modificar la estructura del *framework*, el mejor enfoque parece ser el uso de modelos que combinan documentación activa con explicaciones detalladas de diseño, tanto formales como informales. Aunque esto haya sido propuesto en algunos de los trabajos de la literatura, especialmente aquellos construidos alrededor de un sistema hipermedia, ninguno de ellos describe cómo debe realizarse la combinación, más allá

del uso de vínculos de hipermedia para relacionar las diferentes partes de la documentación. Esto no es suficiente si se tiene en cuenta la complejidad derivada de la cantidad de información involucrada en la documentación de un *framework*. Son necesarios nuevos enfoques que permitan estructurar y organizar las distintas técnicas de documentación de forma tal que puedan ser presentadas al usuario más eficientemente.

Aún disponiendo de documentación apropiada, comenzar a utilizar un *framework* para construir aplicaciones puede ser una tarea extremadamente costosa. Por esto es necesario un mecanismo de instanciación de alto nivel, que brinde mayor asistencia al usuario y le permita construir aplicaciones sin tener que conocer el funcionamiento del *framework*. En la próxima sección son analizados algunos enfoques orientados a proveer herramientas que ofrezcan estos mecanismos de instanciación.

3. Herramientas de Apoyo a la Instanciación

Un camino alternativo para apoyar el proceso de instanciación de *frameworks*, además de las técnicas de documentación, es la utilización de herramientas específicas. En el límite entre ambas alternativas se encuentran algunas formas de documentación, las cuales poseen características que permiten considerarlas, bajo ciertos aspectos, herramientas de apoyo a la instanciación. Tal es el caso de los libros de recetas interactivas, los cuales por un lado son una forma de documentar el *framework* y por otra son una herramienta que asiste activamente en su instanciación.

Ya considerando herramientas de instanciación propiamente dichas, un ejemplo simple lo constituye el *Tool Builder* provisto con el *framework HotDraw* y analizado en la sección § II.2.3.2. Esta herramienta guía la instanciación de una parte de *HotDraw*, siendo utilizado el mecanismo de composición como método de especialización (ver figura 2.9). A través del *Tool Builder*, el usuario del *framework* puede definir nuevos *Tools*, asociándoles los *readers*, *commands* e iconos necesarios. Estas nuevas herramientas luego deben ser vinculadas con el resto de la aplicación desde el código del editor que se está construyendo.

3.1. Constructores de Interfaces de Usuario

El tipo de herramienta de instanciación más difundido son los generadores de aplicaciones (*application builders*), a veces también llamados *entornos visuales de programación* o *entornos con soporte para programación visual* [IBM99, JLCH99, OH98, SP99]. Típicamente, estas herramientas son aplicaciones GUI, es decir, aplicaciones para construir interfaces de usuario, generalmente basadas en cuadros de diálogos. Usualmente presentan una paleta de componentes disponibles, desde la cual el diseñador de un programa puede arrastrar ítems y colocarlos en un formulario o ventana cliente. Una vez que la interfaz ha sido construida de esa manera, puede ser vinculada con el resto de la aplicación, la cual es construida utilizando herramientas normales de compilación. Para ayudar en esta vinculación, la herramienta muchas veces genera esqueletos de código, que luego deben ser completados por el programador. En estas herramientas, el único aspecto del desarrollo que puede ser hecho visualmente es la distribución física de los componentes en la interfaz (*graphic layout*).

Un ejemplo sencillo de este tipo de herramientas es el entorno *VisualWorks* para *Smalltalk* [Par94]. *VisualWorks* provee herramientas para la creación de la interfaz de una aplicación, pudiéndose colocar en esta interfaz elementos estándares, como por ejemplo *action buttons, check buttons*, listas, etiquetas, etcétera, como muestra la Figura 3.9. Para cada uno de estos elementos es posible definir propiedades, como por ejemplo el mensaje que debe enviar a la aplicación para tomar el valor a mostrar o qué mensajes enviará al producirse determinados eventos. La Figura 3.10 muestra, por ejemplo, el cuadro de propiedades asociado a un campo de entrada (*input field*), donde se especifica que cuando se produzca el evento de cambio, se deberá enviar el mensaje *accessorsChanged* a la aplicación asociada.

Esta herramienta genera automáticamente algunas porciones de código. Por ejemplo, genera código de inicialización para los componentes; cuando el elemento es creado, se le solicita a la aplicación (modelo) el valor que debe mostrarse, utilizando para ello el mensaje definido como propiedad del elemento. Es responsabilidad del programador proveer para la aplicación un método con esa signatura, que retorne el valor apropiado.

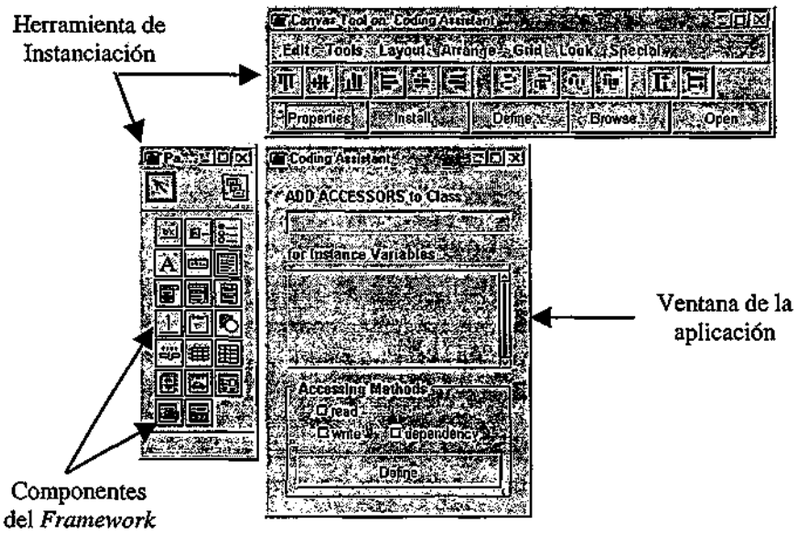


Figura 3.9 Creación de la interfaz de una aplicación con *VisualWorks*

Este tipo de herramientas se basa, generalmente, en un *framework* de interfaz a usuario subyacente. Según la herramienta, este *framework* puede ser "escondido" al usuario en mayor o menor medida. Cada tipo de componente de la paleta de opciones corresponde con una clase de componentes del *framework*. Cuando el usuario usa una de estas herramientas para crear una interfaz, lo que se hace es instanciar el *framework* subyacente. Es decir que estas herramientas son verdaderos entornos de instanciación de *frameworks*, donde el objetivo es que el usuario no tenga que conocer el *framework* en profundidad para ser capaz de crear aplicaciones.

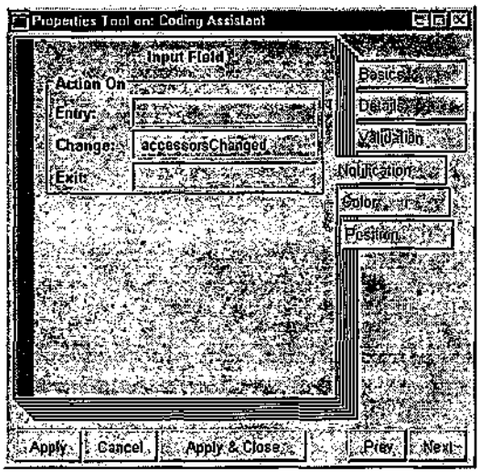


Figura 3.10 Especificación de propiedades de los elementos de la interfaz

Desde el punto de vista de la instanciación de *frameworks*, el mayor problema de estas herramientas es que cada una asiste en la creación de aplicaciones a partir de un *framework* específico, no pudiendo ser utilizada para la instanciación de otro *framework*, aún dentro del mismo dominio.

3.2. Constructores Genéricos de Aplicaciones: *Java Beans*

Si bien las interfaces de usuario son el tipo de aplicación más comúnmente construido con estos entornos, no necesariamente están limitados a este dominio. Existen entornos desarrollados más recientes que intentan ofrecer un gama más amplia de aplicaciones. Además del diseño visual de la interfaz de la aplicación, estos generadores de aplicaciones permiten, a través de la interfaz gráfica, conectar visualmente los componentes, seleccionar los eventos que deben ser disparados y los manipuladores de eventos que poseerá cada componente.

Un paso importante para la utilización de este tipo de generadores de aplicaciones ha sido el desarrollo de los *Java Beans* [IBM99, OH98]. Un *Java Bean* es un componente de software reutilizable escrito en código Java, diseñado para ser manipulado visualmente en herramientas de construcción de aplicaciones. El objetivo de los *Java Beans* es no sólo permitir la construcción de aplicaciones a través de la manipulación visual de componentes, sino que el conjunto de componentes que pueden ser utilizados no esté acotado.

Lo que diferencia los *Beans* de las clases *Java* normales es la introspección. Son construidos de forma tal que las herramientas que reconozcan patrones predefinidos en las signaturas de los métodos y en las definiciones de clases pueden inspeccionar un *Bean* para determinar sus propiedades y comportamiento. El estado de un *Bean* puede ser manipulado en el momento de ensamblarlo como parte de una aplicación mayor (*design time*). Para que este sistema funcione, las signatura de los métodos dentro de un *Bean* deben seguir cierto patrón, de forma que las herramientas de introspección puedan saber cómo los *Beans* pueden ser manipulados, tanto en tiempo de diseño como de ejecución.

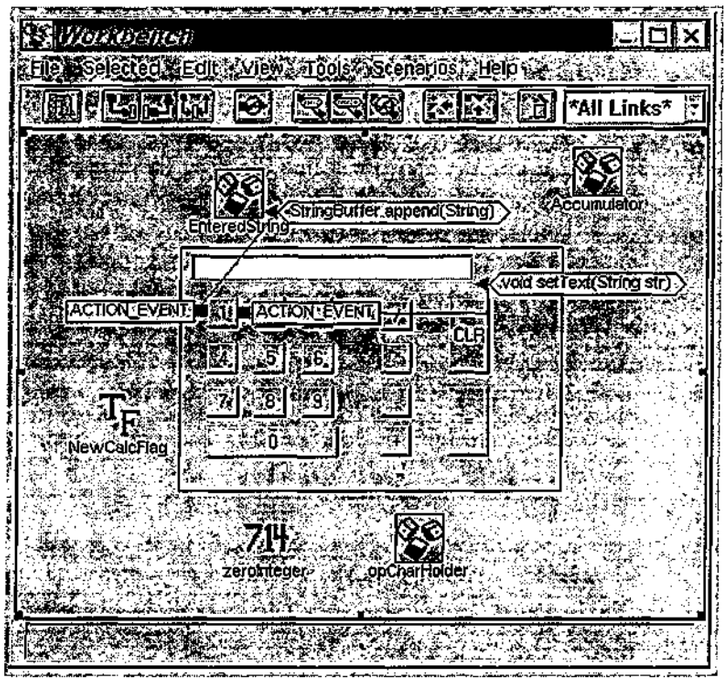


Figura 3.11 Creación de aplicaciones utilizando *Java Beans*

Un ejemplo de la utilización de *Beans* para la construcción de aplicaciones se presenta en la Figura 3.11. En este ejemplo, se muestra como puede ser creado un programa para una calculadora sólo mediante la conexión visual de componentes; los eventos generados y los métodos manipuladores de eventos son seleccionados a través de menús emergentes (*pop-up*). No todos los componentes que pueden ser manipulados a través de estos constructores de aplicaciones tienen necesariamente una representación gráfica en la interfaz de la aplicación. El ejemplo de la Figura 3.11 muestra acumuladores y contenedores de datos que son utilizados por

la aplicación, pero que no tienen una representación visible. La manipulación de las propiedades y eventos asociados a cada elemento se realiza en forma similar a lo descrito para *Visual Works*; es decir, se utilizan ventanas de diálogos (llamadas *properties sheets*) que permiten ver y cambiar las propiedades de cada elemento.

Una característica importante de los constructores de aplicaciones basados en *Java Beans* es que no están diseñados sólo para un conjunto acotados de componentes. Cualquier componente *Java* que cumpla un cierto protocolo es un *Java Bean* y puede ser manipulado por estas herramientas. Por ejemplo, este protocolo establece que por cada propiedad que deba ser manipulada por las herramientas de construcción de aplicaciones, se deben definir dos métodos, uno para solicitar al componente el valor de la propiedad y otro para modificar ese valor. Entonces un *Java Bean* que defina una propiedad color, por ejemplo, debe implementar un método *getColor* y otro *setColor*. Esto implica que, en cierta medida, es posible construir herramientas genéricas para apoyar el proceso de instanciación de *frameworks*, es decir, que puedan ser utilizadas con distintos *frameworks*.

4. Conclusiones

El desarrollo de aplicaciones orientadas objetos por medio de la reutilización de *frameworks* ofrece indudables ventajas con respecto al desarrollo de una aplicación partiendo desde cero y aún respecto del desarrollo basado en bibliotecas de clases. Entre las ventajas más importantes se ha mencionado la posibilidad que ofrecen los *frameworks* orientados a objetos de reutilizar no sólo código, sino también un diseño genérico para aplicaciones del domino de la aplicación. De esta forma, lo que se reutiliza es la experiencia de diseñadores experimentados y expertos en el dominio dado, permitiendo al programador de la aplicación concentrarse en las características particulares de ésta.

Sin embargo, también se han identificado algunos problemas relacionados con el desarrollo basado en *frameworks*. Desde el punto de vista del usuario del *framework*, el mayor problema es la dificultad asociada a la comprensión del mismo. A la complejidad inherente a los programas orientados a objetos deben sumarse las características propias de los *frameworks*, como son su diseño abstracto y muchas veces incompleto, su generalidad y flexibilidad, factores todos que juegan en contra de la comprensibilidad del *framework*. En particular, resulta muy difícil, sobre todo para usuarios inexpertos o novatos, determinar qué debe ser hecho para instanciar una nueva aplicación utilizando el *framework*.

Existe un consenso general entre los expertos en documentación de *frameworks* en que la utilización de técnicas aisladas no es suficiente, sino que es necesario documentar los *frameworks* con combinaciones de distintas técnicas y notaciones. Esto se debe sobre todo a la necesidad de brindar apoyo a distintos tipos de usuario, con diferentes objetivos y niveles de conocimientos. Por este motivo, los más recientes trabajos de documentación se centran en combinaciones de técnicas formales e informales, de técnicas orientadas a explicar cómo implementar una aplicación y técnicas que explican el diseño y funcionamiento del *framework*.

Considerando las necesidades del programador de aplicaciones, lo más importante, al menos en un principio, es que se le indique cómo implementar una determinada funcionalidad utilizando el *framework*, sin necesidad de tener que invertir gran cantidad de tiempo en conocer el diseño del mismo. Los enfoques basados en recetas son un primer paso en este sentido. Ellos intentan aislar al usuario de las cuestiones de diseño permitiéndole concentrarse en qué debe ser hecho para tener su aplicación implementada. Sin embargo, estos enfoques han demostrado ser aún insuficientes. Sus mayores problemas se relacionan con la falta de mecanismos para adaptarse a situaciones no previstas y para ser extendidos a partir del trabajo de los usuarios del *framework*, así como también la falta de un plan general de trabajo o información de contexto en las recetas. Las recetas están basadas en información estática y muchas veces su comportamiento debe ser específicamente programado. Por todo esto, producir un libro de recetas, y más aún un libro de recetas interactivo, resulta extremadamente costoso y los

resultados no son del todo satisfactorios. Además, la falta de información global sobre los objetivos del usuario obliga a la aplicación de cada receta en forma individual, no permitiendo aprovechar información de otras recetas aplicadas para la misma aplicación. Esto impide, por ejemplo, proveer asistencia para resolver eventuales conflictos que podría surgir durante la aplicación de las recetas.

Un ejemplo sencillo de las limitaciones ocasionadas por la falta de información global puede ser visto en el *framework HotDraw*. Cuando un objeto editado tiene atributos, estos atributos pueden ser modificados interactivamente a través del uso de menús emergentes, herramientas (*tools*) o manipuladores (*handlers*). Una receta para agregar la funcionalidad de edición a un atributo ofrecerá estas opciones al usuario, sin tener en cuenta el método utilizado para la edición de otros atributos, si existen. Esto puede llevar a situaciones en las que, por ejemplo, en un diagrama de clases el nombre de las clases se edita utilizando un menú y el nombre de los métodos pinchando directamente sobre su representación gráfica y escribiendo allí el nuevo nombre. Como un aspecto deseable en una interfaz a usuario es la consistencia de manipulación, sería más conveniente que la herramienta de apoyo a la instanciación tuviera en cuenta que se quieren editar los nombres de las clases y los métodos, y orientara al usuario a utilizar el mismo mecanismo para ambas situaciones. Para esto es necesario que la herramienta posea conocimiento global de la funcionalidad requerida para la aplicación, y no trabajar cada vez sobre porciones aisladas de esta funcionalidad.

Otro tipo de asistencia ofrecida para la instanciación de *frameworks* son los llamados constructores de aplicaciones. La mayor parte de estos sistemas proveen un entorno en el cual el usuario puede crear aplicaciones basadas en un *framework* específico. Para ello, ofrecen herramientas especialmente diseñadas para adaptar ese *framework*, muchas de las cuales permiten la utilización de notaciones gráficas para especificar las adaptaciones. Además, generalmente son capaces de generar código con el esqueleto de la aplicación, debiendo el usuario en algunos casos sólo completar pequeñas porciones específicas. En general, este tipo de entornos han tenido éxito dentro del dominio de las interfaces de usuario, no existiendo ejemplos importantes en otros dominios. Más aún, el gran problema que presentan es su completa falta de generalidad: el creador de un *framework* que quiera una herramienta de este tipo para asistir en su instanciación, debe programarla específicamente. Por otro lado, estas herramientas también obligan al usuario a utilizar el *framework* de la forma prevista u optar por desarrollar todo su trabajo sin asistencia.

Un esfuerzo importante, tendente a permitir la construcción visual de aplicaciones sin restringirse a un *framework* específico, son los *Java Beans*. En base a los protocolos definidos para los *Java Beans*, se pueden diseñar herramientas que permitan la construcción de aplicaciones mediando la composición visual de los *Beans*. Además, cualquier clase o conjunto de clases *Java* que respeten ese protocolo puede ser manipulado por esas herramientas, permitiendo utilizar, por ejemplo, componentes definidos por el usuario o comprados a terceros. Pero este enfoque también presenta problemas:

- los componentes deben ser programados siguiendo un protocolo determinado y por lo tanto la técnica no se puede aplicar a *frameworks* ya existentes
- el tipo de aplicación que se puede construir a través de composición visual es bastante limitado; el enfoque sigue siendo adecuado principalmente para aquellas aplicaciones donde la mayor parte de la funcionalidad está en la interfaz de usuario.

4.1. Resumen de Requisitos

Analizando los requisitos del proceso de instanciación de *frameworks* y las herramientas y técnicas para asistir ese proceso existentes, puede observarse la necesidad de disponer de herramientas más avanzadas, que permitan al usuario del *framework* concentrarse en la

funcionalidad requerida y en cómo implementar esa funcionalidad, conociendo el diseño del mismo sólo en la medida en que sea necesario, pero sin restringirlo excesivamente en su trabajo.

Una herramienta de apoyo a la instanciación de *frameworks* es esencial que provea asistencia adecuada para cada tipo de usuario y de especialización requerida:

- Usuarios novatos, que sólo quieren utilizar el *framework* de la forma prevista y que no necesitan extenderlo de ninguna manera: la orientación debe permitirles concentrarse en la funcionalidad que requieren de su aplicación, llevándolos a realizar los pasos necesarios para la implementación sin necesidad de que se impliquen en detalles del diseño del *framework*.
- Usuarios con experiencia, que necesitan ir más allá de las adaptaciones previstas para el *framework*, extendiéndolo de formas no anticipadas por los diseñadores. En este caso, la guía provista por el entorno debe ser tan completa como sea posible con la información disponible, dejándole al usuario la posibilidad de analizar alternativas o escoger caminos distintos a los previstos en la documentación. La asistencia debe ser lo suficientemente flexible para adaptarse, dentro de los posible, a las acciones del usuario y no dejarlo totalmente desprovisto de guía.

Esta herramienta, por ejemplo, debiera permitir al usuario seleccionar la funcionalidad deseada para su aplicación y producir indicaciones que lo guíen para obtener esa funcionalidad utilizando el *framework*. Otro factor importante es que la herramienta de asistencia no considere cada porción de la funcionalidad requerida de forma aislada, sino que sea capaz de ofrecer una visión global de las actividades necesarias para implementar la aplicación. Así mismo, debe ser capaz de ofrecer cursos de acción alternativos y resolver eventuales situaciones de conflictos entre las soluciones propuestas para implementar los distintos objetivos.

Finalmente, es necesario que la herramienta sea genérica. Esto quiere decir que no debe estar diseñada para un *framework* específico, sino que debe poder ser utilizada para la instanciación de distintos *frameworks*. Más aún, los *frameworks* deben poder ser de distinto tipo, y no estar restringidos al dominio de las interfaces de usuario o *frameworks* que pueden ser instanciados a través de la composición visual de sus componentes.

En los próximos capítulos se presentará una propuesta para la construcción de herramientas que brinden mayor asistencia al proceso de instanciación de *frameworks*.

IV SmartBooks: Asistiendo la Instanciación de Frameworks

A partir del análisis efectuado en los capítulos anteriores, se concluyó que las herramientas disponibles no son suficientes para las necesidades de los usuarios de *frameworks* orientados a objetos. Los requerimientos de estos usuarios, que varían de acuerdo a su nivel de experiencia y conocimientos, así como también al tipo de adaptación o extensión que necesiten realizar para implementar la aplicación, no son adecuadamente satisfechos por las herramientas actuales.

En este capítulo se presenta *SmartBooks*, un método para asistir a los usuarios de *frameworks* en el proceso de instanciación de una aplicación. Este método está basado en técnicas de planificación (*planning*) que, partiendo de la funcionalidad requerida para la aplicación, permiten elaborar un plan de trabajo, es decir, una secuencia de las actividades que debería realizar el usuario para implementar su aplicación. Utilizando estas técnicas, *SmartBooks* es capaz de guiar el trabajo de instanciación, centrando esta guía en la funcionalidad que se quiere implementar. Una característica importante de *SmartBooks* es que la asistencia provista es flexible, pudiendo ser adaptada, en cierta medida, a las necesidades del usuario. En ciertos casos es posible guiar al usuario cuando lleva a cabo actividades no planificadas, o las ejecuta de forma distinta a la prevista. Para esto el *framework* es documentando utilizando dos tipos de reglas: reglas funcionales, que proveen el conocimiento necesario para construir planes de instanciación, y reglas de consistencia, que ofrecen guías para la especialización del *framework* en situaciones arbitrarias.

Con el objetivo de introducir *SmartBooks*, el capítulo comienza con un ejemplo concreto de las necesidades de un creador de aplicaciones basadas en *frameworks* orientados a objetos, explicándose el tipo de asistencia que se pretende proveer para satisfacer esas necesidades. A continuación, es realizado un análisis de los requisitos necesarios para proveer esa asistencia, introduciéndose los conceptos básicos en los que se fundamenta la propuesta de *SmartBooks*: técnicas de planificación y modelos de tareas de usuario. Finalmente, la propuesta es desarrollada, mostrando cómo las técnicas de planificación son utilizadas para generar los planes de instanciación, y el concepto de tarea de instanciación es utilizado para estructurar dichos planes. También se explica el tipo de documentación que debe proveer el diseñador del *framework* para posibilitar este tipo de asistencia, en forma de reglas de instanciación. El capítulo finaliza con una descripción sobre la forma en que esta información puede ser utilizada en un entorno de instanciación de *frameworks*.

1. Desarrollo de Aplicaciones Basado en *Frameworks*

Para analizar los requisitos de una herramienta de apoyo al proceso de instanciación de *frameworks*, deben tenerse en cuenta las características de dicho proceso. Un *framework* es una solución genérica para una familia de aplicaciones. Implementar una aplicación utilizando un *framework* requiere que las características particulares de esta aplicación sean especificadas de alguna forma. Es decir, la solución genérica debe ser adaptada a las particularidades de la aplicación. Por este motivo, el usuario debe determinar qué debe ser hecho con el *framework* para implementar esta adaptación. Esto no implica, mayormente, una labor creativa, sino que el usuario debe comprender cómo está previsto que el *framework* sea utilizado para implementar la funcionalidad requerida.

El proceso de producir una aplicación utilizando un *framework* basado en herencia consiste, generalmente, en una secuencia de tareas de dos tipos: crear clases y proveer implementaciones para métodos de estas clases [FSJ99]. Idealmente, las nuevas clases deberían ser subclases de clases del *framework* y los nuevos métodos sólo deberían redefinir métodos

abstractos y/o *hooks* [Jol99]. Esto se cumple, generalmente, cuando el *framework* ha alcanzado un nivel de maduración adecuado para proveer soporte a un número amplio de aplicaciones dentro del dominio. En otras oportunidades, si el *framework* no se adapta bien a las necesidades de la aplicación, las modificaciones deben ser mayores pero, aún así, estas modificaciones estarán basadas en tareas de creación de clases y métodos.

En los *frameworks* basados en composición, por otro lado, el usuario adapta el *framework* utilizando distintas combinaciones de objetos. Esto es hecho a través de la implementación de uno o más métodos y, en ciertas ocasiones, la utilización de herramientas específicas. Es decir que, generalmente, instanciar un *framework* consiste en llevar a cabo una combinación de especializaciones de clases, implementaciones de métodos y composiciones de instancias de clases del *framework*.

No obstante, a pesar de que el proceso de instanciación comprende sólo actividades de tres tipos, determinar en cada caso qué clases deben ser especializadas, qué métodos implementados y qué comportamiento es esperado de estos métodos, es una tarea que puede resultar compleja, principalmente para usuarios inexpertos.

En el capítulo II se analizó un ejemplo sencillo de utilización del *framework HotDraw*[1] para la creación de un editor gráfico que permita editar líneas, rectángulos, círculos y triángulos. En ese ejemplo, el usuario debía definir una nueva clase (*TriangleFigure*) y proveer una implementación para dos métodos abstractos, *extent* y *draw*. Además, debía ser definida una subclase de *DrawingEditor* que cree la estructura de objetos que efectuarán el trabajo en tiempo de ejecución.

En otros casos, a pesar de que la aplicación sea simple y el *framework* adecuado para su implementación, será necesaria una serie más elaborada de acciones para instanciar la aplicación. Por ejemplo, supóngase que se quiere crear un editor para diagramas PERT, donde existen nodos que representan tareas, las cuales están unidas por relaciones de precedencia. Cada tarea tiene una duración y una fecha de finalización. La duración debe ser editada a través de la interfaz a usuario, mediante manipulación directa, y la fecha de finalización es calculada a partir de la información de tareas precedentes y la duración de la propia tarea. En este ejemplo, es relativamente sencillo determinar que se debe definir una subclase de *Figure* para representar las tareas y una subclase de *DrawingEditor* para crear las instancias correspondientes. Sin embargo, también se debe resolver, por ejemplo, cómo implementar la edición interactiva de la duración de cada tarea, y permitir luego que ese valor sea usado para calcular la fecha de finalización de la tarea. *HotDraw* provee los medios necesarios para implementar esta funcionalidad, por lo tanto el trabajo del programador consiste en utilizar esos medios de forma adecuada.

Para implementar, parcialmente, esta funcionalidad, el usuario del *framework* debe representar tanto la duración como la fecha de finalización de la tarea a través de instancias de una clase provista por *HotDraw*, *NumberFigure*; representar la fecha de inicio con una variable, instancia de la clase *HotDrawVariable*; y finalmente crear restricciones que vinculen las distintas variables. El lugar donde este código debe ser colocado es en el método de inicialización de la figura que representa la tareas y se deben tener en cuenta algunos detalles, como por ejemplo que las restricciones pueden ser creadas y asociadas a las tareas sólo luego de que las correspondientes variables han sido inicializadas. La Figura 4.1 muestra el código simplificado del método de inicialización de la clase *PertTask*. Este método recibe como parámetro el punto de origen de la figura (extremo superior izquierdo). Primero crea una figura para representar la duración; luego otra para la fecha de finalización; después crea e inicializa la variable de la fecha de inicio; a continuación compone la figura y la muestra por pantalla; finalmente crea las restricciones: la primera establece que la fecha de inicio más la duración será igual a la fecha de fin, la segunda que la duración debe ser siempre mayor o igual a cero y la

[1] En el mismo capítulo dos es también presentada una descripción de este *framework*

tercera es utilizada para el cálculo de la fecha de inicio (a medida que se asocien tareas predecesoras, sus fechas de fin serán utilizadas para este cálculo).

```
initializeAt: aPoint
        | durationFigure endFigure |
        "Creación e inicialización de la figura que representa la duración de la tarea"
        durationFigure := NumberFigure number: 0 at: aPoint + (10 @ 10).
        "Creación e inicialización de la figura que representa la fecha de finalización"
        endFigure := NumberFigure number: 0 at: aPoint + (10 @ 20).
        "Creación e inicialización de la variable que representa la fecha de inicio"
        startDate := HotDrawVariable with: 0 owner: self.
        "Armado de la estructura de la figura"
        self setFigures: (OrderedCollection with: durationFigure with: endFigure).
        "Muestra la figura"
        self showVisibleAreaIndicator.
        "Creación e inicialización de las restricciones"
        HotDrawConstraint plus: startDate
                        and: durationFigure numberVariable
                        equal: endFigure numberVariable.
        HotDrawConstraint greaterThanEqualBetween: durationFigure numberVariable constant: 0.
        HotDrawConstraint maximize: startDate with: Array new default: 0
```

Figura 4.1 Ejemplo de especialización de un método en el *framework HotDraw*

Para terminar de implementar la edición interactiva de la duración, este código debe ser complementado con la adición al editor del *tool* correspondiente. Para finalizar la implementación del cálculo de fechas de finalización, por otro lado, el código asociado a la creación de vínculos debe proveer la información correspondiente (es decir, completar el vector utilizado en la última restricción creada).

Como es posible ver en este ejemplo, la instanciación del *framework* es una composición de tareas relativamente simples, las cuales sin embargo deben ser hechas de forma correcta y en el orden adecuado. La dificultad del proceso de instanciación se debe principalmente a la **determinación, selección y combinación** de las actividades que componen el proceso de creación de una aplicación específica. Específicamente, el problema consiste en determinar:

- qué actividades deben ser ejecutadas de acuerdo a la funcionalidad requerida para la aplicación. Es decir, cuál es el trabajo que se debe hacer para implementar dicha aplicación.
- cómo estas actividades interaccionan, es decir, cómo se afectan mutuamente, si se debe respetar algún orden entre ellas, condiciones para su ejecución, etc.
- cómo está previsto (en el diseño del *framework*) que estas actividades sean ejecutadas. Si se especializa un método, por ejemplo, el nuevo método debe hacer lo que se espera de él y debe respetar los protocolos definidos por el *framework*.

Es posible proveer al usuario dos tipos distintos de documentación para ayudarlo a resolver estas cuestiones. La primer alternativa es basarse en explicaciones de diseño e implementación. A partir de esta información, el usuario puede conocer y entender el *framework*, deduciendo en función de eso qué debe ser hecho para instanciar una aplicación. La documentación de este tipo es denominada **descriptiva** y presenta algunas desventajas considerando la perspectiva del usuario del *framework*. Ya se han explicado las dificultades que

un usuario debe enfrentar a la hora de entender un *framework*, agravadas por el hecho de que muchas veces no se trata de un experto. Un enfoque basado en el entendimiento del *framework* por parte del usuario no sólo sería excesivamente costoso en términos de esfuerzo y tiempo, sino que existe además la posibilidad de que este entendimiento sea parcial o erróneo, conduciendo a utilizaciones incorrectas del *framework* y soluciones de baja calidad.

Una alternativa preferible es proveer documentación **prescriptiva**. Este tipo de documentación está enfocada en cómo implementar la funcionalidad ofrecida por el *framework*, permitiendo profundizar en los detalles de diseño sólo en la medida en que esto sea necesario. La mayoría de los enfoques para documentación orientados a los programadores de aplicaciones son prescriptivos, alternativa también adoptada en este trabajo.

De acuerdo con este criterio, una herramienta de instanciación debe guiar al usuario utilizando descripciones procedurales, construidas en base a las actividades necesarias para implementar la aplicación; estas descripciones deben explicar uno por uno los pasos que el usuario debería seguir, sin explicar, en principio, las razones de diseño que determinan estos pasos. La información de diseño debe estar disponible para ser mostrada a requisición del usuario, o cuando sea necesario realizar actividades no contempladas en los procedimientos.

Basados en estos conceptos, para asistir al programador de aplicaciones es deseable una herramienta que:

- Le presente la funcionalidad que puede ser implementada utilizando el *framework*, permitiéndole elegir qué parte de esa funcionalidad es requerida para su aplicación
- De acuerdo a la funcionalidad elegida le muestre los pasos necesarios para crear la aplicación
- Utilizando esa información, lo guíe durante el proceso de instanciación

Para materializar esta asistencia, la herramienta debe conocer no sólo las actividades que deben ser ejecutadas, sino también las condiciones bajo las cuales se pueden ejecutar y las que se deben cumplir a su finalización, así como también información de contexto para orientar al usuario en la ejecución de cada actividad.

1.1. Implementación del Soporte

Una posible solución para proveer este tipo de asistencia es utilizar libros de recetas interactivos. No obstante, como ya fue analizado en el capítulo III, estas técnicas no satisfacen de forma adecuada los requisitos del usuario de un *framework*, siendo su principal problema la falta de flexibilidad y capacidad de adaptación. Para asistir más efectivamente al usuario, la capacidad de la herramienta de conocer y administrar la lista de actividades no debe redundar en detrimento de la flexibilidad. Esta flexibilidad es necesaria tanto para adaptarse a usuarios con distinto nivel de conocimiento como para apoyar el trabajo de usuarios que deben extender el *framework* más allá de lo previsto en la documentación.

La falta de capacidad de adaptación de los libros de recetas interactivos y enfoques similares, se debe a que los posibles caminos de instanciación son fijados en el momento de escribir la documentación. Es decir, los usuarios están limitados a seguir un modelo estático de instanciación, sin posibilidades de que la lista de actividades sugerida sea adaptada de alguna forma a sus necesidades específicas.

Para alcanzar el grado de asistencia y flexibilidad necesario, deben utilizarse herramientas más inteligentes, que sean capaces de determinar en forma dinámica la lista de actividades de instanciación adecuadas para implementar una funcionalidad determinada. Para ello debe tenerse en cuenta que el proceso de instanciación de *frameworks* puede ser considerado como una secuencia de actividades predefinidas. Teniendo la información acerca de qué actividades deben ser ejecutadas para implementar una funcionalidad dada, debe ser posible generar, al

menos, una secuencia parcial de esas actividades, que guíe al usuario en el proceso de instanciación. En base a ello, es posible tener un estrecho control de las actividades del usuario, conociendo la semántica detrás de ellas, sin perder la flexibilidad de adaptarse a distintas situaciones y necesidades.

Un ejemplo de este tipo de asistencia puede ser visto considerando nuevamente el caso de creación de tareas para los diagramas *PERT*. Aquí el usuario seleccionaría como parte de la funcionalidad requerida para la aplicación, el siguiente objetivo:

Establecer relaciones entre atributos de una figura

A partir de ese requisito, una herramienta de asistencia que posea el conocimiento adecuado establecerá que esa funcionalidad puede ser implementada en *HotDraw* utilizando el sistema de restricciones. Luego pedirá al usuario que elija el tipo de relación que quiere crear y en función de ello creará las actividades para la definición e inicialización de las variables correspondientes, creando por último las restricciones. Durante este proceso, solicitará al usuario información adicional, como por ejemplo el nombre de las variables, el tipo de valores que contendrán y si estos valores podrán ser modificados a través de la interfaz a usuario. Finalmente, la herramienta presentará la siguiente lista de actividades para ser ejecutadas (ver Figura 4.1):

1. DefineAttribute(starDate, PertTask)
2. AddLocalVar(PertTask, initializeAt:, durationFigure)
3. AddLocalVar (PertTask, initializeAt:, endFigure)
4. UpdateMethod(PertTask, initializeAt:, "durationFigure := NumberFigure number:0 at: aPoint.")
5. UpdateMethod(PertTask, initializeAt:, "endFigure := NumberFigure number:0 at: aPoint.")
6. UpdateMethod(PertTask, initializeAt:, "startDate := HotDrawVariable with: 0 owner: self.")
7. CreateConstraint(PertTask, initializeAt:, plus:and:equal, [startDate, durationFigure, endFigure])

La actividad 1 indica que el usuario debe definir un atributo (variable de instancia en la terminología Smalltalk) llamado *startDate* para la clase *PertTask*. Las restantes actividades (2-7) indican modificaciones que deben ser hechas sobre el método *initializeAt:* de la clase *PertTask*. Las actividades 2 y 3, por ejemplo, representan la definición de variables locales para ese método; la 4, 5 y 6 generan el código de inicialización de los atributos de la tarea; finalmente, la última actividad muestra la forma en que debería ser creada la restricción. La implementación final del método se completaría con la ejecución de actividades relacionadas con otros requisitos funcionales.

Debe notarse que todas estas actividades pueden ser resumidas en dos: definir un atributo para la clase *PertTask*, y modificar el método *initializeAt:* de la misma clase. Un herramienta encargada de generar esta lista de actividades puede detectar actividades cuyos objetivos estén superpuestos y presentarlas unificadas.

La inclusión de código en algunas de las actividades presentadas al usuario no deben confundirse con el concepto de programación automática. Este texto sólo describe patrones a los que, generalmente, responde el código correspondiente a la actividad. En casi todos los casos, estos patrones deben ser completados con detalles específicos de cada aplicación, que no pueden ser anticipados en el *framework* ni en su documentación, y que deben ser provistos por el programador de la aplicación.

Las principales ventajas de generar dinámicamente las listas de actividades de instanciación, con respecto a utilizar un modelo estático del proceso de instanciación, son:

- Disminuir el volumen de información que debe proveer el diseñador del *framework*: el diseñador (o el autor de la documentación, en caso de que no sea la misma persona) no debe proveer un modelo complejo de las acciones que el usuario debe

ejecutar para la instanciación, contemplando las combinaciones de adaptaciones que pueden ser necesarias. En vez de eso, debe proveer reglas que consideren cada posible adaptación de forma individual, estando la combinación a cargo del mecanismo de generación de la lista de actividades.

- Flexibilizar y extender la asistencia provista: la generación dinámica de las listas de actividades permite poder responder a la interacción del usuario. Por ejemplo, a requerimiento del usuario se pueden buscar caminos de implementación alternativos o se puede intentar reaccionar a acciones no previstas (por ejemplo, la redefinición de un método base). También es posible graduar la asistencia provista: si los objetivos del usuario no están totalmente previstos en el *framework* o su documentación, es posible generar listas parciales de actividades, que asistan en la medida de la información disponible.

Determinar dinámicamente la lista de actividades que deberían ser ejecutadas implica la utilización de algún tipo de mecanismo que permita, de acuerdo a la funcionalidad, generar o deducir esta lista a partir de información básica de documentación del *framework*. Este mecanismo debe permitir guiar activamente el trabajo del usuario, así como también reaccionar ante acciones no previstas.

2. Entorno de Instanciación

Analizando las características de la herramienta de apoyo a la instanciación de *frameworks* que se está proponiendo, surgen diversas cuestiones que deben ser resueltas. Para disponer de un entorno como el descrito, que sea capaz de guiar activamente al usuario de acuerdo a una lista de actividades generada a partir de información básica, sin perder flexibilidad, es necesario considerar:

- La forma en que la lista será generada. Se deben definir las técnicas y mecanismos que se utilizarán para producir la información que el usuario debería seguir para instanciar el *framework*.
- El tipo de guía que se dará al usuario. Es decir, es necesario definir cómo será la interacción del usuario con el sistema y cómo se le presentarán las actividades de programación que debería llevar a cabo. Estas decisiones pueden ser dependientes del mecanismo utilizado para generar el plan.
- La información en que se basará la herramienta para generar la lista de actividades. La definición del tipo de información que será necesario proveer (por parte de los encargados de documentar el *framework*) para permitir la generación de la lista está estrechamente relacionada con la selección del mecanismo que se utilizará para dicha generación.

De la elección de estos factores depende el nivel de asistencia que será posible alcanzar con la herramienta y la facilidad con que el usuario será capaz de instanciar un *framework* utilizándola. De estas decisiones depende también la complejidad de la documentación que deba ser provista para cada *framework*. A continuación estos factores serán estudiados con más detalle.

2.1. Derivación de la Lista de Actividades de Instanciación

Como ya se vio, es conveniente que la lista de actividades utilizada para guiar al usuario sea generada mayormente de forma dinámica, partiendo de los requisitos funcionales de la aplicación. Entonces, es necesario considerar el mecanismo que se utilizará para generar esta lista.

La derivación de cursos de acción en función de ciertos objetivos que se quieren alcanzar es un problema que puede ser resuelto utilizando técnicas de Inteligencia Artificial. Así, dependiendo de la técnica utilizada, la lista de actividades o plan de instanciación podría ser generada o deducida a partir de información básica, teniendo en cuenta los requerimientos funcionales. La técnica que será analizada en esta tesis es Planificación (*Planning*). Algunos enfoques utilizan otras técnicas de Inteligencia Artificial para asistir en la instanciación de *frameworks*, como el Razonamiento Basado en Casos [Lea96, Gon99].

2.1.1. Planificación

Planning es una técnica que, dado un objetivo y un conjunto de posibles acciones, elabora una secuencia de acciones que al ejecutarse permitirán alcanzar el objetivo deseado. Cuando se utiliza planificación, la solución es armada desde el principio, partiendo de descripciones de los pasos posibles que pueden ser dados en dirección a conseguir un objetivo determinado.

En el caso de la instanciación de *frameworks*, el objetivo es construir una aplicación que satisfaga los requisitos funcionales. El desarrollo es hecho a través de la adecuada combinación de actividades de programación, lo cual produce la aplicación final, o al menos las partes principales que puedan ser construidas utilizando el *framework*. Esta lista de actividades de programación, generada a través de técnicas de planificación, es denominada **plan de instanciación**.

A diferencia de otras técnicas de Inteligencia Artificial, la planificación no utiliza directamente conocimiento de anteriores experiencias. En contrapartida, produce soluciones más confiables, en el sentido que todo plan que genera proviene directamente de la información provista por el diseñador del *framework*. Esto es especialmente importante si se tiene en cuenta que la tecnología de *frameworks* asume que el diseñador tiene conocimiento y experiencia tanto en el dominio de aplicación como en las técnicas de diseño, mientras que del programador de aplicaciones sólo se requiere que sea capaz de programar las cuestiones específicas a su aplicación.

2.2. Interacción del Usuario con la Herramienta

Respecto al tipo de interacción que una herramienta de apoyo a la instanciación de *frameworks* debe tener con el usuario, a partir de análisis del capítulo anterior se ha establecido que la ayuda provista debe ser orientada por la funcionalidad y procedural. Es decir, la herramienta debe estar enfocada a mostrar cómo hacer el trabajo, dejando la explicación de diseño sólo como información accesoria, para ser accedida a petición del usuario o cuando el tipo de especialización del *framework* que se debe realizar así lo requiera.

También es necesario considerar el grado de adhesión respecto al plan o lista de actividades previstas que se requerirá del usuario. Una alternativa, por ejemplo, es permitirle actuar libremente y luego verificar que sus acciones se correspondan con lo previsto en el plan de instanciación. En este caso la herramienta sólo reaccionaría ante las acciones del usuario. En el otro extremo, se puede acotar la actividad del usuario a que sólo pueda seleccionar y ejecutar acciones que están en el plan, no pudiendo hacer nada que se aparte del mismo.

Como se ha destacado, es importante alcanzar un buen balance entre la supervisión que se haga de las actividades del usuario y la flexibilidad para permitirle apartarse del plan cuando sea necesario. Idealmente, el usuario debe poder acceder al plan, consultar y ejecutar las actividades allí prescritas, pero no estar limitado a ejecutar sólo actividades previstas en el plan para realizar su trabajo.

Un punto muy importante a definir es cómo se le mostrará la información al usuario y cómo se le permitirá a éste actuar sobre la representación de esa información. En otras palabras, cuáles serán los componentes que formarán el plan de instanciación presentado al usuario y que

él manipulará. Por ejemplo, los libros de recetas utilizan, como su nombre lo indica, la receta como el concepto básico, alrededor del cual se estructura la explicación. Estas recetas son organizadas, generalmente, siguiendo la metáfora de libro electrónico, es decir, como un conjunto de documentos de hipertexto que le permiten al usuario navegar a través de las explicaciones hasta llegar a las instrucciones de instanciación.

Para evitar los inconvenientes de falta de flexibilidad asociados a las recetas, los componentes del plan de instanciación pueden ser adecuadamente representados utilizando tareas de usuario, un concepto proveniente del dominio de las interfaces a usuario. Las tareas de usuario son utilizadas para modelar la interacción del usuario con el sistema y describen los patrones que puede seguir esta interacción [Pue97, SSC95, JWMP93, Pat97]. El uso de modelos de tareas en aplicaciones complejas permite una interacción más rica entre el usuario y el sistema, porque el sistema es capaz de entender con mayor precisión los objetivos del usuario y brindarle asistencia para que los alcance. Esta asistencia puede materializarse de distintas formas, como por ejemplo:

- crear un modelo del usuario y adaptar la aplicación a sus necesidades,
- asistir en el desarrollo de sistemas de ayudas para las aplicaciones [Pan95],
- asistir en el uso de múltiples aplicaciones conectadas.

Una de los funciones de los modelos de tareas es relacionar la semántica de las acciones del usuario con las acciones básicas de nivel inferior (eventos) impuestas por los *toolkits* para interfaces gráficas. Usualmente, las aplicaciones consideradas tienen un conjunto bien definido de acciones de usuario básicas, el cual es aún demasiado complejo para ser entendido por usuarios novatos y para ser recordado por los diseñadores mientras están concentrados en otros aspectos de la interfaz. En estos casos, el uso de modelos de tareas facilita la comprensión y utilización de las interfaces. La utilización de modelos de tareas de usuario en el dominio de las interfaces a usuario es explicada con más detalle en el Apéndice A.

Aplicando el concepto de tareas de usuario a la instanciación de *frameworks*, un plan de instanciación estará compuesto de **tareas de instanciación**; cada tarea de instanciación corresponde a una de las actividades que debería ejecutar el usuario para crear su aplicación. Estas tareas pueden ser básicas, correspondiendo a las actividades básicas que componen el proceso de instanciación, o complejas, estando en este caso formadas por tareas más simples. Por ejemplo, una receta de instanciación puede ser representada a través de una tarea compleja, compuesta de subtareas.

Es esta posibilidad de representar las actividades del usuario con distinto nivel de granularidad lo que otorga mayor flexibilidad y utilidad a las tareas de instanciación respecto de las recetas:

- En primer lugar, actividades más simples pueden ser reutilizadas y combinadas más fácilmente.
- La información disponible para generar los planes de instanciación a veces sólo cubre parte de la funcionalidad requerida. Utilizando tareas de instanciación es posible crear planes parciales, que muestren cómo implementar aquellas partes contempladas en la documentación. En el caso de las recetas, es muy difícil especificar que una receta debe aplicarse sólo parcialmente.
- Finalmente, la utilización de tareas de instanciación es útil para intentar reconocer el objetivo de las acciones que ejecuta el usuario, es decir, ponerlas en correspondencia con las tareas previstas en el plan de instanciación.

Por otro lado, el concepto de libro electrónico utilizado en los libros de recetas y otros métodos de documentación también puede ser utilizado en combinación con las tareas para organizar la información de diseño e instanciación. Así mismo, la utilización de herramientas

específicas también puede ser combinada con las tareas de instanciación. En este caso, las herramientas no se utilizan para instanciar toda la aplicación, sino para llevar a cabo partes de las actividades del proceso de instanciación. La selección de una tarea dada para ser llevada a cabo puede ocasionar la ejecución de una herramienta asociada.

2.3. Información Necesaria

Independientemente del mecanismo utilizado para generar el plan de instanciación, será necesario proveer información adicional, más allá de la documentación tradicional de diseño, que haga posible este proceso de generación. El tipo concreto de información necesaria es dependiente del mecanismo elegido y, en el caso de los algoritmos de planificación, lo que se necesita es la descripción de las acciones que pueden ejecutarse para alcanzar el objetivo. Como el foco de la herramienta estará en la funcionalidad de la aplicación, estas acciones deben estar descritas en término de la funcionalidad obtenida con cada acción.

Analizando el dominio de creación de aplicaciones a partir de *frameworks*, las acciones involucradas pueden ser clasificadas en dos categorías. La primer categoría incluye aquellas acciones que resuelven situaciones recurrentes en el desarrollo de aplicaciones basadas en *frameworks*. Estas acciones son generales para todos los *frameworks* y pueden ser provistas por la herramienta misma. La segunda categoría comprende las acciones dependientes del *framework* y representan conocimiento acerca de cómo instanciar el *framework* en particular. Estas acciones deben ser provistas por los documentadores del *framework* y pueden estar basadas en acciones más generales.

Cuanto mayor ser el grado de formalidad de la documentación del *framework*, mayor será la asistencia que pueda ser provista al usuario. Si se utilizaran notaciones formales para especificar el *framework*, las acciones necesarias para instanciarlo y los requisitos de la aplicación que se quiere implementar, sería posible derivar la forma de implementar esa funcionalidad utilizando el *framework*.

Por otra parte, un factor que debe ser tenido en cuenta es el esfuerzo que supone documentar el *framework* y, más concretamente, la creación de la documentación adicional necesaria para la generación del plan de instanciación. El desarrollo de un *framework* es una labor que requiere gran cantidad de esfuerzo y el éxito de esta labor, es decir, que el *framework* pueda ser utilizado para crear aplicaciones de forma eficaz, depende en gran medida de la documentación que lo acompaña. Por este motivo, es conveniente invertir tiempo y esfuerzo en facilitar todo lo posible el trabajo de los usuarios del *framework*.

Aún así, el esfuerzo requerido para elaborar la documentación necesaria no debiera ser excesivo. Un proceso de documentación costoso no sólo no es conveniente desde el punto de vista económico, sino que muy probablemente ocasione que la tarea no sea hecha o sea hecha de forma incompleta. Por este motivo, al diseñar la técnica de documentación a utilizar, es importante mantenerla lo más simple posible.

Una especificación formal del *framework* y de la acciones necesarias para instanciarlo puede resultar tan costoso como la implementación misma de la aplicación. Además, este tipo de especificaciones son demasiado rígidas, careciendo de la capacidad de adaptarse a situaciones imprevistas. En consecuencia, es necesario alcanzar un equilibrio entre la formalidad necesaria para posibilitar la generación planes de instanciación y la informalidad necesaria para facilitar el proceso de documentación y flexibilizar el proceso de instanciación.

En este sentido, la utilización de técnicas de planificación ayuda a obtener un mayor grado de asistencia a partir de especificaciones semiformales, tanto de las acciones de instanciación como de la funcionalidad requerida para la aplicación. De todas formas, aunque la técnica de documentación no es excesivamente complicada, es conveniente disponer de notaciones y herramientas que faciliten al diseñador la creación de la documentación.

Una de las formas de facilitar la creación de la documentación es utilizar notaciones gráficas semiformales para describir esta documentación, al menos parcialmente. Estas notaciones gráficas pueden ser diseñadas como extensiones de notaciones gráficas utilizadas tradicionalmente para la documentación de diseños orientados a objetos, como por ejemplo *UML* [RJB99], *OOSE* [JCJO92] u *OMT* [RBP+91]. Esta alternativa facilita la comprensión de la descripción, al mismo tiempo que permite integrar toda la documentación, uniformando la forma en que ésta es presentada al usuario.

También algunas de las técnicas formales utilizadas para documentar *frameworks*, como por ejemplo los contratos de interacción [HHG90], pueden ser utilizados como complemento, porque permiten verificar si el código desarrollado cumple los protocolos de comunicación que deben ser seguidos por los objetos de la aplicación. La utilización de estas técnicas supone la ventaja que no requiere esfuerzos adicionales a la documentación normal del *framework*.

3. Enfoque Propuesto

En función de las necesidades planteadas para un entorno de apoyo a la instanciación de *frameworks*, aquí se propone un método para documentar e instanciar *frameworks* llamado ***SmartBook***. Este enfoque se basa en el concepto de asistencia orientada por la funcionalidad. Es decir, los *SmartBooks*, al igual que los libros de recetas interactivos, guían activamente al usuario durante el proceso de creación de una aplicación, teniendo en cuenta la funcionalidad que el usuario quiere implementar. Sin embargo, *SmartBook* propone que no se utilicen recetas estáticas, sino que las acciones a ejecutar para la instanciación sean generadas dinámicamente, utilizando para ello **técnicas de planificación**.

Para lograr esto, la documentación normal del *framework* es extendida con reglas de instanciación. Utilizando estas reglas y en base a una descripción de la funcionalidad requerida, las técnicas de planificación son capaces de generar **planes de instanciación** orientados a obtener esa funcionalidad. Estos planes de instanciación son representados como una secuencia de **tareas de instanciación** que el usuario debería ejecutar. Toda la documentación del *framework*, utilizada para generar los planes y brindar información al usuario durante la instanciación, es estructurada en base al concepto de libro electrónico.

De esta forma, la documentación y utilización de *frameworks* a través de *SmartBook* pueden ser divididas en tres etapas (Figura 4.2):

- En una primera etapa, el diseñador documenta el *framework* utilizando para ello notación tradicional, como por ejemplo *UML*, y reglas que describan cómo utilizar el *framework*. Estas reglas serán descritas, parcialmente, en base a notaciones gráficas semiformales, siendo necesario un complemento utilizando notaciones textuales semiformales.
- La segunda etapa comprende el proceso de planificación. Los datos manipulados en esta etapa son descripciones semiformales de las acciones de instanciación. Esta información es extraída a partir de las descripciones semiformales, tanto gráficas como textuales, provistas por el diseñador, más acciones generales independientes del *framework*. El producto de esta etapa es el plan de instanciación.
- Finalmente, la tercera etapa es la instanciación propiamente dicha. El usuario interacciona con la información de instanciación (documentación y plan) en base al concepto de tarea de instanciación.

Estas etapas serán analizadas con más detalle a continuación. Para una mayor claridad en esta explicación, el orden en que estas etapas son analizadas no corresponde al orden en el que las etapas se ubican a lo largo del proceso de instanciación.

Es importante destacar que esta división de datos y fases del proceso, si bien existe desde un punto de vista conceptual, no siempre es claramente distinguible en la práctica. Así, por

ejemplo, en el estado actual de desarrollo de la propuesta, el diseñador debe documentar parte del *framework* utilizando directamente las acciones de instanciación.

3.1. Planificando la Instanciación de un *Framework*

El problema general de la planificación consiste en construir un plan completo a partir de una serie de entradas. Para esto necesita definiciones de acciones que le permitan encontrar un camino desde el estado inicial hasta el estado final buscado. Básicamente, un algoritmo de planificación recibe tres entradas:

- Descripción del mundo en algún lenguaje formal.
- Descripción de los objetivos en algún lenguaje formal.
- Descripción de las acciones que puede ejecutarse en algún lenguaje formal. Esto se denomina **teoría del dominio**.

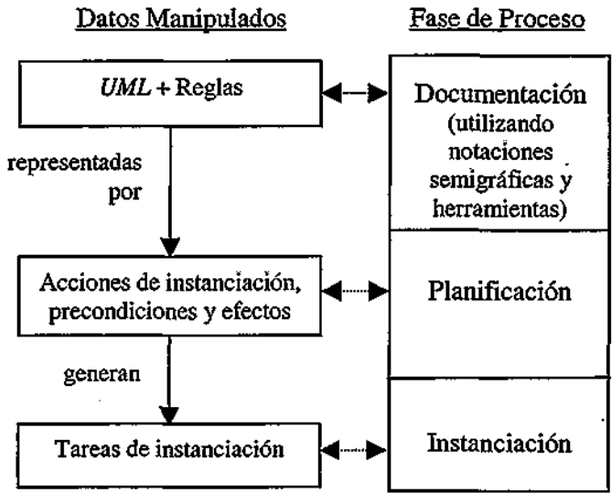


Figura 4.2 Relación entre los distintos tipos de datos manipulados en las diversas fases del proceso de documentación e instanciación.

El algoritmo produce como salida un plan completo a partir de estas entradas. Es decir, una secuencia de acciones que, cuando son ejecutadas en cualquier mundo que satisfaga las condiciones iniciales, permitirán alcanzar el o los objetivos.

En el caso concreto de la instanciación de *frameworks*, el estado inicial está representado por el código del *framework* y el estado final es el código de la aplicación que implementa la funcionalidad requerida. De esa forma, lo que se necesitan son acciones que permitan partir de la funcionalidad requerida y obtener las tareas de instanciación correspondientes. Es decir, información que describa cómo utilizar el *framework* para implementar una determinada funcionalidad, en términos de tareas de instanciación.

Además, las condiciones generales de la técnica de planificación deben ser adecuadas al contexto del dominio de instanciación de *frameworks*. Según el análisis previo, existen tres aspectos en particular que deben ser considerados:

- la formalidad de los lenguajes de las entradas. Estos lenguajes deben ser simplificados tanto como sea posible, de manera que se facilite el proceso de documentación.

- la presunción de generación de un plan completo: en general puede ocurrir que, dado un *framework* y una descripción de la funcionalidad requerida, no se posea la suficiente información para generar un plan de implementación de esa funcionalidad usando el *framework*. Esto puede deberse a que el *framework* no provee esa funcionalidad, o simplemente que no fue prevista en la documentación. En cualquier caso, una herramienta de apoyo debe ser capaz de generar un plan parcial para implementar aquella parte sobre la que sí se posee información, dando la posibilidad al usuario de implementar el resto por su cuenta.

  En este sentido, el uso de técnicas de planificación denominadas **planificación con minimización de compromisos** (*least commitment planning*) [VB94, VS95, Wel94,99] aparece como la alternativa más adecuada para construir de forma flexible planes parciales, compuestos de secuencias de tareas de instanciación. En el próximo capítulo, la técnica de planificación con minimización de compromisos es descrita en detalle y a continuación el algoritmo de planificación que ha sido desarrollado para utilizar en *SmartBooks*.

- información necesaria para la generación: es necesario disponer de información que permita asistir al usuario aún cuando no se conozca el plan completo que debería ejecutar para implementar su aplicación. En otras palabras, debe ser posible reaccionar y adaptarse en la medida de lo posible a las acciones del usuario aún cuando este ejecutando acciones no previstas. La próxima sección describe la información que es necesario agregar para obtener este soporte.

3.2. Documentando el *Framework*

En base al análisis de necesidades específicas de la instanciación de *frameworks*, es posible ver que se necesita tanto información para asistir en la implementación de la funcionalidad prevista, como información que ofrezca guías para la instanciación independientemente de la funcionalidad que se está implementando. Para disponer de esta información, *SmartBooks* propone complementar la documentación tradicional de diseño utilizando dos tipos de reglas: **reglas funcionales** y **reglas de consistencia**.

3.2.1. Reglas Funcionales

Las reglas funcionales deben permitir al sistema asistir al usuario en la implementación de requisitos funcionales determinados. Para esto es necesario que asocien cada requisito con el conjunto de tareas de instanciación que deben ser ejecutadas para obtener esa funcionalidad, así como también las condiciones necesarias para aplicar la regla y las condiciones que deberían cumplir las tareas correspondientes. En función de esto, un algoritmo de planificación puede generar la lista de tareas que deben ser ejecutadas para implementar la funcionalidad requerida.

Por ejemplo, considérese la implementación de diagramas animados utilizando *HotDraw*. Para implementar esta funcionalidad, es necesario especializar la clase *Drawing*, cuyas instancias son utilizadas para representar el diagrama. Además, el método *step* debe ser redefinido. Este método es invocado automáticamente y se presume que en cada invocación actualizará la posición en pantalla de los objetos del diagrama, de acuerdo con alguna lógica. Como esta lógica es dependiente de la aplicación, nada de ello puede ser especificado en la documentación general del *framework*, quedando su implementación en manos del usuario. Sin embargo, sí es posible establecer que después de actualizar las posiciones, el diagrama debe enviar el mensaje *displayOn*: a si mismo, para que la presentación en pantalla sea actualizada.

De este modo, una regla funcional que describa esta situación tendrá la siguiente forma:

Funcionalidad obtenida:
Diagramas Animados

Requisitos:

1. Crear una subclase de *Drawing*
2. Para la nueva clase, definir el método *step*.
3. El método *step* debe invocar "*self displayOn:*"

En esta regla, los requisitos 1 y 2 describen tareas de instanciación que deberían ser ejecutadas por el usuario (*DefinirClase* y *DefinirMétodo*). El tercer requisito, por su parte, no implica la ejecución de una tarea distinta, sino que establece restricciones sobre la segunda tarea.

3.2.2. Reglas de Consistencia

Las reglas de consistencia, por su parte, describen estados consistentes del software y que el código escrito por el usuario del *framework* debe respetar. Es decir, definen convenciones y protocolos establecidos por el *framework*, los cuales deben ser respetados por las aplicaciones derivadas, independientemente de la funcionalidad implementada. Estas reglas son utilizadas tanto por el planificador para completar detalles del plan de instanciación, como por el **Administrador de Consistencia**. Este Administrador de Consistencia puede controlar y orientar la actividad del usuario durante el proceso de instanciación, informándole cuando se produzca alguna situación inconsistente con el diseño del *framework*, y guiándolo para ejecutar las tareas necesarias para restaurar la consistencia. Esta asistencia es provista tanto cuando el usuario está ejecutando tareas del plan de instanciación como cuando actúa por su cuenta.

Mientras las reglas funcionales permiten guiar al usuario en la implementación de funcionalidad descrita en la documentación, las reglas de consistencia permiten orientarlo también cuando la funcionalidad requerida no ha sido prevista en esta documentación.

Las reglas de consistencia, al igual que las funcionales, son descritas en términos de tareas de instanciación, y consisten de tres partes distintas:

- Las tareas que pueden dar origen al estado inconsistente;
- Las condiciones que debe cumplir el software para que la regla sea aplicable, es decir, para considerar que el software no cumple con el diseño general del *framework*;
- Las tareas de instanciación que el usuario debería ejecutar para salvar el estado de inconsistencia.

Un ejemplo de regla de consistencia es una regla asociada a un método abstracto. Por definición, un método abstracto deber ser redefinido en cada clase concreta, es decir, en cada clase de la cual se creen instancias. Entonces el diseñador podría establecer que cada vez que el usuario defina una subclase de una clase dada, y que esa nueva clase sea utilizada para crear instancias, determinado método debe ser redefinido. En *HotDraw* esto sucede con el método *displayOn:* de la clase *Figure*. Una regla de consistencia para describir esta situación tendrá la siguiente forma:

Tarea de origen:

Definir una nueva clase, subclase de Figure.

Condiciones de inconsistencia:

La nueva clase hereda de Figure, no tiene implementado un método llamado displayOn:, no tiene una superclase que tenga implementado displayOn: y se crean instancias de la nueva clase

Tareas reparadoras:

Implementar el método displayOn: para la nueva clase.

Utilizando este tipo de reglas, un Administrador de Consistencia puede analizar las condiciones de las reglas cada vez que una tarea es ejecutada por el usuario y crear las tareas necesarias para mantener la consistencia. Este mecanismo ofrece guías al usuario para desarrollar su aplicación, aún cuando se desconozca la funcionalidad que está implementando.

Una de las principales ventajas de las reglas de consistencia es que facilitan la descripción de situaciones recurrentes en la instanciación de *frameworks*. Estos casos generales, independientes del *framework*, pueden ser provistos al encargado de documentar el *framework* en forma de biblioteca de reglas. Cada una de estas reglas será una descripción genérica, en la cual no se hace referencia a componentes concretos sino a roles que pueden desempeñar los componentes.

Generalizando el ejemplo del método *displayOn:*, la regla de consistencia para métodos abstractos será la siguiente (donde *ClaseAbstracta* es una clase que tiene definido un método abstracto llamado *MétodoAbstracto*):

Tarea de origen:

- Definir una nueva clase *NuevaClase*, subclase de *ClaseAbstracta*.

Condiciones de inconsistencia:

La clase *NuevaClase* hereda de *ClaseAbstracta*, no tiene implementado un método llamado *MétodoAbstracto*, no tiene una superclase que tenga implementado *MétodoAbstracto* y se crean instancias de *NuevaClase*

Tareas reparadoras:

Implementar el método *MétodoAbstracto* para la clase *NuevaClase*.

En esta definición, *NuevaClase*, *ClaseAbstracta* y *MétodoAbstracto* son roles a ser cumplidos por clases y métodos concretos del *framework* que se está documentando. Cuando el diseñador quiera utilizar la regla para documentar su *framework*, sólo tendrá que especificar el componente asociado con cada participante de la regla.

Más aún, estas situaciones recurrentes pueden ser descritas a través de las notaciones de diseño creadas especialmente con este objetivo, como por ejemplo los patrones de diseño [GHJV94] o los meta-patrones [Pre94]. En este caso, el diseñador no tendrá que pensar en describir las distintas reglas que deben ser respetadas, sino que simplemente deberá describir los distintos patrones que fueron aplicados en el diseño de los componentes del *framework*. Cada aplicación de un patrón dará lugar a la asociación de las reglas de consistencia correspondientes a ese patrón. De esta forma, la documentación del *framework* utilizando técnicas tradicionales permite la incorporación de reglas semiformales para ofrecer al usuario asistencia más avanzada. En el capítulo VIII se profundiza más en la aplicación de reglas de consistencia en asociación con los patrones de diseño.

El enfoque propuesto no limita la utilización de reglas a las explícitamente definidas por el autor de la documentación y a las derivadas de los patrones de diseño. Información adicional para las reglas de consistencia es extraída de documentación tradicional, como los diagramas de secuencia definidos por *UML*. De la misma forma, este tipo de información puede ser extraída de notaciones formales, como por ejemplo los contratos de interacción.

3.3. Guiando el Proceso de Instanciación

A partir de la información sobre las tareas de instanciación que deberían ser ejecutadas para implementar una aplicación determinada, es posible construir un administrador de tareas que se encargue de asistir al usuario durante el proceso de instanciación. Este sistema administrador debe permitir que las tareas sean ejecutadas en cualquier orden permitido, sean interrumpidas para trabajar en otras tareas, e incluso canceladas en cualquier momento, y debe ser capaz de adaptarse a las nuevas situaciones que tengan lugar en cada uno de esos pasos. Una de la funciones más importantes de este sistema es actuar como un agenda de asuntos pendientes, recordándole al usuario las tareas que aún debe completar.

3.4. Entorno de Documentación e Instanciación

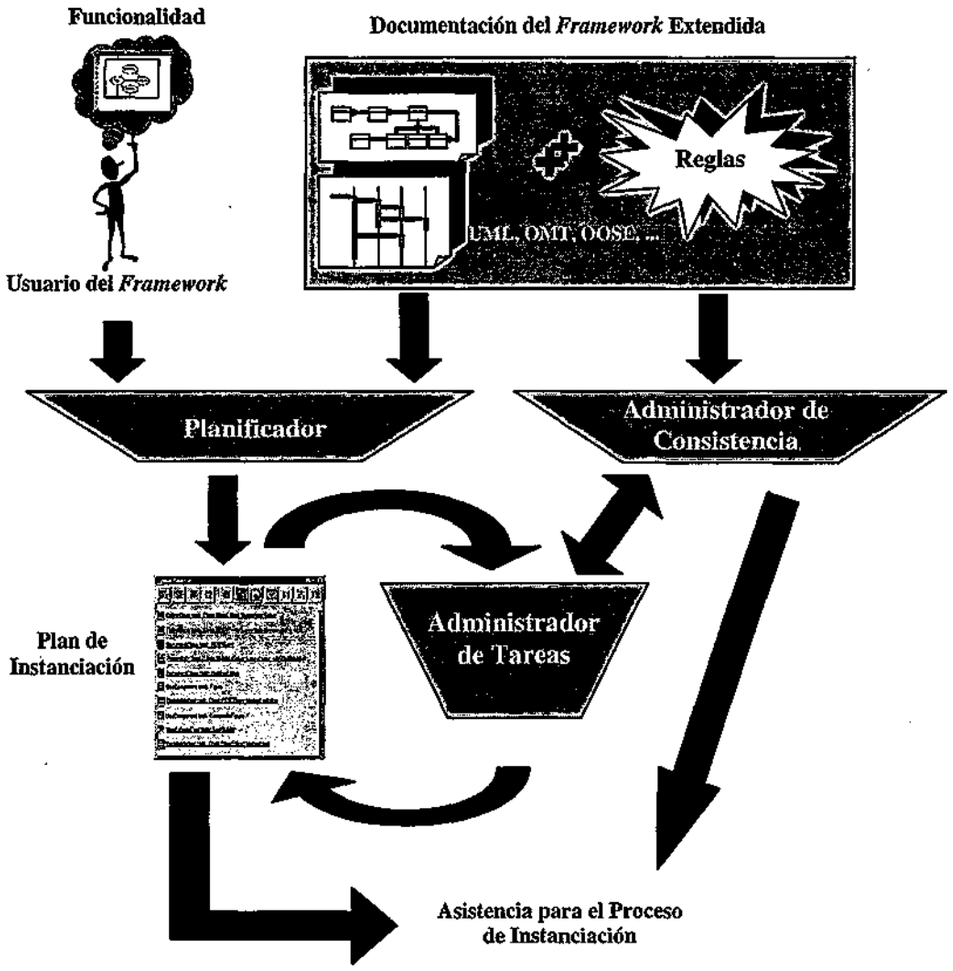


Figura 4.3 Mecanismos propuestos por *SmartBooks* para guiar el proceso de instanciación.

El diseño de un entorno de apoyo a la documentación e instanciación de *frameworks* basado en *SmartBooks* requiere el análisis de varios aspectos relacionados. Los próximos capítulos están dedicados al estudio de estos aspectos:

- Algoritmo de Planificación (capítulo V): se describen las características del algoritmo de planificación ***PIT***, diseñado para satisfacer los requerimientos del proceso de instanciación de *frameworks*.

- Notación Gráfica (capítulo VI): se presenta la notación diseñada para facilitar el proceso de definición de las reglas de instanciación.
- Administrador de Consistencia (capítulo VI): se describe el diseño de un sistema capaz de controlar las actividades del usuario, guiándolo en la creación de su aplicación en la medida de la información disponible
- Administrador de Tareas (capítulo VI): se realiza un análisis de las características de un sistema de control de tareas de instanciación.

La Figura 4.3 resume los mecanismos propuestos por *SmartBooks* para apoyar la instanciación de *frameworks*.

4. Ejemplo de Instanciación Utilizando *SmartBooks*

En esta sección se presenta un ejemplo de la interacción de un usuario con una herramienta basada en el método *SmartBooks*. La herramienta utilizada para este ejemplo es *HiFi*, cuyo diseño e implementación es explicado en el capítulo VIII. Volviendo al ejemplo introducido en §IV.1, se mostrará el uso de la herramienta para implementar un editor de diagramas *PERT* utilizando *HotDraw*.

En primer lugar, el usuario debe ser capaz de especificar la funcionalidad requerida para la nueva aplicación. En el caso del editor *PERT*, la descripción informal de este editor es la siguiente:

> *"Debe ser posible crear de forma interactiva objetos gráficos que representen tareas, y relacionar estas tareas a través de vínculos de precedencia. Las tareas deben tener una representación visual para sus atributos y dos de los atributos serán editados a través de la interfaz gráfica de usuario: para editar Duration se utilizará un tool, mientras que con EndDate se usará un menú. Además, cada atributo puede estar relacionado con otros, tanto de la misma tarea como de tareas relacionadas."*

Para que el usuario pueda proveer esta especificación, se le ofrece una lista con la funcionalidad disponible en el *framework*, la cual es construida a partir de la documentación provista por el diseñador. En el caso de *HotDraw*, la funcionalidad mostrada al usuario es la que se muestra en la Figura 4.4.

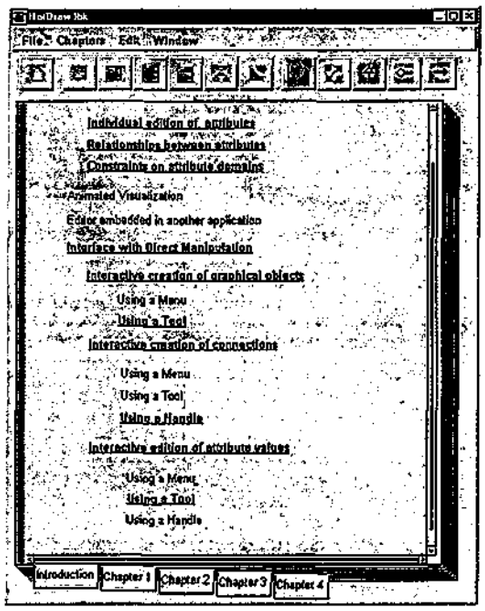


Figura 4.4 Especificación de la funcionalidad requerida para la nueva aplicación

A través de esta interfaz el usuario puede seleccionar cuáles de las funciones provistas por el *framework* deben estar disponibles en la nueva aplicación. La sangría representa las relaciones de opciones / subopciones que se le presentan al usuario. Por ejemplo, una de las opciones que se muestra al comienzo es la de dotar al editor de una interfaz de usuario con manipulación directa. Si el usuario escoge esta opción, se le presenta tres alternativas posibles (*Creación interactiva de objetos gráficos, Creación de relaciones, Edición de los valores de atributos*), cada una de las cuales tiene a su vez alternativas. Al elegir alguna de las opciones (opciones subrayadas en negrita) se le solicita al usuario información adicional. Así, por ejemplo, si escoge editar interactivamente un atributo utilizando un menú, deberá especificar el atributo concreto al que se refiere.

En el ejemplo del editor *PERT*, la lista de requisitos funcionales seleccionados por el usuario será la siguiente (la información introducida por el usuario aparece al final de cada requisito entre paréntesis):

1. Interactive creation of graphical objects using a Tool (PertTask)
2. Individual edition of attributes using a Tool (PertTask, Duration)
3. Individual edition of attributes using a Menu (PertTask, EndDate)
4. Relationships between attributes(PertTask)
5. Relationships between attributes (PertTask, PertTask)
6. Interactive creation of relationships using a Handler (PertTask, PertTask)

Por ejemplo, el requisito 1) indica que el editor debe proveer una forma de crear interactivamente *PertTasks*, utilizando para ello un *Tool* (un tipo de componente provista por *HotDraw* para implementar manipulación directa). El requisito 3), por su parte, representa que el usuario quiere que el atributo *EndDate* de las *PertTasks* pueda ser editado utilizando un menú. Los requisitos 3) y 4) hacen referencia a la creación de relaciones (restricciones) entre atributos de las *PertTask*; el primero representa relaciones entre atributos de una misma figura, mientras que el segundo se refiere al establecimiento de relaciones entre atributos de distintas figuras.

La Figura 4.5 representa la interfaz de usuario del Administrador de Tareas provisto por *HiFi*. Esta interfaz muestra una lista de las tareas de instanciación ejecutadas y pendientes de ejecución, brindándole al usuario la posibilidad de seleccionar una tarea y, entre otras cosas, ejecutarla o solicitar información relacionada con la tarea. Cada ítem de esta lista se corresponde con una tarea de instanciación que debería ser ejecutada por el usuario: especializar clases, escribir o modificar métodos, documentar un nuevo componente, etcétera. Esta lista es generada, inicialmente, por el planificador, a partir de la especificación de funcionalidad requerida para la aplicación. Así, el sistema puede guiar al usuario a través de los pasos necesarios para implementar su aplicación. A partir de esta lista, el usuario puede inspeccionar las tareas que debe ejecutar y elegir la próxima a llevar a cabo; también puede estudiar la documentación asociada a cada tarea e incluso puede optar por modificar una tarea que ya ha sido ejecutada.

La primera tarea de la lista, por ejemplo, representa que el usuario debe crear una clase, llamada *PertEditor*, subclase de *Editor*. Algunas tareas no son necesarias para implementar la funcionalidad requerida, pero le indican al usuario actividades que probablemente deba llevar a cabo, como por ejemplo la tarea de modificación del método *addSource:*. Para distinguir las tareas obligatorias de las optativas, las primeras se presentan subrayadas.

La lista de tareas presentada al usuario no necesariamente representa todas las actividades de programación que se necesitan para implementar la funcionalidad especificada, sino solamente aquellas que el planificador ha logrado determinar a partir de la documentación disponible sobre el *framework*. Así, por ejemplo, no se provee ninguna tarea para implementar la edición del atributo *EndDate* utilizando un menú. Esto no quiere decir que no se pueda hacer,

sino que el planificador no fue capaz de encontrar una forma de hacerlo. Por este motivo, el sistema también retorna la lista de objetivos funcionales no contemplados en el plan actual.

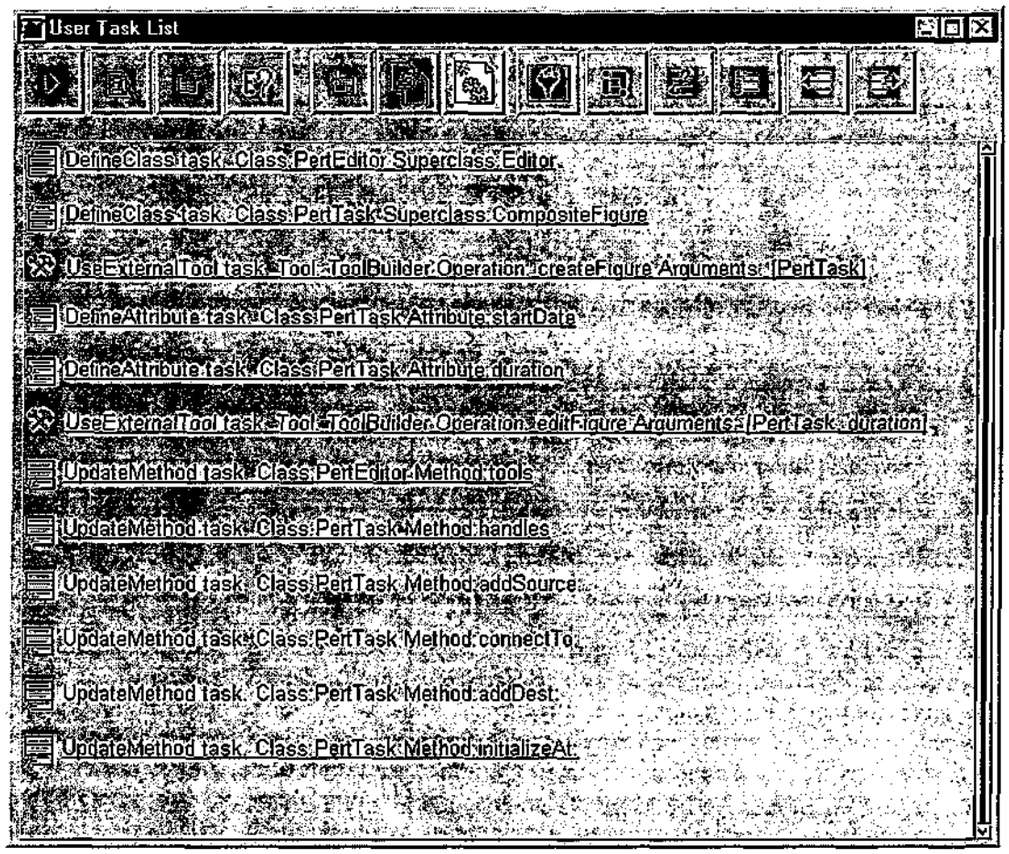


Figura 4.5 Interfaz del Administrador de Tareas

Además de la funcionalidad básica aquí mostrada, *HiFi* dispone, entre otras cosas, de medios para asistir la ejecución de tareas, por ejemplo proveyendo formularios específicos para la definición de clases o la implementación de métodos. También permite, dada una determinada tarea, averiguar los requisitos funcionales que dieron lugar a su inclusión en el plan de instanciación o solicitar la elaboración de un plan alternativo que no comprenda la ejecución de dicha tarea. En lo que respecta a la documentación del *framework*, *HiFi* representa la misma como hiperdocumentos, permitiendo navegar a través de la misma y estudiar las relaciones existentes entre la documentación provista por el diseñador y el software que está desarrollando el usuario.

Algunas de estas funciones provistas por *HiFi* son explicadas más detalladamente en el capítulo VIII.

V Generación de Planes de Instanciación

Para utilizar el modelo *SmartBooks* propuesto en el capítulo anterior, es necesario disponer de herramientas que apoyen tanto el proceso de documentación como el proceso de instanciación de *frameworks*. Uno de los aspectos más importantes que debe ser tenido en cuenta en el diseño e implementación de estas herramientas es el mecanismo que se utilizará para generar los planes de instanciación, denominado planificador (*planner*).

Para la generación de los planes de instanciación fue necesario diseñar un algoritmo específico, denominado *PIT (planning instantiation tasks* – planificación de tareas de instanciación). Este es un algoritmo generativo, que utiliza refinamiento para construir el plan. Además, es regresivo y se basa en la técnica de minimización de compromisos, realizando búsquedas a través del espacio de planes y siendo capaz de producir planes parciales.

Este capítulo explica el algoritmo *PIT* y cómo se representa la información del *SmartBooks* para ser utilizada por el planificador. En primer lugar se hará una introducción a la técnica de planificación (*planning*), comenzando por algoritmos simples a los que se agregará paulatinamente complejidad y potencia, finalizando con el algoritmo *UCPOP* [Wel94], utilizado como base para el desarrollo de *PIT*. Luego se analizarán los requisitos particulares del problema de instanciación de *frameworks* y finalmente se describirán las soluciones propuestas a través de *PIT*.

1. Técnicas de Planificación

Una de las áreas de investigación en Inteligencia Artificial es la resolución de problemas en los que un agente, en su interacción con el mundo, necesita cumplir objetivos mediante la ejecución de **acciones**. Existen dos formas de determinar las acciones necesarias para resolver objetivos específicos [Wel94]:

- Si la naturaleza del problema es simple, es posible calcular en cada paso la mejor acción a partir del estado del mundo. Es decir, no es necesario anticipar acciones, porque éstas no se afectan unas a otras. En estos casos es conveniente aplicar técnicas relativamente simples, en particular una técnica conocida como **acción situada** (*situated action*).
- Si el problema es más complejo y se necesitan varias acciones encadenadas en cierto orden, cumpliendo ciertas restricciones o las acciones interaccionan de forma compleja, es necesario un **plan**; este plan se obtiene aplicando **técnicas de planificación**. Los algoritmos de planificación [VS95, Wel99] buscan generar las acciones que permitan alcanzar los objetivos.

Dadas las características de los problemas de generación de planes de instanciación de *frameworks*, aquí nos concentraremos en el estudio de las técnicas de planificación.

1.1. El Problema de la Planificación

Como ya se vio en el capítulo anterior (§IV.3.1), un algoritmo de planificación debe producir un plan completo a partir de una descripción del mundo, una descripción de los objetivos del agente y una descripción de las acciones que pueden ser ejecutadas (**teoría del dominio**). El plan está constituido por una secuencia de acciones que, cuando son ejecutadas en cualquier mundo que satisfaga las condiciones iniciales, permitirán alcanzar el objetivo o los objetivos.

Existen dos formas de construir un plan, denominadas **planificación generativa** y **planificación basada en casos**. El primer enfoque consiste en construir un plan completo desde el inicio, es decir, comenzando desde cero e incorporando acciones y restricciones. El segundo consiste en reutilizar un plan o parte de un plan sintetizado con anterioridad, realizando previamente las modificaciones necesarias.

Independientemente de si se utiliza planificación generativa o basada en casos, la construcción del plan puede ser hecha de tres formas alternativas:

- Refinamiento: consiste en incorporar gradualmente acciones y restricciones a un plan parcial
- Reducción: consiste en eliminar elementos incorporados previamente a un plan
- Transformación: combina actividades de refinamiento y reducción.

En este trabajo, el enfoque se basará en algoritmos generativos, que utilizan refinamiento para incorporar acciones y restricciones

1.1.1. Simplificaciones

El tipo de problemas que puede ser resuelto con un algoritmo de planificación depende del lenguaje utilizado para representar el problema, es decir, las entradas del algoritmo. Cuanto más potente sea el lenguaje utilizado, mayor será el número de situaciones que puedan ser representadas utilizándolo. Pero junto con la potencia de un lenguaje también crece la complejidad del algoritmo necesario para tratarlo.

El mundo real es dinámico, es decir, cambia independientemente de las acciones que un agente pueda tomar. Es también impredecible, cambia de una manera demasiado compleja como para que un agente pueda predecir sus cambios. Además, el mundo y sus cambios afectan a los objetivos y a las acciones que un agente puede ejecutar. La representación de estas situaciones necesita de lenguajes complejos, lo cual a su vez incrementa la complejidad de los algoritmos de planificación.

Por lo tanto, todos los algoritmos generales presentados en esta sección, incluyendo *UCPOP*, están basados en simplificaciones del mundo real, con el fin de facilitar su construcción y explicación. Las simplificaciones adoptadas son las siguientes:

- Tiempo atómico: la ejecución de una acción es indivisible y no puede ser interrumpida, es decir, no se pueden ejecutar acciones simultáneas y, por lo tanto, no se necesita considerar el estado del mundo mientras procede la ejecución de la acción. Entonces se pueden modelar las acciones como transiciones de un estado a otro.
- Efecto determinista: el resultado o efecto de ejecutar una acción es una función determinista del estado del mundo y de la acción.
- Omnisciencia: el agente tiene un conocimiento total o completo del estado inicial del mundo y de las acciones que él puede tomar.
- Única causa de cambio: sólo el agente puede cambiar el estado del mundo aplicando alguna acción, es decir, el mundo y los demás agentes son estáticos.

Como se verá más adelante, alguna de estas simplificaciones son excesivamente restrictivas en el caso de la instanciación de *frameworks*, por lo que luego deberán ser eliminadas y sus consecuencias tenidas en cuenta en la generación de planes de instanciación.

1.1.2. Lenguaje de representación

Las descripciones proporcionadas al algoritmo de planificación requieren la utilización de algún lenguaje que permita comparar objetivos, precondiciones y efectos de acciones. En los

primeros ejemplos se utilizará un lenguaje muy simple: la representación proposicional *STRIPS* [FN71]. Luego este lenguaje se extenderá para representar situaciones más complejas.

La representación STRIPS describe el estado inicial del mundo con un conjunto completo de literales básicos (*ground*). Para que la descripción inicial sea completa, toda formula atómica no representada explícitamente se supone falsa (Presunción de Mundo Cerrado).

STRIPS está restringida a **objetivos de logro**. La mayor parte de los algoritmos de planificación no aceptan descripciones arbitrarias del comportamiento del agente, sino que sólo aceptan objetivos que especifiquen características que deben ser verdaderas en el mundo después de ejecutarse el plan. *STRIPS*, más aún, restringe el tipo de **estados objetivos** que pueden ser especificados a aquellos que se expresen como una conjunción de literales positivos (Figura 5.1). [1]

(on A Table) (on C A) (on B Table)
(clear B) (clear C)

estado

(on B C) (on A B)

objetivo

Figura 5.1 Descripción de una situación simple utilizando STRIPS

La tercera parte de un problema de planificación está formada por una teoría del dominio. Es una descripción formal de las acciones disponibles para ser ejecutadas por el agente. En *STRIPS*, las acciones se representan a través de precondiciones y efectos. La precondición de cada acción se representa, al igual que el objetivo del problema, con una conjunción de literales positivos. El efecto, por su parte, puede contener tanto literales positivos como negativos.

Por ejemplo, la acción *move-C-from-A-to-Table* es definida así:

Precondición: *(and (on C A) (clear C))*

Efecto: *(and (on C Table) (not (on C A)) (clear A))*

La acciones pueden ser ejecutadas sólo cuando las precondiciones son verdaderas. Cuando una acción es ejecutada, cambia las condiciones del mundo de la siguiente forma: todos los literales positivos de la conjunción que describe el efecto son agregados a la descripción del estado, mientras que todos los negativos son quitados.

1.2. Funcionamiento de los Algoritmos de Planificación

La forma más simple de construir algoritmos de planificación es transformar el problema de planificación en un problema de búsqueda a través de un espacio. Existen dos alternativas para realizar esta conversión; la primera es transformar el problema de planificación en una búsqueda a través de un espacio de estados del mundo, mientras que la segunda consiste en una búsqueda a través de un espacio de planes.

En el primer caso los nodos del grafo son estados del mundo y los arcos son las acciones. Un plan es un camino que conduce del estado inicial al estado final. En el segundo caso, los

[1] En estos primeros ejemplos se utilizará un problema simple, encontrado con mucha frecuencia en la literatura de algoritmos de planificación: el problema de ordenar cubos. Esto se hace para evitar ejemplos irreales del dominio de la instanciación de *frameworks*, ya que las primeras representaciones y algoritmos explicados no son suficientes para manejar un dominio tan complejo como la representación de *frameworks*. Así, una vez que se empiecen a explicar los algoritmos específicos, se volverá a tratar con ejemplos más adecuados para el dominio de interés en este trabajo.

nodos representan planes parcialmente especificados y los arcos denotan operaciones de refinamiento de un plan. En este caso, la solución es el estado final representando a un plan completo.

Una de las ventajas de tratar el problema de planificación en uno de búsqueda es que cualquier algoritmo de búsqueda en grafos puede ser utilizado, desde los algoritmos exhaustivos como búsqueda en profundidad (DFS) o búsqueda en ancho (BFS) hasta algoritmos más avanzados, con heurísticas elaboradas.

Para simplificar el análisis de los algoritmos *POP* y *UCPOP*, sólo se estudiará la estructura del espacio de búsqueda, sin especificar ninguna estrategia concreta para recorrer el grafo. Para esto, se utilizará el recurso, propuesto por Weld [Wel94], de definir algoritmos de planificación no deterministas. Esto se hace en base a una función no determinista *seleccionar*, la cual se presume que toma el conjunto de alternativas para la próxima acción y elige la que considera más apropiada. La utilización de esta función permite simplificar la descripción de los algoritmos, así como modificar la estrategia de búsqueda sin modificar el algoritmo.

Otra importante consecuencia es que, como se presume que la función *seleccionar* siempre elige la mejor opción, al llegar a un punto muerto se sabe que no hay solución posible, pudiendo el programa terminar inmediatamente. Es decir, no es necesario que los algoritmos retrocedan y reconsideren opciones, utilizando por ejemplo *backtracking*.

1.3. Búsqueda a Través de un Espacio de Estados

El problema de planificación, en este caso, consiste en encontrar un camino desde el estado inicial al final (Figura 5.2), eligiendo en cada paso una acción apropiada por medio de la función *seleccionar*. Para generar un camino o plan existen dos tipos de algoritmos:

- Algoritmos progresivos: son aquellos que partiendo desde el estado inicial llegan al final, incorporando gradualmente acciones que le permiten cambiar de estado.
- Algoritmos regresivos: al contrario de los anteriores, parten del estado final y retroceden hasta el estado inicial.

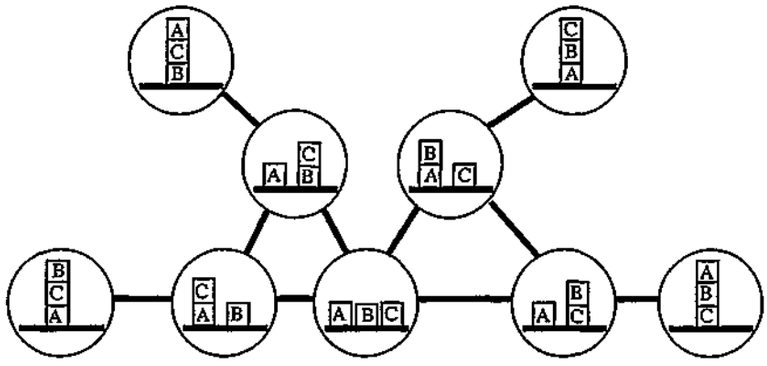


Figura 5.2 Espacio de búsqueda de estados

1.3.1. Algoritmos Progresivos

La Figura 5.3 muestra la estructura básica de un algoritmo progresivo. El parámetro *estadoDelMundoActual* es la descripción del estado del mundo (al principio es el estado inicial), y *listaDeObjetivosFinal* contiene los objetivos finales. En cada paso de la ejecución el algoritmo selecciona una acción para cambiar de estado utilizando la función *seleccionar*. Finalmente, cuando cada objetivo de *listaDeObjetivosFinal* está presente en el *estadoDelMundoActual* entonces el mundo final ha sido alcanzado y el camino o plan es retornado.

Progresivo (*estadoDelMundoActual* , *listaDeObjetivosFinal* , *listaDeAcciones* , *unCamino*)

1. Si *estadoDelMundoActual* satisface cada *objetivo* de la *listaDeObjetivosFinal,* entonces retornar *unCamino*
2. De lo contrario sea *acción* = ***seleccionar*** desde *listaDeAcciones*
   a) Si tal selección no es posible retornar *fracaso*
   b) De lo contrario retornar

   Progresivo (*siguienteEstado* , *listaDeObjetivos* , *listaDeAcciones* , *unCaminoAcual*) donde *siguienteEstado* = resultado de ejecutar acción en *estadoDelMundoActual*, y *unCaminoActual* es concatenar(*unCamino* , *acción*)

Figura 5.3 Estructura de un algoritmo progresivo

1.3.2. Algoritmos Regresivos

Antes de explicar el funcionamiento de los algoritmos regresivos, se establecerá qué significa *regresión.* Sea L una sentencia lógica (conjunción de objetivos) y A una acción con precondiciones P y efecto E. La regresión G de L a través de A es otra sentencia lógica que comprende la condición más débil que debe cumplirse antes de ejecutar A para asegurar que L sea verdadera después de la ejecución. Es decir, por definición, la ejecución de A cuando se cumple G hará verdadero L. En otras palabras, G es el resultado de unir P con los objetivos de L que no son efectos de la acción A. Para que la regresión tenga sentido, es necesario que L sea compatible con los efectos de la acción, es decir, que ninguno de los objetivos de L esté negado en los efectos de la acción.

Si se hace L = *listaDeObjetivosActual*, entonces la aplicación del concepto de regresión para elegir la acción A y las precondiciones necesarias garantiza que *listaDeObjetivosActual* sea verdadera después de ejecutar A. Más formalmente, la regresión G de L a través de A es:

$G = \mathrm{P} \cup [listaObjetivosActual - listaEfectos(A)]$.

Regresivo (*estadoDelMundoInicial, listaDeObjetivosActual, listaDeAcciones, unCamino*)

1) Si *estadoDelMundoInicial* satisface cada *objetivo* de la *listaDeObjetivosActual*, entonces retornar *unCamino*
2) De lo contrario sea A = ***seleccionar*** (desde *listaDeAcciones*) una acción que verifique las siguientes propiedades:

   a) su efecto ha de satisfacer algún objetivo de la *listaDeObjetivosActual* que no sea una precondición de la acción

   b) su regresión con dicha lista ha de estar definida
3) Retornar Regresivo (*estadoDelMundoInicial, G, listaDeAcciones, unCaminoActual*) donde *unCaminoActual* es concatenar (*unCamino, A*), a menos que A no esté definida, en cuyo caso retornar *fracaso*

Figura 5.4 Estructura de un algoritmo regresivo

La estructura de un algoritmo regresivo es mostrada en la Figura 5.4. Debe notarse que no existe nada que prevenga que el algoritmo entre en un bucle infinito si, por ejemplo, a cada paso la *listaDeObjetivosActual* incrementa su tamaño.

1.3.3. Comparación

Si bien para analizar la eficiencia de los algoritmos descritos es necesario especificar la función *seleccionar*[2], es posible establecer una comparación entre ambas técnicas. Intuitivamente es posible ver que los algoritmos regresivos deberán considerar menos alternativas, porque cada vez que la función *seleccionar* deba elegir la próxima acción, sólo tendrá que considerar aquellas que resuelvan alguno de los objetivos de *listaDeObjetivosActual*. En cambio, en los algoritmos progresivos no hay ninguna indicación de qué acción hará acercarse más al objetivo final y se deberán considerar todas las acciones cuya precondición sea satisfecha por el estado actual. Aunque este argumento no es concluyente, muestra una propiedad de los algoritmos regresivos que hace conveniente su utilización.

1.4. Acciones e Instancias de Acciones

Antes de continuar analizando los algoritmos de planificación, es necesario definir algunos conceptos.

Una determinada acción puede ser ejecutada repetidamente como parte de un mismo plan. Por ejemplo, para alcanzar determinados objetivos puede ser necesario ejecutar más de una vez la acción de mover el bloque *A* sobre el bloque *B*. Entonces es necesario distinguir entre una acción y las instancias de esa acción. Cada ejecución de una acción como parte de un plan es una **instancia** de esa acción.

En base a esta distinción, se define el **conjunto de las instancias de acciones** como el producto cartesiano del conjunto de acciones por el conjunto de números naturales. De esta forma, si el conjunto de acciones incluye una acción *Move-A-to-B*, entonces el conjunto de las instancias correspondiente incluirá *Move-A-to-*B_1, *Move-A-to-*B_2, *Move-A-to-*B_3, ... A partir de esta definición, se define un **orden parcial de instancias de acciones** como una relación de orden sobre un conjunto finito arbitrario de instancias de acciones.

Siempre que no exista ambigüedad, se hablará de las instancias de acciones como si fueran acciones, haciendo referencia a sus precondiciones y efectos, o a su ejecución directamente, en vez de referirse a las acciones asociadas a ellas previamente.

También es necesario definir cómo serán representadas las relaciones de orden. Una relación de orden en un conjunto C se representará por un conjunto de desigualdades de la forma $x<y$; estas desigualdades deben ser compatibles entre sí, es decir, debe existir un orden en C en el que todas las desigualdades sean ciertas.

Un conjunto dado de desigualdades representará al orden más pequeño que lo contiene. Por ejemplo, dado el conjunto de acciones {1,2}, el conjunto de relaciones {1<2, 2<1} no representa ningún orden, mientras que {}, {1<2} y {2>1} representan los tres órdenes diferentes existentes.

1.5. Búsqueda a Través de un Espacio de Planes

En este caso, los nodos representan planes especificados parcialmente y los arcos representan operaciones de refinamiento de los planes, como por ejemplo el agregado de acciones. La Figura 5.5 muestra un ejemplo de espacio de búsqueda de planes. El nodo inicial es un plan vacío (ninguna acción) mientras que el final contiene el plan completo para resolver el problema. A diferencia de la búsqueda en el espacio de estados, los planificadores que trabajan con esta representación no retornan el camino seguido entre el nodo inicial y el final, sino que el nodo final es la solución.

[2] En su forma no determinista, ambos algoritmos harán n (número de pasos del plan) elecciones antes de construir el plan, si esto es posible.

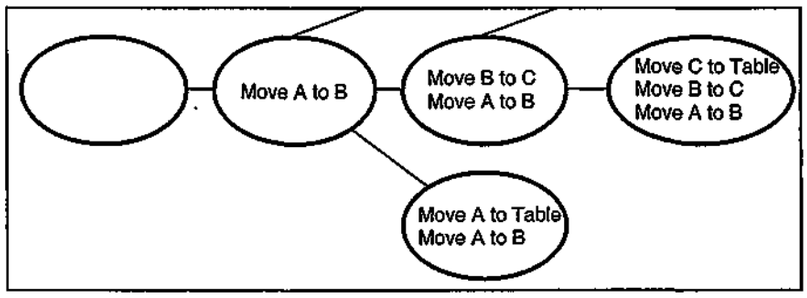


Figura 5.5 Espacio de búsqueda de planes

La ventaja de utilizar un espacio de planes reside en la facilidad para definir operadores de refinamiento y conduce a algoritmos más poderosos. Por ejemplo, se pueden adicionar acciones en puntos arbitrarios dentro de un plan. La clasificación de algoritmos progresivos y regresivos, establecida anteriormente, también es válida para los algoritmos que buscan en espacios de planes.

1.5.1. Orden total y parcial

Hasta el momento se ha definido a un plan como una secuencia de acciones que deben ser ejecutadas en cierto orden.

A medida que se construye un plan se puede establecer un orden total o parcial de acciones. La utilización de un orden total dentro de un plan parcial tiene la desventaja de comprometer las decisiones futuras; es decir, restringe totalmente el orden de ejecución de una acción con respecto a todas las demás. En cambio, el orden parcial no compromete decisiones futuras debido a que sólo las restricciones de orden esenciales son registradas. Esto resulta en una mayor flexibilidad. De hecho, dada una ordenación arbitraria de un conjunto finito, siempre existe al menos un orden total que es compatible con el inicial, y si la ordenación inicial es parcial pueden existir muchos órdenes totales compatibles con ella

1.5.2. Representación de planes parciales

En esta sección se define la estructura que se utiliza para representar a un plan parcial, pero antes se identifican sus componentes. Anteriormente se establecieron dos de los componentes básicos de un plan: un conjunto finito de instancias de acciones y un orden entre esas instancias. El tercer componente es un conjunto de **enlaces causales.**

Un enlace causal es una estructura que permite representar explícitamente las dependencias existentes entre las distintas acciones que conforman un plan. Esto surge de la necesidad de evitar que una nueva acción interfiera con alguna decisión pasada. Entonces, un enlace causal mantiene información sobre la razón por la cual una decisión fue tomada.

La estructura de un enlace causal está formada por tres partes. La primera es la instancia de acción productora del enlace, la segunda es la consumidora y la tercera es una proposición que las vincula. La proposición es un efecto de la acción productora y al mismo tiempo es una precondición de la consumidora. Formalmente un enlace causal se representa de la siguiente manera: $A_p \rightarrow^{Q} A_c$, donde A_p es la instancia de acción productora, A_c es la consumidora y Q es la proposición que las vincula. Este enlace representa la decisión tomada en el plan parcial correspondiente de efectuar la acción A_p con la finalidad de conseguir que se satisfaga la precondición Q de la acción A_c.

Sea por ejemplo A_c = *move-C-from-A-to-Table*$_2$. Una de las condiciones de A_c es que el bloque C esté libre (*clear C*). Sea A_p = *move-B-from-C-to-Table*$_5$, una instancia de acción que tiene como uno de sus efectos (*clear C*). Entonces si se utiliza A_p para obtener la precondición de A_c, debe crearse una enlace de la forma $A_p \rightarrow^{(clear\ C)} A_c$.

Formalmente, un **plan parcial** se representa como una tripla $<A,O,L>$ donde A es el conjunto de instancias de acciones del plan, es decir, instancias de acciones que se deben ejecutar en algún momento, sin ninguna indicación acerca de este instante; O es el conjunto de restricciones del orden de esas instancias, esto es, restricciones temporales en cuanto a qué acciones deben de preceder a qué otras; finalmente L es el conjunto de enlaces causales, los cuales representan información respecto a qué precondiciones de determinadas acciones a realizar se pretenden satisfacer con otras acciones previas.

Como ya se dijo, los enlaces causales son utilizados para detectar si una nueva acción interfiere con decisiones pasadas. En tal situación existirá al menos un enlace amenazado por el efecto de una nueva acción. Formalmente, sea $<A,O,L>$ un plan y A_n una nueva instancia de acción, entonces A_n **amenaza** el enlace $A_p \rightarrow^{Q} A_c$ si se cumplen las dos condiciones siguientes:

1. $O \cup \{ A_p < A_n, A_n < A_c \}$ define una relación de orden consistente, es decir, sería compatible con el orden parcial O el ejecutar A_n después de A_p y antes de A_c.
2. A_n tiene $not(Q)$ como un efecto.

Los algoritmos de planificación que utilizaremos utilizan mecanismos de **protección** de los enlaces amenazados. La protección de enlaces amenazados es una operación no determinista basada en dos posibilidades: *promoción y retracción*[3].

La *promoción* consiste en colocar una restricción de orden indicando que la instancia de acción conflictiva se ejecuta después de la instancia de acción consumidora del enlace. Al contrario, la *retracción* consiste en exigir que la instancia de acción conflictiva se ejecute antes que la productora del enlace. Formalmente, sea un plan $< A, O, L >$, A_n la instancia de acción conflictiva y $A_p \rightarrow^{Q} A_c$ el enlace amenazado, entonces existen dos alternativas:

- Promoción, colocando una restricción de orden $A_n > A_c$ en O
- Retracción, colocando una restricción de orden $A_n < A_p$ en O

Cualquiera de los dos mecanismos es aplicable siempre y cuando la incorporación de las nuevas restricciones de ordenación sigan definiendo una relación de orden entre las instancias de acciones.

1.5.3. Plan Inicial

Un punto importante es la representación del problema de planificación, es decir, cómo se modela la entrada de objetivos iniciales al algoritmo. Una forma simple y directa es utilizar un plan parcial inicial cuya compleción represente una solución del problema planteado. Este plan parcial inicial representa conjuntamente la descripción del estado inicial y los objetivos.

Para esto se definen dos acciones estándar, A_0 **(acción inicial)** y A_∞ **(acción final)**, de las que solamente habrá una instancia que podemos identificar con la acción en si, y un conjunto básico de restricciones de orden $O = \{(A_0 < A_\infty)\}$. Si se hace $A = \{A_0, A_\infty\}$, entonces el plan parcial inicial estará representado por $<A, O, L>$, donde el conjunto de enlaces causales L es vacío. La acción inicial A_0 no tiene precondiciones y su efecto está formado por el conjunto de proposiciones que describen el estado del mundo inicial. En cambio la acción final A_∞ no tiene efecto y su precondición es el conjunto de proposiciones que describen los objetivos finales.

1.6. El Algoritmo *POP*

El algoritmo regresivo *POP* [Wel94] realiza la búsqueda a través del espacio de planes parcialmente especificados, comenzando por un plan nulo y utilizando orden parcial entre acciones. Esta búsqueda se basa en operaciones de refinamiento como promoción y retracción.

[3] *promoción* y *retracción* en el sentido de adelantar y retrasar

El algoritmo *POP*, presentado en la Figura 5.6, recibe como entrada tres elementos: el plan parcial obtenido hasta el momento (al comienzo el inicial correspondiente al problema que se trata de resolver), la **agenda** de objetivos pendientes (según se explica a continuación, cada objetivo va acompañado de cierta información adicional; inicialmente, la agenda está formada por todos los objetivos que se pretenden alcanzar) y la teoría del dominio (Δ). Con la finalidad de poder mantener la cuenta de los enlaces causales a medida que se genera el plan, cada objetivo de la agenda, que se introduce en la misma en función de que es una precondición de una acción que se planea ejecutar, se mantiene en la agenda acompañado de la instancia de acción correspondiente. Así, si los objetivos iniciales son $\{O_1, O_2\}$, entonces la agenda inicialmente será igual a $\{< O_1, A_\infty >, < O_2, A_\infty >\}$. Además, se puede observar que por la forma en que se define el propio algoritmo POP, si $<Q, A_{need}>$ es un elemento cualquiera de la agenda se cumplirá automáticamente que Q es un elemento de la precondición de A_{need} y $A_{need} \in A$. La semántica de la agenda es que queremos garantizar que sus objetivos se satisfacen en el momento en que se ejecute la instancia de acción correspondiente.

POP ($<A,O,L>$, *agenda*, Δ)

1) **Terminación:** Si agenda está vacía retornar el plan <A,O,L>
2) **Selección de objetivo:** elegir un par $< Q, A_{need} >$ de la agenda
3) **Selección de una instancia de acción:** Sea A_{add} = *seleccionar* una instancia de una acción cuyo efecto hace que Q se cumpla. Además, $A_0 < A_{add} < A_{need}$ debe de ser compatible con el orden O en el sentido de que $O' = O \cup \{A_0 < A_{add}, A_{add} < A_{need}\}$ debe definir una nueva relación de orden. A_{add} puede ser una instancia de una acción del universo que no se ha instanciado con anterioridad o puede ser una instancia de una acción ya instanciada previamente en el plan, en cuyo caso puede a su vez ser una instancia ya existente o una nueva (salvo en el caso de las acciones inicial o final, que solamente se pueden instanciar una vez). Si tal elección no es posible retornar fracaso.
4) **Actualización del Plan Parcial:**
   a) Hacer $A' = A \cup \{A_{add}\}$, $L' = L \cup \{A_{add} \rightarrow^{Q} A_{need}\}$ y, como se indicó en el punto anterior, $O' = O \cup \{A_0 < A_{add}, A_{add} < A_{need}\}$.
   b) Actualización de la agenda: hacer $agenda' = agenda - \{< Q, A_{need} >\}$. Si A_{add} es una nueva instancia entonces para cada precondición Q_i agregar $<Q_i, A_{add}>$ a *agenda'*.
   c) Protección de enlaces causales: para cada instancia de acción $A_t \in A'$ que amenace un enlace causal $A_p \rightarrow^{Q} A_c$ de L', proteger el enlace de la amenaza.
5) **Invocación recursiva:** POP($<A',O',L'>$,*agenda'*, Δ)

Figura 5.6 Algoritmo *POP* (adaptado de [Wel94])

El algoritmo *POP*, presentado en la Figura 5.6, recibe como entrada tres elementos: el plan obtenido hasta el momento (inicialmente nulo), la agenda de objetivos pendientes (inicialmente, todos los objetivos que se pretenden alcanzar) y la teoría del dominio (Δ). La agenda debe tener el mismo formato utilizado para el plan. Por este motivo si los objetivos iniciales son $\{O_1, O_2\}$, entonces la agenda inicialmente será igual a $\{< O_1, A_\infty >, < O_2, A_\infty >\}$.

1.7. El Algoritmo *UCPOP*

El algoritmo *UCPOP* [Wel94] es una generalización del *POP* y, al igual que éste, utiliza orden parcial entre acciones. Adicionalmente, introduce algunas mejoras tales como parametrización de acciones, efectos condicionales y cuantificación universal. En las siguientes secciones se describen las modificaciones introducidas, comenzando por las acciones parametrizadas.

1.7.1. Acciones parametrizadas

Hasta ahora el universo de acciones posibles está formado por acciones específicas. En el ejemplo de los cubos, una acción posible sería *move-B-from-Table-to-C*. Una forma más flexible sería permitir que, si se necesita mover el bloque *B* a *C*, no sea necesario especificar inmediatamente de dónde se lo toma. Para esto sería conveniente no definir acciones, sino esquemas de acciones con variables. Esta descripción abstracta de las acciones o **parametrización** también redunda en una mayor eficiencia y sencillez de definición del universo de acciones posibles.

La parametrización de acciones, básicamente, consiste en definir en forma genérica a las mismas. Por ejemplo en la Figura 5.7 se muestra una definición parametrizada de la acción *move*. A partir de esta acción genérica pueden crearse instancias de acciones para mover cualquier bloque desde cualquier origen a cualquier destino (excepto la mesa). Esta descripción es mucho más económica que definir una acción particular para cada combinación de bloques.

Operador: move

parámetros: (?b ?x ?y)

precondición: (and (on ?b ?x) (clear ?b) (clear ?y)

(≠ ?b ?x) (≠ ?b ?y) (≠ ?y ?x) (≠ ?y Table))

postcondición: (and (on ?b ?y) (not (on ?b ?x)) (clear ?x) (not (clear ?y)))

Figura 5.7 Definición del operador *move* con variables

Cuando una acción con parámetros es seleccionada para formar parte del plan, se le deben asignar a los parámetros aquellos valores que sean necesarios para la construcción del plan. Por ejemplo, si la acción de la Figura 5.7 es seleccionada para satisfacer un objetivo (*on M N*), entonces el parámetro ?b debe ser asociado con M y ?y con N. En este caso, todavía no hay información para fijar el valor de ?x.

Para poder establecer restricciones sobre los valores que un parámetro de una acción puede tomar, se definen restricciones de codesignación y no-codesignación. En el ejemplo anterior, podemos restringir el valor de *?y* por medio de la restricción de no-codesignación (≠ *?y Table*) indicando que el parámetro puede asumir cualquier valor distinto de *Table*. Al contrario, una restricción (= *?x ?y*) indicaría que *x* e *y* deben tener el mismo valor. Las restricciones de codesignación y no-codesignación forman parte de la precondición de una acción, aunque no son predicados sobre el estado del mundo sino sobre el estado del proceso de planificación.

El algoritmo de planificación utiliza, ahora, acciones parcialmente instanciadas porque el valor de algunos de sus parámetros no está determinado en el momento en que se elige la acción. El valor de un parámetro puede ser el producto de subsecuentes restricciones, surgidas de las acciones que se van agregando al plan, hasta que eventualmente se vincule a un único valor.

A continuación se describen las modificaciones que deben ser introducidas al algoritmo *POP* para utilizar acciones parcialmente instanciadas y restricciones de codesignación y no-codesignación.

1. Debe ser posible representar en el plan el conjunto de las codesignaciones o vínculos entre variables y valores (*bindings*). Entonces un plan consiste ahora en una *n_upla* <*A,O,L,B*> donde *B* es la tabla donde se almacenan los vínculos y las restricciones de codesignación y no-codesignación. En el plan inicial, $B = \{\}$.

2. Es necesario definir una función *MGU(Q,R,B)* que retorne la unificación más general de los literales Q y R respecto a las restricciones de codesignación existentes en B. Formalmente, *MGU(Q,R,B)* = U, donde U tiene dos posibles valores:

   - U es un conjunto de pares (u_i, v_i) indicando que u_i y v_i deben codesignar para que Q y R unifiquen.
   - $U = \perp$ (nulo) si no unifican Q y R en B.

   Por ejemplo:

   MGU((on ?x B), (on A B), {}) = {(?x, A)}

   MGU((on ?x B), (on A B), {(?x, C)}) = $\perp$, porque A no unifica con C.

   MGU((on ?z B), (on C ?y), {(?x, C), (≠ ?y ?x)}) = {(?x, C), (?y, B) , (?z, C), (≠ ?y ?x)}

   Si $B = \{\}$, entonces *MGU (Q,R,B)* se puede escribir *MGU (Q,R)*.

3. Sea Δ una **sentencia lógica** (es decir, una conjunción de términos), la operación: $\Delta \setminus B$, retorna la sentencia lógica que resulta de sustituir las variables por literales básicos (*ground*) de acuerdo a las codesignaciones B. Es importante notar que B puede contener restricciones de no-codesignación.

4. La selección de una acción A_{add} (paso 3 de *POP*) debe considerar todas las acciones tales que algún efecto E de A_{add} unifica con el objetivo Q de acuerdo a las restricciones de codesignación del plan. Formalmente, considera una acción A tal que $MGU(Q,E,B) \neq \perp$ donde E es uno de los efectos conjuntos de A. Por ejemplo, si Q = *(on L M)*, será valida una acción con efecto *(on ?x ?y)* si $B = \{\}$, pero no será válida si B = *{(?x, N)}*.

5. Cuando una nueva instancia de acción es creada a partir de una acción abstracta (con variables) del universo, se tiene que asegurar que las variables no han sido previamente usadas en el plan. Entonces puede ser necesario cambiar los nombres de las variables de una acción abstracta. Es decir, al incorporar por ejemplo una instancia de una acción con efecto *(on ?x ?y)*, transformarlo en *(on ?x1 ?y1)*, suponiendo que *?x1* e *?y1* no se hayan utilizado anteriormente.

6. Actualización del conjunto de objetivos: las precondiciones de las acciones pueden ser tanto restricciones de no-codesignación (como por ejemplo (≠ *?b ?x*)) como precondiciones lógicas (por ejemplo (*clear M*)). Sólo las precondiciones lógicas deben ser agregadas a la agenda.

7. Actualización de la tabla de vínculos B: deben agregarse a B el resultado de la unificación del efecto E con el objetivo Q, para asegurarse que el efecto soporte la condición. De la misma forma, deben agregarse a B las restricciones de no-codesignación incluidas en la precondición de la nueva acción.

8. El algoritmo *POP* utiliza *promoción* o *retracción* como alternativas de protección de enlaces causales amenazados. Ahora, como el algoritmo maneja variables, la cuestión es cómo definir si una acción amenaza a un enlace causal. Por ejemplo, es necesario determinar si una acción A_t que tiene como efecto (*not (clear ?y1)*), puede amenazar a un vínculo etiquetado (*clear M*). A menos que el plan tenga una restricción de no-codesignación *?y1*≠ B, en algún momento puede agregarse la codesignación (*?y1, B*), con lo que la amenaza existiría. De todas formas, por razones de eficiencia y simplicidad [Wel94] es mejor esperar que la unificación (*?y1, B*) se produzca, si es que eso sucede. En ese momento es cuando el planificador debe decidir si usar promoción o retracción. Es decir, que la protección de enlaces debe aplicarse sólo con las precondiciones que tengan sus variables instanciadas con valores concretos

9. La condición de terminación de *POP* (si la agenda está vacía) es insuficiente porque ahora es necesario que todas las variables tengan asignado un valor para asegurar que todos los

enlaces causales amenazados han sido reconocidos; esto es debido al punto anterior. Esta última condición implica que el estado inicial no contenga variables y, además, que todas las variables nombradas en el efecto de una acción aparezcan en la precondición de la misma.

1.7.2. Efectos condicionales

Para lograr una mayor expresividad y eficiencia del lenguaje utilizado para definir efectos, se introduce el concepto de **efecto condicional**. Un efecto condicional es una proposición, a la que se denomina *when*, compuesta por dos argumentos: un **antecedente** y un **consecuente**. El antecedente se refiere al estado del mundo antes de ejecutar la acción, mientras que el consecuente se refiere al estado del mundo después de ejecutar la misma. Entonces, la acción tendrá el efecto del consecuente en el caso de que el antecedente sea verdadero.

Volviendo al ejemplo de la Figura 5.7, esta definición del operador *move* no abarca el caso que el destino del bloque sea la mesa, porque en este caso no se puede afirmar como efecto *(not (clear ?y))*. Esto no sólo obliga a una definición separada para cuando el destino sea la mesa, sino que obliga al algoritmo a tomar una decisión inmediata, aunque tal vez el destino no era importante en ese punto de la planificación.

Los efectos condicionales buscan solucionar este problema, permitiendo que alguna parte del efecto dependa de ciertas condiciones en la precondición. Un operador *move* más genérico podría ser definido como en la Figura 5.8.

La planificación con efectos condicionales requiere introducir dos modificaciones al algoritmo *POP*. La primera es que se permite seleccionar una instancia de acción que tiene un efecto condicional, cuyo consecuente unifica con el objetivo a cumplir *Q*. En este caso se tiene que asegurar que el antecedente del efecto es verdadero para que se cumpla el consecuente y por lo tanto el enlace causal; para esto se agrega el antecedente a la agenda, junto con la instancia de acción, como un objetivo pendiente.

Operador: move

parámetros: (?b ?x ?y)

precondición: (and (on ?b ?x) (clear ?b) (clear ?y)

(≠ ?b ?x) (≠ ?b ?y) (≠ ?y ?x))

postcondición: (and (on ?b ?y) (not (on ?b ?x)) (clear ?x)

(when (≠ ?y Table) (not (clear ?y))))

Figura 5.8 Operador *move* con efectos condicionales

La segunda modificación es en la definición de las amenazas de enlaces causales y en su protección. Por una parte se admiten las amenazas por el consecuente de un efecto condicional. En ese caso la protección del enlace puede consistir de forma indeterminista en la *promoción*, la *retracción* y un nuevo mecanismo llamado *confrontación*. La *confrontación* consiste en agregar la negación del antecedente del efecto condicional a la agenda junto con la instancia de acción correspondiente; esto asegura que la parte condicional del efecto (o consecuente del efecto condicional) no se aplique y por lo tanto el enlace no sea amenazado.

Debe notarse que la inclusión de literales negados en la agenda no crea problemas, obligando únicamente a considerar a todos los efectos los literales correspondientes sin negar como su negación. Por otro lado, la presencia simultánea en la agenda de un literal y su

negación no dan lugar, en principio, a una contradicción. Esto se debe a que, como se explicó al introducir el algoritmo *POP*, la semántica de la agenda es que los objetivos que contiene deben satisfacerse en el momento de la ejecución de las acciones correspondientes. Así pues, únicamente se pueden calificar como estados de fracaso aquéllos en que aparecen en la agenda sendos literales incompatibles asociados a la misma acción.

1.7.3. Precondiciones disyuntivas

Para poder trabajar con precondiciones disyuntivas es necesario realizar una modificación al algoritmo *POP*. Cuando se selecciona el objetivo Q de la agenda en la línea 2 de *POP*, si Q = (OR Q_1 Q_2) entonces se elimina Q de la agenda y se selecciona un solo subobjetivo (Q_1 ó Q_2) de Q. El subobjetivo seleccionado debe ser agregado a la agenda porque éste, a su vez, puede ser otra disyunción. Aquí de nuevo se recurre a la utilización de una función no determinista para seleccionar el subobjetivo que se utilizará. Esto también se hace para ocultar los detalles que implica una elección de este tipo y al igual que la función seleccionar, se supone que elige la mejor alternativa: si existe una solución, será encontrada.

Esto permite representar disyunciones en las precondiciones y en los antecedentes de efectos condicionales, pero no efectos disyuntivos, los que implican un tratamiento mucho más complicado [Wel94]. Los efectos disyuntivos permitirían representar una acción no determinista, es decir, una acción cuyos efectos son aleatorios (por ejemplo, el lanzamiento de una moneda). Este tipo de efectos no resultan de interés en el caso de la instanciación de *frameworks*.

Estrictamente hablando, la presencia de precondiciones disyuntivas no aumenta necesariamente el poder de expresión del lenguaje utilizado para representar las acciones, en el sentido de que no hay nada que pueda se expresado exclusivamente utilizando disyunciones. Por ejemplo, si se quiere expresar que la precondición de una acción es (*or (on A C) (on B C)*) y no se dispone de precondiciones disyuntivas, es posible hacerlo utilizando dos acciones con el mismo efecto, una con precondición (*on A C*) y otra con (*on B C*).

1.7.4. Efectos cuantificados universalmente

Para poder utilizar la cuantificación universal en el algoritmo de planificación necesitamos establecer algunas simplificaciones. En primer lugar, todos los objetos deberán tener un tipo asociado, cuya definición se incluye en el estado inicial del plan (efecto de A_0).

La segunda simplificación es establecer un universo finito y estático de objetos, esto es, ninguna acción puede crear o destruir objetos. Un efecto cuantificado universalmente es utilizado para definir aquellas acciones que provocan un cambio de estado en varios objetos a la vez. Por ejemplo al mover un maletín de un lugar a otro, se mueven todos lo objetos contenidos dentro de él

1.7.5. La base universal

Para asegurar el establecimiento sistemático de objetivos (precondiciones) que tienen cuantificadores universales, *UCPOP* convierte estas expresiones en su equivalente en cláusulas básicas (*ground*).

Formalmente, se define la base universal de primer orden Y de una sentencia libre de función Δ como:

$$Y(\Delta) = \Delta, \text{ si } \Delta \text{ no contiene cuantificadores}$$

$$Y(\text{forAll } ((T_1\ ?x))\ \Delta_{?x}) = Y(\Delta_{x1}) \wedge Y(\Delta_{x2}) \wedge \ldots\ldots \wedge Y(\Delta_{xn})$$

donde Δ_{xi} corresponde a cada posible interpretación de $\Delta_{?x}$ bajo el universo de discurso $\{x_1,x_2,...,x_n\}$ de los posibles objetos de tipo T_1. Es decir, en cada Δ_{xi} las referencias a ?*x* han sido reemplazadas por la constante x_i correspondiente.

Por ejemplo suponiendo un universo de libros {*Física*, *Matemáticas*, *Lengua*} entonces si Δ es (*forAll* ((*libro* ?y)) (*in* ?y B)) la base universal Y(Δ) es la siguiente conjunción ((*in Física* B) *and* (*in Matemáticas* B) *and* (*in Lengua* B)).

En consecuencia se denomina *cálculo de la base universal* al reemplazo de la expresión cuantificada universalmente por la conjunción de las expresiones básicas equivalentes.

1.8. Definición del Algoritmo *UCPOP*

Como se ha dicho anteriormente, el algoritmo *UCPOP* es una generalización del *POP* que permite utilizar acciones parcialmente instanciadas, precondiciones disyuntivas, efectos condicionales y cuantificación universal. La Figura 5.9 (página siguiente) presenta el algoritmo *UCPOP*. Resumidamente, la descripción del algoritmo es la siguiente:

1) Si el objetivo o una precondición es una sentencia universalmente cuantificada entonces calcular la base universal y aplicar el algoritmo de planificación sobre la misma.
2) La base universal de un efecto cuantificado universalmente es computada gradualmente en la medida que es utilizado para establecer enlaces causales. Esto es por razones de eficiencia.
3) La definición de amenaza de un enlace causal es cambiada. Formalmente, sea una acción A_t y un enlace causal $A_p \rightarrow^Q A_c$ entonces A_t amenaza al enlace si:
   a) $A_p < A_t < A_c$ es consistente con O
   b) A_t tiene un efecto R (R puede ser el consecuente de un efecto condicional).
   c) La unificación MGU(Q, *not*(R), B) $\neq \perp$.
   d) Para todo par (u, v) $\in$ MGU(Q, *not*(R), B) o bien u o v forma parte de las variables cuantificadas universalmente.

2. *PIT*: Planificación del Proceso de Instanciación de *Frameworks*

Las ideas de planificación presentadas se adaptan muy bien al problema de generación de planes de instanciación. En este caso, los planes no son generados para un agente artificial, sino para el usuario que quiere utilizar el *framework* para construir una aplicación.

El algoritmo *UCPOP* visto en la sección anterior es uno de los algoritmos más comúnmente utilizados para resolver problemas generales de planificación. Aunque este algoritmo satisface en gran medida los requisitos para la generación de planes de instanciación, cuando analizamos las necesidades específicas de este dominio, surgen características propias que deben ser tenidas en cuenta.

Una de las características más importantes del dominio de instanciación de *frameworks* es que no siempre toda la información necesaria para generar el plan está disponible desde el primer momento. Parte de la información no disponible puede ser solicitada al usuario durante el proceso de planificación; son decisiones que debe tomar el usuario y de acuerdo a estas decisiones será la forma en que prosiga la planificación. En otras ocasiones, la información disponible no será suficiente para generar un plan completo que permita alcanzar todos los objetivos pretendidos por el usuario.

Como se verá más adelante, el usuario elige la funcionalidad en base a las opciones presentadas por el sistema. Estas opciones son construidas a partir de la funcionalidad documentada por el diseñador. Entonces pueden ocurrir dos tipos de situaciones en las que no

sea posible crear un plan para satisfacer todos los objetivos del usuario: o bien las opciones ofrecidas por el sistema no son suficientes para expresar toda la funcionalidad requerida, o bien no es posible construir un plan para la combinación específica de objetivos definidos por el usuario. Esto último puede deberse tanto a limitaciones reales del *framework* como a que el sistema de planificación no es capaz de encontrar un plan para ello.

UCPOP (*<A,O,L,B>, agenda, Δ*)

1) **Terminación:** Si agenda está vacía retornar el plan *<A,O,L,B>*.
2) **Reducción de objetivo:** seleccionar y borrar un objetivo $<Q, A_c>$ de *agenda*
   a) Si Q es sentencia cuantificada universalmente, calcular la base universal e insertarla en la agenda y retornar a 2.
   b) Si Q es una conjunción de Q_i entonces colocar cada $<Q_i, A_c>$ en la agenda y volver a 2.
   c) Si Q es una disyunción de Q_i entonces elegir un Q_k , colocar$<Q_k, A_c>$en la agenda y volver a 2.
   d) Si Q es un literal y existe un enlace $A_p \rightarrow^{not(Q)} A_c$ en L, entonces fallar (es imposible elaborar un plan).
3) **Selección de una acción:** Sea Ap = *seleccionar* una instancia de acción nueva o ya existente en el plan, tal que $Ap < Ac$ es consistente con O y un literal R del efecto de la acción unifica con Q dado B. Si tal elección no existe retornar fallo. De lo contrario seguir con 4)
4) **Actualización del Plan:**
   a) $A' = A + A_p$,
   b) $L' = L \cup \{A_p \rightarrow^{Q} A_c\}$
   c) $B' = B \cup \{(u,v) | (u,v) \in MGU(Q,R,B) \wedge u,v$ no son variables cuantificadas del efecto$\}$.
   d) $O' = O \cup \{A_p < A_c\} \cup \{A_0 < A_p < A_\infty\}$.
   e) agregar las restricciones de no codesignación (A_p) a B'.
5) **Actualización de la Agenda:**
   a) *agenda'* = *agenda* + <precondiciones lógicas (A_p) \ MGU(Q,R,B), A_p>
   b) Si el efecto es condicional y no ha sido usado para establecer un enlace en L, entonces agregar su antecedente a *agenda'* después de sustituir con MGU(Q,R,B).
6) **Protección de enlaces causales:** para cada enlace $l = A_i \rightarrow^{P} A_j$ en L y por cada acción $A_t \in A'$ que pueda amenazar el enlace causal l seleccionar una de las siguientes opciones:
   a) Retracción: agregar $A_t < A_i$ a O'
   b) Promoción: agregar $A_j < A_t$ a O'.
   c) Confrontación: si el efecto de A_t que amenaza el enlace es condicional con antecedente S y consecuente R, entonces agregar <not(S)\MGU(P, not(R)) , A_t > a *agenda'*.
   d) Si ninguna opción es aplicable retornar fallo
7) **Invocación recursiva:** Si B es inconsistente entonces fallar; en caso contrario invocar UCPOP(*<A',O',L',B'>, agenda', Δ*)

Figura 5.9 Algoritmo *UCPOP* (adaptado de [Weld94]).

En estos casos, es importante que el algoritmo de planificación permita el manejo de planes parciales; debe ser capaz de retornar el plan más completo posible en función de la documentación disponible, dejando al usuario aquellos aspectos no contemplados en la documentación de planificación. Cuando esto sucede, no es posible decir nada sobre las

posibilidades de completar el plan parcial para implementar una aplicación que cumpla con todos los objetivos funcionales. Es decir, la única certeza es que no se sabe cómo implementarlo, no que no se pueda implementar.

El plan generado también debe ser parcial en el sentido dado por *UCPOP* al término: debe ser posible que la secuencia de tareas generada no esté totalmente ordenada, para brindarle al usuario toda la flexibilidad posible. De esta forma, siempre que no exista una restricción de orden entre dos tareas dadas, el usuario tendrá la posibilidad de elegir cuál ejecutar primero.

El algoritmo además debe proveer facilidades para actualizar dinámicamente los objetivos y la información del *framework*. Es decir, la planificación debe permitir la modificación de los datos de entrada al algoritmo, tanto de los objetivos como la información relativa al *framework*, en cualquier momento del proceso, ya sea durante la planificación o durante la ejecución del plan. Por ejemplo, puede ocurrir que durante el proceso de instanciación el usuario decida cambiar los objetivos funcionales que originalmente había previsto para su aplicación. Otro caso, menos frecuente, puede darse al recibirse información actualizada del *framework* o más aún, al modificarse el *framework* mismo. Estas características plantean la necesidad de un algoritmo que permita "replanificar" en función del cambio en las entradas. Esta replanificación no debe descartar toda la información anterior. Si algunas tareas ya habían sido ejecutadas como parte de un plan viejo, y estas tareas también son necesarias en el nuevo plan, el sistema debe ser capaz de reconocer su existencia y evitarle al usuario el trabajo de realizar las mismas tareas más de una vez.

En función de estos requisitos, el algoritmo *UCPOP* fue extendido para soportar no sólo planificación parcial, es decir que la secuencia de tareas no está totalmente ordenada, sino también planificación incremental. El nuevo algoritmo es denominado ***PIT*** (*Planning Instantiation Tasks*, Planificación de Tareas de Instanciación). La planificación incremental implica que la generación del plan de instanciación no se hace completamente de una sola vez, sino que es hecha gradualmente, de acuerdo a las tareas ejecutadas por el usuario. En algunos momentos, la generación del plan es detenida, a la espera de información ingresada por el usuario. Más aún, el usuario puede definir de forma incremental sus objetivos, o modificar decisiones anteriores y la generación del plan se actualiza en correspondencia.

Esto implica la necesidad de reconsiderar decisiones ya tomadas. Cada vez que el plan actual debe ser modificado, el proceso de planificación comienza desde el principio. Sin embargo, en estos casos, se conserva información sobre todas las actividades que han sido ejecutadas como parte del plan anterior, para evitar que el mismo trabajo deba ser hecho más de una vez.

Otra modificación importante respecto de *UCPOP* es que el algoritmo *PIT* utiliza el mecanismo de *backtracking*. Si bien *PIT* es definido independientemente de la implementación de la función *seleccionar*, no presume que *seleccionar* hará siempre la mejor elección. Por esto, el algoritmo prevé que al llegar a puntos muertos, se retroceda al último punto donde se tomó una decisión y se pruebe con otra opción.

2.1. Relajación de Restricciones

Una de las características del algoritmo original que fue conservada es la minimización de compromisos. Es decir, siempre que el algoritmo debe elegir uno entre posibles cursos de acción, lo hace adoptando la menor cantidad necesaria de decisiones. Expresado en otros términos, las decisiones son retrasadas tanto como sea posible en el proceso de planificación. Sin embargo, a diferencia de *UCPOP*, *PIT* no necesariamente termina en un estado donde todas las decisiones han sido tomadas, o lo que es lo mismo, todas las variables tengan valores asignados. Es decir, puede darse el caso donde el plan de instanciación tenga variables sin asignar.

Esta propiedad resulta útil por varios motivos. En primer lugar, es posible modelar situaciones en las cuales el usuario debe proveer ciertos valores, ya sea porque el algoritmo es incapaz de determinarlos o simplemente porque se le quiere dar al usuario la posibilidad de elegir. Por ejemplo, la mayor parte de las veces en las que el plan incluya una tarea de creación de una clase, el nombre de la clase no estará fijado; ya en el caso de la creación de métodos, es mucho más frecuente el caso donde el método debe ser creado con un nombre específico, siendo por lo tanto una variable con un valor asignado.

Además, el uso de variables libres al finalizar el proceso de planificación permite omitir la restricción impuesta por *UCPOP* de un universo finito y estático. Es fácil ver que si el usuario tiene la posibilidad de crear objetos (clases y métodos, por ejemplo) esta restricción no puede ser mantenida. En consecuencia, se hace necesario que el lenguaje permita manipular objetos que no existen en el momento de la planificación y que no pueden ser nombrados explícitamente. En estos casos también resulta de utilidad que al finalizar la planificación las variables que representan esos objetos estén sin instanciar y se les asocie un valor sólo en el momento en que los objetos correspondientes sean creados.

Un universo dinámico, sin embargo, complica la implementación de las variables universalmente cuantificadas. Debe recordarse que la forma de implementar esta funcionalidad es reemplazando cada ocurrencia de la variable por todos los objetos pertenecientes al tipo asociado. Sin embargo, si se considera por ejemplo un predicado que hace referencia a todas las subclases de una clase determinada, sólo se podrá incluir en el plan aquellas subclases que ya existan en el momento de planificar.

Por este motivo, la cuantificación universal es provista a través de un operador (descrito en la próxima sección) y está principalmente orientada a utilizar listas explícitas de objetos y no tipos, como en el caso de *UCPOP*.

El método *SmartBooks* requiere, además, que otras restricciones sean retiradas, como por ejemplo la ejecución atómica de las tareas de instanciación y el hecho de que el plan sea la única causa de cambio. La primera de estas restricciones ha sido relajada a través de la representación del plan, tal como se explica en la próxima sección. La propiedad de ser única causa de cambio es eliminada utilizando un mecanismo de administración de tareas, explicado en el próximo capítulo.

Para describir el funcionamiento de *PIT*, a continuación se comienza explicando las representaciones utilizadas para la documentación del *framework*, de los objetivos funcionales y de los planes de instanciación generados.

2.2. Representación de Información de Diseño

En el caso de los planes de instanciación de *frameworks*, la teoría del dominio corresponde a las acciones que puede efectuar el usuario para crear su aplicación utilizando el *framework*. La representación de estas acciones es hecha a través de **acciones de instanciación**. Estas acciones de instanciación se basan en la representación propuesta para el algoritmo *UCPOP*, modificada para permitir las extensiones necesarias para generar planes de instanciación. Esto quiere decir que las acciones se representan, básicamente, por medio de precondiciones y efectos.

Las acciones de instanciación se dividen en dos grupos: **generales** o **primitivas** y específicas. Las acciones específicas son aquellas que el diseñador de un *framework* define como parte de la documentación de ese *framework*. Describen las acciones necesarias para obtener la funcionalidad provista por el *framework*. Las acciones primitivas, por su parte, son independientes del *framework* y constituyen un conjunto fijo, que no puede ser modificado por el diseñador de un *framework*. Estas acciones describen, por ejemplo, cómo puede ser utilizada una clase de un *framework* arbitrario, bien a través de su instanciación directa, bien a través de

una especialización. En el Anexo A se ofrece una lista de las acciones primitivas disponibles para la definición de las acciones específicas de cada *framework*.

Es necesario destacar que las acciones de instanciación no son equivalentes a las tareas de instanciación que componen el plan presentado al usuario. Estos conceptos fueron disociados para ofrecer una mayor flexibilidad y sobre todo facilitar la descripción de las acciones. Si bien desde el punto de vista del usuario el resultado del proceso de planificación es una lista de tareas de instanciación a ejecutar, esta lista de tareas no es proporcionada directamente por el planificador sino que es construida a partir del plan resultante. En la figura 4.2 es posible ver la relación entre estos conceptos. En este capítulo se enfocará sólo la representación de las acciones de instanciación.

Una ventaja adicional obtenida al diferenciar los conceptos de acción y tarea de instanciación es que se elimina la restricción de que las acciones deben ser ejecutadas en un tiempo atómico. Más concretamente, la restricción carece de sentido al no corresponder las acciones con ninguna actividad que deba ejecutar el usuario. Las tareas, por su parte, pueden tener sus lapsos de ejecución superpuestos en el tiempo. Los inconvenientes que se pueden derivar de esta ejecución en paralelo son evitados a través de un agente de administración de tareas, explicado en el próximo capítulo.

Por otra parte, al no corresponder a actividades concretas del usuario, no es conveniente hablar de la *ejecución* de acciones, ya que nada es ejecutado en realidad. En lugar de eso, se utilizará el término *aplicar* para describir que una acción es utilizada en el plan para conseguir satisfacer una condición necesaria. Es decir, siempre que una acción sea incluida en el plan, se dirá que es *aplicada*.

2.2.1. Acciones de instanciación

El objetivo de las acciones de instanciación su utilización por el planificador para generar los planes de instanciación. La forma general de una acción de instanciación es la siguiente:

```
precondiciones → efecto
```

Tanto las precondiciones como el efecto pueden ser una conjunción de términos, los cuales pueden contener variables. La acción sólo puede ser aplicada cuando las precondiciones son verdaderas; cuando la acción es aplicada, todos los términos positivos de la conjunción que describe el efecto son agregados a la descripción del estado del mundo, mientras que todos los negativos son quitados. En términos de la instanciación del *framework*, el estado del mundo representa el estado de la aplicación en desarrollo.

El siguiente ejemplo de acción de instanciación describe lo que debe hacerse en el *framework HotDraw* para permitir en la aplicación resultante la edición de atributos de una figura a través de una herramienta (*tool*) (ver §II.2.3 para una explicación de *HotDraw*). Para alcanzar este objetivo, debe utilizarse la clase *CompositeFigure*, creando una subclase de ella. Además, debe inicializarse el atributo para que pueda ser editado. Finalmente, deben ejecutarse las acciones necesarias para usar una herramienta para editar atributos. Otras acciones, de las que se verán ejemplos más adelante, describirán cómo deben ser alcanzados estos subobjetivos.

```
defineSubclass( ClassName, "CompositeFigure", "An attribute has to be edited" ),
    specificInitialization( "edit", ClassName, AttributeName, Description ), useTool( "editAttribute" )
    → editAttribute( "Tool", ClassName, AttributeName )
```

Utilizando esta acción de instanciación, cuando el planificador encuentre entre los objetivos funcionales la edición de atributos de figuras utilizando las herramientas, creará las tareas necesarias.

En esta especificación, los términos entre comillas denotan constantes, en tanto que los términos en mayúsculas y sin comillas representan variables. Un ejemplo de variable es

ClassName, la cual es utilizada para representar el tipo de figuras cuyos atributos deben ser editados interactivamente.

El uso de variables no sólo permite aumentar el poder de expresión del lenguaje utilizado para representar las acciones. También le otorga un grado de informalidad que facilita la definición de las acciones. Por ejemplo, las variables son utilizadas sin especificar el tipo de objeto que cada variable puede representar. Este tipo es deducido en el momento en que la variable deba ser asociada con un valor concreto. Por ejemplo, la variable *ClassName* de la acción anterior será asociada con un nombre de clase porque al intentar resolver el objetivo *defineSubclass* los únicos valores que unificarán serán los nombres de clases. Esto si bien quita formalidad a la definición de acciones, no crea ambigüedades a la hora de interpretar el significado de cada una.

Hasta el momento se han visto dos tipos de términos que pueden formar parte de las precondiciones de una acción:

- objetivos definidos por el diseñador: son los objetivos introducidos por el autor de la documentación del *framework* para describir su funcionalidad. Deben tener asociadas una o más acciones de instanciación que describan cómo pueden ser alcanzados. En la acción vista anteriormente, *specificInitialization*, *useTool* y *editAttribute* son objetivos de este tipo.
- objetivos primitivos: estos objetivos son alcanzados a través de acciones primitivas. Las precondiciones de las acciones primitivas son siempre objetivos primitivos. El término *defineSubclass* visto en el ejemplo anterior es un objetivo primitivo.

Tanto los objetivos definidos por el programador como los primitivos pueden ser utilizados en número y orden arbitrario para formar las precondiciones de una acción.

Las siguientes acciones son ejemplos de objetivos y acciones primitivas. Estas acciones describen que la utilización de una componente puede hacerse ya sea reutilizando la componente tal como está o especializándola a través de una subclase.

```
selection(["as-is", "refine"], Description, Return), useComponent (Component, Description, Return)
        → tryUseComponent(Component, Description)

reuseComponent(Component, Description) → useComponent(Component, Description, "as-is")

defineSubclass(NewComponent, Component, Description)
        → useComponent(Component, Description, "refine")
```

En este caso *Component* es una variable, que puede tomar el valor de cualquier clase del *framework*. El objetivo *selection* es utilizado para pedir al usuario que escoja el camino a seguir y de acuerdo a esta elección tomar el objetivo *reuseComponent* o *defineSubclass*. Para ello es necesario que el planificador se detenga, espere la respuesta del usuario y en función de ella considere las próximas acciones a utilizar para resolver los objetivos pendientes.

Para implementar este comportamiento debe utilizarse un nuevo tipo de componentes, llamados **operadores**, como parte de las precondiciones de una acción, además de los objetivos definidos por el usuario y los primitivos.

2.2.2. Operadores

Los operadores representan instrucciones específicas para el planificador, y permiten representar las acciones necesarias para construir un plan de instanciación. Al igual que los objetivos primitivos, los operadores forman un conjunto fijo que no puede ser extendido por el autor de la documentación. Hasta el momento se han definido cinco operadores, aunque más podrían ser definidos en caso de ser necesarios. Dentro del conjunto de operadores, dos de ellos están relacionados con la creación de tareas de instanciación: su función es indicarle al planificador cuándo debe crear las tareas que deberían ser ejecutadas por el usuario.

términos entendibles por el planificador. Un ejemplo del tipo de información que debe proveer el diseñador es la siguiente definición[4]:

```
select("Class Name", Class), select("Attribute Name", Attribute), option(["Tool", "Menu", "Handler"], Ret),
    goal(editAttribute(Ret, Class, Attribute) → functionality("Interactive Edition of Attribute Values")
```

A partir de esta definición, el sistema puede mostrar "*Interactive Edition of Attribute Values*" como una de las opciones de funcionalidad. Si el usuario la selecciona, el sistema le solicita el nombre de la clase, del atributo y la forma de editarlo. A partir de esa información genera el objetivo para el planificador. En este caso el objetivo generado sería editAttribute ("Tool", "PertTask", "Duration"). La interpretación de esto es que se debe implementar la edición del atributo *Duration* de la figura *PertTask* utilizando una herramienta. Finalmente, la funcionalidad descrita arriba da lugar a las siguientes precondiciones de la acción final en la agenda:

```
useFigure( "PertTask", "" ), relateAttributes ( "PertTask" ), relateAttributes( "PertTask", "PertTask" ),
    editAttribute( "Tool", "PertTask", "Duration" ), editAttribute( "Menu", "PertTask", "EndDate" ),
    makeRelationship ( "PertTask", "PertTask" ).
```

El trabajo del planificador es encontrar un plan que permita satisfacer estos objetivos, es decir, implementar una aplicación con esa funcionalidad.

2.4. Representación del Plan de Instanciación

La representación del plan generado por el planificador consta de dos componentes principales. Por una parte está la descripción del plan tal cual es visto por el usuario. Esta parte del plan es una lista de tareas de instanciación que deben ser ejecutadas (tareas pendientes). Por otra parte, está toda la información "de control" producida por el planificador, y que será utilizada en caso de una replanificación o de continuar con un plan incompleto. Así, el plan producido puede representarse a través de una tupla:

<A,O,L,B, Agenda, Tareas, Pendientes>

donde:

- *A* es el conjunto de acciones que componen el plan actual;
- *O* es el conjunto de restricciones de orden;
- *L* es el conjunto de enlaces causales;
- *B* es el conjunto de vínculos entre variables;
- *Agenda* es la lista de objetivos por satisfacer;
- *Tareas* es la lista de tareas que serán creadas como resultado del plan
- *Pendientes* es la lista de objetivos para los cuales no fue posible elaborar un plan con la información disponible.

2.5. El Algoritmo *PIT*

Utilizando las representaciones descritas, el algoritmo *PIT* fue diseñado a partir de *UCPOP*, teniendo en cuenta las requisitos particulares de la instanciación de *frameworks*. El algoritmo es invocado con dos argumentos: un plan de instanciación inicial basado en la funcionalidad descrita por el usuario y el conjunto de acciones asociado a la documentación del *framework*.

[4] Esta definición no debe ser confundida con una acción para el planificador. Sólo es un ejemplo del tipo de información que podría proveer el diseñador para facilitar la especificación de la funcionalidad requerida por el usuario.

En líneas generales, el algoritmo intentará en cada paso construir un plan para un conjunto determinado de los objetivos iniciales, utilizando la función auxiliar *PIT_Aux*. En primer lugar, ese conjunto corresponderá al conjunto total de objetivos y si no existe un plan, probará con diferentes combinaciones de objetivos. La selección del objetivo a eliminar en cada iteración se realiza a través de una función no determinista, por lo que su exacta definición no es importante para la estructura del algoritmo.

La Figura 5.10 presenta el algoritmo principal, en tanto que las Figura 5.11, 5.12 y 5.13 presentan las funciones auxiliares.

El paso 1 del algoritmo verifica si la agenda de objetivos está vacía, en cuyo caso retorna el plan actual. Si es así, a partir de la información de *tareas* se generarán las tareas que deberían ser llevadas a cabo por el usuario. El paso 2 es la invocación a la función auxiliar, la cual realiza el trabajo principal de planificación. Si esta invocación retorna exitosamente, entonces un plan fue encontrado para los objetivos propuestos, y el algoritmo retorna el plan generado. Si la invocación falla, intenta generar un plan sin alguno de los objetivos originales. Para esto en el paso 3 elimina de forma no determinista algún objetivo y en el paso 4 se invoca recursivamente. Si esta invocación recursiva falla, intenta eliminando un objetivo diferente. Este algoritmo intentará eventualmente con todas las combinaciones de objetivos iniciales hasta encontrar un plan factible o devolver un plan vacío. Un plan vacío significa que el planificador no pudo construir un plan para ninguno de los objetivos iniciales, ya sean combinadas o de forma aislada.

Algoritmo: PIT $(\langle A,O,L,B, agenda, tareas, pendientes\rangle, \Delta)$

1. **Terminación:** Si *agenda* es vacía, retornar $(\langle A,O,L,B, agenda, tareas, pendientes\rangle)$
2. **Invocación función auxiliar:** *retorno* = $PIT\_Aux\ (\langle A,O,L,B, agenda, tareas\rangle, \Delta)$. Si tiene éxito, retornar *retorno* más la lista de objetivos pendientes. En caso contrario, ir al paso 3
3. **Cancelación de Objetivo:** Eligir no determinísticamente un objetivo $O_{pend} \in agenda$. Hacer $pendientes' = pendientes \cup \{O_{pend}\}$, $agenda' = agenda - \{O_{pend}\}$.
4. **Invocación recursiva:** *retorno* = $PIT\ (\langle A,O,L,B, agenda', tareas, pendientes'\rangle)$. Si la invocación tiene éxito, retornar *retorno*. En caso contrario, ir al paso 3 y cambiar el objetivo a cancelar.

Figura 5.10 El algoritmo *PIT*

Para garantizar que el algoritmo se detiene y que intenta todas las alternativas antes de fallar, es necesario que la función que elige el subconjunto de objetivos dejados de lado sea cuidadosamente diseñada. Debe asegurar que, en cada nivel, cada objetivo sea considerado una y sólo una vez. Este enfoque de "fuerza bruta" puede ser mejorado en varios aspectos para obtener una mejor estrategia de construcción del plan. Por ejemplo, se podría pensar que ciertos objetivos son más difíciles de implementar (es decir, armar un plan para implementarlos), por lo que deberían ser considerados en primer término para su eliminación. Esto ayudaría a reducir el tiempo utilizado para la ejecución del algoritmo, el cual es claramente de complejidad exponencial.

De la misma forma, la estructura actual no garantiza que el plan construido corresponda a la máxima cantidad de objetivos posibles. Estrategias alternativas pueden buscar maximizar el número de objetivos cubiertos por el plan.

2.5.1. La función *PIT_Aux*

La actividad principal de planificación es realizada en la función auxiliar *PIT_Aux*. Esta función es la encargada de buscar el camino de acciones que permitan satisfacer, en lo posible, todos los objetivos de la agenda. Cada vez que llega a un punto muerto en la planificación, es

decir, que no encuentra una alternativa para satisfacer los objetivos, el algoritmo retrocede hasta donde se tomó la última decisión, e intenta considerando una alternativa diferente (*backtracking*). El punto principal de toma de decisión es donde se elige una acción para satisfacer los objetivos (paso 3 del algoritmo). También se toman decisiones que en caso de fallo deben ser reconsideradas cuando la precondición de una acción presenta alternativas disyuntivas (paso 2.b). Finalmente, si ninguna combinación de opciones consigue producir un plan para los objetivos propuestos, se retorna al algoritmo principal para intentar con una nueva combinación de objetivos.

Función PIT_Aux: ((A,O,L,B), agenda, tareas, Δ)

1. **Terminación:** si agenda es vacía, retornar *{A, O, L, B, agenda, tareas}.*
2. **Reducción de Objetivo:** borrar un objetivo *⟨Q, A_c⟩* de *agenda*
   a) Si *Q* es una conjunción de Q_i entonces colocar cada *<Q_i,A_c>* en la *agenda*; volver al paso 2.
   b) Si *Q* es una disyunción de Q_i entonces elegir el próximo Q_k de la disyunción a considerar y colocar *<Q_k,A_c>* en la *agenda*; volver al paso 2.
   c) Si *Q* es una *waitingTask(TaskClass, Description, Return)*, crear la tarea y colocarla en la lista de tareas para ejecutar. Hacer *Resultado = NextEvent.* Luego de retornar, *B'= B ∪ {(Return,Resultado) }.* El resto de las variables permanece sin cambios. Ir al paso 6.
   d) Si *Q* es una *pendingTask*, hacer *tareas' = tareas ∪ {Q}.* El resto de las variables permanece sin cambios. Ir al paso 6.
   e) Si Q es un *operador*, invocar *HandleOperator*(Q, A_c, ⟨A, O, L, B⟩, agenda, ⟨A', O', L', B'⟩, agenda'). Si esta función no falla, ir al paso 6, con tareas' = tareas. En otro caso, fallar.
3. **Selección de una acción:** elegir una acción A_p, con *R* un literal del efecto conjunto de A_p (o consecuente conjunto si el efecto es condicional). Primero se intenta con las acciones ya existentes en A y luego se intenta instanciar una nueva acción de *Δ*. En cualquiera de los dos casos, $A_p < A_c$ debe ser consistente con *O* y el literal *R* debe unificar con *Q* dado *B.* Si tal acción no existe, entonces fallar. De lo contrario, hacer:
   a) $L' = L \cup \{A_p \rightarrow^{Q} A_c\}$
   b) $B' = B \cup \{(u,v) | (u,v) \in MGU(Q,R,B) \wedge u,v$ son variables no universalmente cuantificadas del efecto*}*
   c) $O' = O \cup \{A_p < A_c\}$.
4. **Habilitar nuevas acciones y efectos:** sea *A'= A* y *agenda' = agenda* . Si $A_p \notin A$ entonces agregar A_p a A', agregar *<precondiciones lógicas(A_p)* ∥ *MGU(Q,R,B),A_p>* a *agenda'*, agregar *{A_0<A_p<A_∞}* a *O* y agregar las restricciones de no codesignación(A_p) a *B'*. Si el efecto es condicional y no ha sido usado para establecer un enlace en *L*, entonces agregar su antecedente a *agenda'* después de sustituir con *MGU(Q,R,B).*
5. **Protección de enlaces causales:** para cada enlace $l = A_i \rightarrow^{P} A_j$ en *L* y por cada acción A_t en A' que pueda amenazar el enlace causal *l* seleccionar una de las siguientes opciones:
   a) Retracción: agregar $A_t < A_i$ a *O'*
   b) Promoción: agregar $A_j < A_t$ a *O'*.
   c) Confrontación: si el efecto de A_t que amenaza el enlace es condicional con antecedente *S* y consecuente *R*, entonces agregar *<not(S)∥MGU(P, not(R)) , A_t >* a *agenda'*, a menos que el objetivo sea la negación de una precondición de A_t, en cuyo caso la confrontación no es posible.
   d) Si ninguna opción es aplicable retornar fallo
6. **Invocación recursiva:** Si *B* es inconsistente entonces fallar; si no invocar *PIT_Aux(<A',O',L',B'>,agenda', tareas',Δ)*

Figura 5.11 Función auxiliar *PIT_Aux*

La lista de argumentos enviados a la función *PIT_Aux* son prácticamente los mismos enviados al algoritmo *PIT*, explicados en §V.2.4. La función comienza verificando si la agenda está vacía, en cuyo caso un plan fue encontrado y es retornado (paso 1). El paso 2 intenta eliminar el próximo objetivo de la agenda y es el paso que presenta más cambios con respecto al algoritmo *UCPOP*. El orden de los objetivos es significativo, por lo cual siempre se toma el primer objetivo de la lista:

- En el punto 2.a) determina si la precondición de la acción es compuesta (conjunción), en cuyo caso la descompone en condiciones simples y las coloca en la agenda
- En 2.b) verifica si la precondición posee alternativas. En este caso, escoge las alternativas en el orden en que las encuentra e intenta completar el plan. Si más adelante se llega a un punto muerto, volverá a este punto y escogerá otra alternativa (*backtracking*).
- En el punto 2.c) son tratadas las *waitingTasks*. Cuando la próxima acción en la agenda comprende una tarea de este tipo, la tarea es creada y presentada al usuario para su ejecución. Luego, la llamada a la función *NextEvent* suspende el proceso de planificación hasta que el usuario finaliza la tarea. Una vez finalizada, el resultado de la ejecución de la tarea es retornado a *PIT_Aux*, el cual actualiza el plan para que el resultado de la tarea sea utilizado a continuación. En este caso actualizar el plan significa actualizar la asociación entre variables y valores.

Función NextEvent ()

1. **Esperar el próximo evento:** detenerse hasta que el usuario produzca un evento, al finalizar una tarea. *Sea evento = ⟨EventName, TaskDesc, Result⟩.*
2. **Reiniciar planificación:** retornar *Result*.

Figura 5.12 Función *NextEvent*

- El punto 2.d) se encarga del tratamiento de las *pendingTasks*. Estas tareas son simplemente agregadas a la lista de tareas que serán creadas al finalizar la planificación.
- El punto 2.e) trata el caso de los operadores distintos de *waitingTask* y *pendingTask*. Para esto invoca a la función *HandleOperator*, que decidirá si se puede seguir con el plan actual o debe hacerse *backtracking*. En algunos casos también modificará la agenda y/o la lista de vínculos causales.

Function HandleOperator (Q, A_c, ⟨A, O, L, B⟩, agenda, ⟨A', O', L', B'⟩, agenda')

case(Q)

if (condition) Si (Ap, condition)$\notin$ *A*, fallar. En otro caso, L' = L + $\{A_p \rightarrow^{condition} A_c\}$, y el resto de las variables permance igual.

not (action) Si (Ap, condition) $\in$ *A*, fallar. En otro caso, L' = L + $\{A_p \rightarrow^{not(condition)} A_c\}$, y el resto de las variables permance igual.

forEach(Var, List, Objetive) *agenda' = agenda*. Por cada elemento *X* de *List*, colocar en *agenda'* un nuevo objetivo que sea igual a *Objetive* con cada ocurrencia de *Var* reemplazada por *X*

Figura 5.13 Función *HandleOperator*

Aquí debe notarse que, a diferencia de *UCPOP*, *PIT_Aux* no verifica que existan condiciones contradictorias asociadas a la misma acción. Esto es así porque dado el tipo de acciones que interpretará *PIT*, este tipo de contradicciones sólo pueden provenir de la utilización de confrontación para evitar la amenaza de vínculos causales. De esta forma, basta con

establecer como condición de la protección de vínculos causales que la utilización de confrontación no provoque estas contradicciones.

El paso 3 intenta encontrar una acción que permita satisfacer la condición que se está considerando en ese momento (*Q*). Así, primero intenta verificar si alguna de las acciones que ya ha instanciado le permite satisfacer Q (es decir, intenta encontrar una acción previa que ya haya hecho el trabajo) y luego busca una nueva acción que lo haga.

En el caso de buscar entre las acciones de instanciación, esta búsqueda es ordenada y no es una elección no determinista como era el caso del algoritmo *UCPOP*. Esto es así porque el orden de definición de las acciones es significativo. Así, por ejemplo, las acciones representando información específicas de un *framework* son colocadas en primer lugar, mientras que las acciones generales son colocadas a continuación. Entonces, para una situación particular, si existe una acción específica que pueda ser aplicada, esta acción es la escogida. Si no existe una acción específica, se busca entre las acciones generales.

Por ejemplo, existe una acción primitiva que describe la creación de nuevas clases:

```
pendingTask ( "DefineClass", [NewClass, SuperClass], Description, "Required" ),
        → defineClass( NewClass, SuperClass, Description )
```

En determinado *framework* puede ser deseable refinar dicha acción de instanciación. Por ejemplo, es posible especificar que en *HotDraw* todas las subclases de Figure deben ser documentadas. Entonces es posible definir una acción de instanciación para *HotDraw* de la siguiente forma:

```
pendingTask ( "DefineClass", [NewClass, SuperClass], Description, "Required" ),
        pendingTask( "DocumentClass", [Class, Description], "Optional" )
        → defineClass( NewClass, "Figure", Description )
```

Siempre que la nueva clase creada tenga como superclase Figure, se utilizará esta acción. En todos los otros casos, se utilizará la acción primitiva.

En cualquier caso, es necesario que la acción elegida permita satisfacer la condición (es decir, que el efecto de la acción A contenga un literal que unifique con la condición *Q*) y debe poder ser ordenada consistentemente con las acciones ya existentes. Si esto se cumple, las variables del plan son actualizadas consecuentemente. El paso 4, así mismo, actualiza las listas de acciones y objetivos, mientras que el paso 5 actualiza la protección de enlaces causales y en el 6 las asociaciones entre variables. Si en cualquiera de estos puntos no es posible encontrar alternativas consistentes con el plan actual, se vuelve atrás para considerar nuevos caminos. En otro caso, se realiza una invocación recursiva para intentar resolver los objetivos todavía pendientes.

En la próxima sección se describe un ejemplo de generación de un plan de instanciación utilizando el algoritmo *PIT*. En el capítulo VIII se describen algunos detalles de implementación del algoritmo.

2.6. Ejemplo de Planificación con el Algoritmo *PIT*

En esta sección se dará un corto ejemplo de utilización del algoritmo *PIT* para la generación de planes de instanciación. Para ello primero se dará una especificación parcial de las acciones de instanciación asociadas al *framework HotDraw*, y luego se mostrará la forma en que el algoritmo genera los planes de instanciación a partir de esas acciones.

2.6.1. Acciones de instanciación para el *framework HotDraw*

Las acciones a utilizar en este ejemplo son las presentadas en la Figura 5.14. Como se ha dicho, la parte izquierda de cada acción describe las condiciones necesarias para que puedan ser

aplicada la parte derecha (los efectos). Por ello, para facilitar su comprensión muchas veces es más conveniente comenzar su lectura por la parte derecha.

Estas acciones son un subconjunto de las utilizadas para documentar *HotDraw* y han sido simplificadas ligeramente para facilitar su explicación. Por ejemplo, el símbolo '_' denota variables cuyo nombre no se coloca por no ser de interés para el ejemplo. Así mismo, la línea de puntos representa subobjetivos que han sido omitidos. La acciones de la Figura 5.14 han sido definidas específicamente para documentar *HotDraw*, mientras que las de la Figura 5.15 son acciones primitivas. En los anexos A y B están descritas las acciones de instanciación primitivas y las de *HotDraw*, respectivamente.

La acción 1) especifica simplemente que para definir una lista de atributos, debe definirse cada uno de ellos individualmente. La acción 2) describe precisamente cómo definir un atributo: se deben resolver los subobjetivos de definir una nueva variable para la clase e inicializarla. Las acciones 3) a 5) definen la forma de implementar la edición interactiva de un atributo de una figura. Cada una trata con una de las formas de implementar esa funcionalidad provista por *HotDraw*: la acción 3) utiliza un *Tool*, la 4) un *Menú* y la 5) un *Handler*. En los tres casos, los dos primeros subobjetivos verifican que no se esté utilizando métodos inconsistentes para la edición de distintos atributos (esto es para obtener uniformidad en la interfaz de usuario).

Analizando con más detalle la acción 3), el primer subobjetivo verifica que no exista hasta el momento en el plan un objetivo para editar un atributo utilizando un *Handler* y el segundo hace la misma verificación con los menús. Si no se dan estos casos, entonces continúa con la implementación de la funcionalidad, es decir, establece que la clase del atributo debe ser una subclase de *CompositeFigure* y que el atributo debe ser inicializado como un atributo a ser editado.

La acción 6) define el mecanismo general de inicialización de un atributo: primero se le pregunta al usuario si el atributo será editado y luego se sigue el camino correspondiente. La acción 7) define la inicialización de un atributo que será editado, mientras que la acción 8) hace lo propio con un atributo que no será editado.

1) forEach(X, List, defineAttribute(Class, X, "")) → defineAttributeList(Class, List)

2) defineVariable (Figure, Name, Description), initializeAttribute(Figure, Name, Description)
→ defineAttribute(Figure, Name, Description)

3) not(editAttribute("Handler", _, _), "No Consistent"),
no(editAttribute("Menu", _, _), "No Consistent"),
defineSubclass(ClassName, "CompositeFigure", "An attribute has to be edited"),
specificInitialization ("edit", ClassName, AttributeName, Description),
useTool("editAttribute")
→ editAttribute ("Tool", ClassName, AttributeName)

4) not(editAttribute("Handler", _, _),"No Consistent"), no(editAttribute("Tool", _, _),
"No Consistent"),
.....
→ editAttribute("Menu", ClassName, AttributeName)

5) not(editAttribute("Menu", _, _), "No Consistent"), no(editAttribute("Tool", _, _),
"No Consistent"),
.....
→ editAttribute("Handler", ClassName, AttributeName)

6) selection(["edit", "noEdit"], Name, Ret), specificInitialization(Ret, Figure, Name, Description)
→ initializeAttribute(Figure, Name, Description)

7) selection(["text", "number"], Name, Ret), initEditableAttribute(Ret, Figure, Name, Description)
→ specificInitialization("edit", Figure, Name, Description)

8) updateMethod(Figure, "initialize:", string(Name, ":= *HotDraw*Variable with:0 owner:self."), Description), ...
→ specificInitialization("noEdit", Name, Description)

9) updateMethod(Figure, "initialize:", string(Name, ":= TextFigure string:'' at: aPoint."), Description)
→ initEditableAttribute("text", Figure, Name, Description)

10) updateMethod(Figure, "initialize:", string(Name, "NumberFigure string:'0' at: aPoint."), Description)
→ initEditableAttribute("number", Figure, Name, Description)

11) selectMethod("*HotDraw*Constraint", "class", "instance creation", Method),
assignArguments(Method, Class, Attributes, LocalVars, MessageString)
defineAttributeList (Class, Attributes),
updateMethod(Class, "initializeAt:", string("*HotDraw*Constraint", MessageString), LocalVars, Description)
→ relateAttributes(Class)

12) defineSubclass(ClassName, "Figure", Description) → useFigure(ClassName, Description)

Figura 5.14 Descripción parcial de las acciones de instanciación para el *framework HotDraw*

Las acciones 9) y 10) definen la forma de definir un atributo que será editado, la acción 9) para el caso de un atributo textual y la 10) para uno numérico. En ambos casos se debe modificar el método de inicialización de la figura correspondiente, creando el objeto apropiado.

La acción 11) es una de las acciones que tratan sobre la definición de relaciones entre atributos; en este caso es cuando los atributos pertenecen a la misma clase. Este tipo de relaciones en *HotDraw* se implementan utilizando restricciones. Por este motivo, primero se debe elegir un método de la clase *HotDrawConstraint* para definir el tipo de relación (restricción) a crear, luego definir qué atributos serán relacionados, luego asegurarse de que todos estos atributos estén definidos e inicializados y finalmente crear la restricción, modificando para ello el método de inicialización de la figura correspondiente.

La última acción específica para *HotDraw*, la 12) describe la implementación de la utilización de figuras: por cada tipo de figura que se desee utilizar, debe definirse una subclase de *Figure*.

2.6.2. Acciones primitivas

La acciones 13) a 17) son acciones primitivas, generales para todos los *frameworks*. Las acciones primitivas en general describen la interacción del usuario con el entorno de programación, por lo que generalmente incluyen una *pendingTask* o *waitingTask*.

La acción 13) se encarga de la creación de tareas para la definición de clases nuevas y la 14) de tareas para incorporar variables a las clases.

La acción 15) implementa la funcionalidad necesaria para solicitarle al usuario, durante el proceso de planificación, que escoja una entre varias opciones. La 16) es similar a la anterior, permitiendo elegir un método entre los métodos de una categoría dada de una clase.

Finalmente, la acción 17) crea la tarea que se encarga de modificar el código de métodos ya existentes.

13) pendingTask("DefineClass", [NewClass, SuperClass], Description)
→ defineSubclass(NewClass, SuperClass, Description)

14) pendingTask("DefineAttribute", [Class,Name], Description)
→ defineVariable(Class, Name, Description)

15) waitingTask ("AskSelection", [Options, Description], Return)
→ selection(Options, Description, Return)

16) waitingTask("SelectMethod", [Class, MethodKind, Category], Return)
→ selectMethod(Class, MethodKind, Category, Return)

```
17) pendingTask( "UpdateMethod", [Class, Method, String, LocalVars], Description )
        → updateMethod( Class, Method, String, LocalVars, Description )
```

Figura 5.15 Algunas acciones primitivas utilizadas en la instanciación de *HotDraw*

2.6.3. Objetivos funcionales

Supóngase que dadas las acciones de las Figuras 5.14 y 5.15, se pretende implementar la funcionalidad descrita por la siguiente lista de objetivos (es el mismo ejemplo utilizado para mostrar la representación de objetivos en §V.2.3):

```
useFigure( "PertTask", "" ), relateAttributes ( "PertTask" ), relateAttributes( "PertTask", "PertTask" ),
        editAttribute( "Tool", "PertTask", "Duration" ), editAttribute( "Menu", "PertTask", "EndDate" ),
        makeRelationship ( "PertTask", "PertTask" ).
```

La funcionalidad representada por estos objetivos es que se quiere:

- representar un tipo de objeto llamado *PertTask*, los cuales deben poder ser editados individualmente
- establecer relaciones entre atributos de una misma *PertTask*
- establecer relaciones entre atributos de distintas *PertTasks*
- editar a través de la interfaz de usuario el atributo *Duration* de las *PertTask*, utilizando un *Tool*
- editar el atributo *EndDate*, en este caso utilizando un menú
- establecer relaciones entre las *PertTask*

2.6.4. Generación del plan de instanciación

En esta subsección se explica la forma en que el algoritmo *PIT* encuentra un plan que permita satisfacer los requisitos funcionales especificados. *PIT* es invocado entonces con los argumentos ***A,O,L,B, agenda, tareas, pendientes*** y *Δ*. Como se explicó en §V.2.4, *A* representa el conjunto de acciones que componen el plan actual, *O* es el conjunto de restricciones de orden, *L* es el conjunto de enlaces causales, *B* es el conjunto de vínculos entre variables, *Agenda* es la lista de objetivos por satisfacer, *Tareas* es la lista de tareas que serán creadas como resultado del plan, *Pendientes* es la lista de objetivos para los cuales no fue posible elaborar un plan con la información disponible y Δ es el conjunto de acciones de instanciación mostradas en la Figura 5.15. En esta sección, la notación $A_{m,n}$ representa la aplicación n-ésima de la acción m-ésima.

Inicialmente, *A, O, L, B, tareas* y *pendientes* son conjuntos vacíos. *Agenda*, por su parte, debe contener la lista de objetivos descrita más arriba, puestos en el formato adecuado. Es decir, cada uno de los objetivos funcionales será considerado como precondición de una acción final, A_∞. En consecuencia, será agenda = {⟨useFigure ("PertTask",""), A_∞⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩}.

Con estos argumentos, el algoritmo *PIT* verifica si la agenda está vacía. Como no lo está, invoca a la función auxiliar *PIT_Aux* (Figura 5.11), con los mismos argumentos, excepto el último.

Esta función en el primer paso verifica si la agenda está vacía; como no es así, continúa. En el paso 2) intenta eliminar un objetivo. El primer objetivo de agenda es retirado; en este caso Q = *useFigure ("PertTask","")*. Ninguna de las condiciones a) – f) se cumple, luego pasa el paso 3). El conjunto de acciones ya intanciadas es vacío, por lo que debe elegir una nueva. Elige

la acción 12) que cumple las condiciones exigidas, en especial que un literal de su efecto unifica con Q. Entonces hace

$L' = \{A_{12,1} \rightarrow^{useFigure} A_\infty\}$;

B' = {(ClassName_1[5], "PertTask"), (Description_1, "")};

$O' = \{A_{12,1} < A_\infty\}$.

El paso 4 hace

A' = {($A_{12,1}$, useFigure("PertTask",""))};
agenda' = {⟨defineSubclass ("PertTask", "Figure", ""), $A_{12,1}$⟩,
⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute ("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"),A_∞⟩}
$O' = \{A_0 < A_{12,1} < A_\infty\}$ (subsume a la que estaba antes en O).

El contenido de A' representa el plan actual y en este caso significa que se ha aplicado la acción A_{12}, que tiene como efecto *useFigure("PertTask","")*.

En el paso 5) no es necesario hacer nada, luego va al 6) y se invoca recursivamente con los nuevos valores.

En esta nueva invocación, vuelve a verificar si la agenda está vacía y como no es el caso, pasa a 2). Toma ⟨Q, A_c⟩ = ⟨defineSubclass("PertTask", "Figure", ""), $A_{12,1}$⟩. De nuevo, ninguna de las condiciones del punto 2) se cumple, por lo que el algoritmo sigue por el punto 3). Una acción que cumple las condiciones es la 13). Entonces

$L' = \{ A_{12,1} \rightarrow^{useFigure} A_\infty; A_{13,1} \rightarrow^{defineSubclass} A_{12,1}\}$
B' = {(ClassName_1, "PertTask"), (Description_1, ""), (NewClass_1, ClassName_1), (SuperClass_1, "Figure"), (Description_2, Description_1)}
$O' = \{A_0 < A_{12,1} < A_\infty; A_{13,1} < A_{12,1}\}$.

En el siguiente paso se hace

A' ={($A_{12,1}$, useFigure("PertTask",""), ($A_{13,1}$, defineSubclass("PertTask", "Figure",""))};
agenda' = {pendingTask("DefineClass", [NewClass_1, SuperClass_1, Description_2]), $A_{13,1}$⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩}
$O' = \{A_0 < A_{12,1} < A_\infty; A_{13,1} < A_{12,1}; A_0 < A_{13,1}\}$.

Finalmente, en el paso 5) no se hace nada y en el 6) se efectúa la invocación recursiva nuevamente.

Aquí el objetivo a considerar es ⟨Q, A_c⟩ = ⟨ pendingTask ("DefineClass", [NewClass_1, SuperClass_1], Description_2), $A_{13,1}$⟩. En este caso, se cumple la condición del punto 2.d). Por esto, se hace

tareas' = { pendingTask ("DefineClass", ["PertTask", "Figure"], "")},

se va a 6) y se invoca recursivamente.

El próximo objetivo a considerar es ⟨Q, A_c⟩ = ⟨ relateAttributes ("PertTask"), A_∞⟩.

[5] El número al final es necesario colocarlo para distinguir las distintas ocurrencias (en distintas acciones) de variables con el mismo nombre.

Después va a 3), donde $A_p = A_{11,1}$, haciendo

$L' = \{ A_{12,1} \rightarrow^{useFigure} A_\infty; A_{13,1} \rightarrow^{defineSubclass} A_{12,1}; A_{11,1} \rightarrow^{relateAttributes} A_\infty \}$

B' = {(ClassName_1, "PertTask"), (Description_1, ""), (NewClass_1, ClassName_1), (SuperClass_1, "Figure"), (Description_2, Description_1), (Class, "PertTask")}

$O' = \{A_0 < A_{12,1} < A_\infty; A_{13,1} < A_{12,1}; A_0 < A_{13,1} ; A_{11,1} < A_\infty\}$.

Ya en el paso 4) se hace

$A' = \{A_{12,1}, A_{13,1}, A_{11,1}\}$[6]

$O' = \{A_0 < A_{12,1} < A_\infty; A_{13,1} < A_{12,1}; A_0 < A_{13,1} ; A_0 < A_{11,1} < A_\infty\}$

(otra vez una restricción es subsumida) y

agenda' = { ⟨ selectMethod ("*HotDraw*Constraint", "class", "instance creation", Method), $A_{11,1}$⟩ ,
⟨assignArguments (Method, Class, ArgList), $A_{11,1}$⟩,
⟨defineAttributeList (Class, ArgList), $A_{11,1}$⟩,
⟨updateMethod (Class, "initializeAt:", string ("*HotDraw*Constraint", *MethodWithArgs*), Description), $A_{11,1}$⟩,
⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩,
⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩,
⟨editAttribute ("Menu", "PertTask", "EndDate"), A_∞⟩,
⟨makeRelationship ("PertTask", "PertTask"), A_∞ ⟩ }

El algoritmo continúa de la misma forma y en la próxima interacción encontrará ⟨Q, A_c⟩ = ⟨ selectMethod ("*HotDraw*Constraint", "class", "instance creation", Method), $A_{11,1}$⟩. Para resolverlo deberá utilizar la acción 16), por lo que en la siguiente invocación recursiva se hará ⟨Q, A_c⟩ = ⟨waitingTask("SelectMethod",[Class, MethodKind, Category], Return), $A_{16,1}$⟩. En este caso, se deberá aplicar los descrito en el punto 2.c de *PIT_Aux*. Primero es creada una tarea de espera y se la coloca para que el usuario la ejecute; para esto se utiliza un administrador de tareas, según se explica en el próximo capítulo. Se invoca a la función *NextEvent* y se espera que el usuario ejecute la tarea recién creada. Al finalizar, se hace

$A' = A \cup \{$wasTask("SelectMethod", "plus:and:equal")$\}$,

donde el nombre del método es un ejemplo de lo que podría elegir el usuario. En la lista de variables B se agrega el par (Method_1, "plus:and:equal"), lo que permitirá que el resto de los objetivos conozcan el nombre de método elegido.

Otro punto interesante de destacar se da cuando el algoritmo tiene que considerar el objetivo ⟨*editAttribute ("Menu", "PertTask", "EndDate"), A_∞*⟩. En este momento, ya ha considerado el objetivo ⟨*editAttribute ("Tool", "PertTask", "Duration"), A_∞*⟩ y por lo tanto forma parte del plan. Al tomar la acción 4) que es la que describe cómo tratar de resolver el objetivo actual, detecta que no puede aplicar la acción porque el operador *not* falla por la presencia de otro objetivo relacionado con la edición de atributos, que utiliza un *tool*. Al fallar esta acción, intenta considerar otra, pero no encuentra una acción cuyo efecto satisfaga la precondición del objetivo actual. Entonces tiene que fallar completamente *PIT_Aux* y volver al algoritmo principal *PIT*, donde se intentará descartar algún objetivo. El resultado final será que el planificador será finalmente capaz de generar un plan para todos los objetivos, excepto uno (ya sea el de editar el atributo usando *menú* o el de hacerlo utilizando un *tool*, dependiendo de la elección no determinística del paso 3 del algoritmo *PIT*). Si el usuario reconsidera la lista inicial de objetivos, eligiendo para ambos el mismo método de edición, o dejando la opción abierta (es decir, que el planificador sugiera un método), entonces será posible generar un plan completo para todos los requisitos funcionales.

[6] A partir de aquí se omitirá la representación explícita de los efectos causados por las acciones que forman parte del plan de instanciación.

Finalmente es importante destacar que aplicando las acciones de la Figura 5.15 para los objetivos descritos, se generarán dos tareas de creación de la clase *PertTask*. Esto ocurre porque por efecto del objetivo *useFigure ("PertTask","")*, y utilizando la acción 12), se genera una tarea para la creación de una subclase de *Figure*, llamada *PertTask*. Por otro lado, a raíz del objetivo *editAttribute ("Tool", "PertTask", "Duration")*, por la acción 3) se generará otra tarea para la creación de la clase *PertTask*, en este caso como subclase de *CompositeFigure*.

Este tipo de situación se soluciona uniendo las dos tareas y dejando la más restrictiva, la creación de una subclase de *CompositeFigure* en este ejemplo. Este mecanismo, sin embargo, no es implementado a través del planificador, sino a través del administrador de tareas explicado en el próximo capítulo.

2.7. Especificación de las Acciones de Instanciación

De acuerdo a lo visto, el trabajo del autor de la documentación es, además de utilizar las técnicas normales para documentar el *framework*, proveer las acciones de instanciación que permitirán generar planes de instanciación según la funcionalidad requerida y guiar la actividad del usuario durante el proceso de instanciación. Sin embargo, en el capítulo anterior se enfatizó la importancia de no complicar excesivamente la labor del documentador. Por este motivo, se han provistos distintas ayudas para la creación de las acciones de instanciación:

- Objetivos primitivos y operadores: las acciones de instanciación para los objetivos funcionales específicos para cada *framework* pueden ser descritos en términos de objetivos primitivos y operadores. De esta forma, el diseñador no tiene que describir cada detalle del proceso de instanciación de su *framework*, sino aquellos detalles que no son comunes. Por ejemplo, no es necesario que describa las acciones necesarias para reutilizar un componente existente, para definir un nuevo atributo para una clase o cómo definir los argumentos utilizados en una invocación determinada de un método.
- Notación gráfica: se ha diseñado una representación gráfica para las reglas, tanto funcionales como de instanciación. Esta representación no tiene la suficiente potencia de expresión para representar toda la información necesaria para el algoritmo de planificación, pero facilita la definición de una parte importante de las reglas, sobre todo respecto de las reglas de consistencia. Debido a que esta notación es fuertemente dependiente de la representación de las tareas de instanciación, será explicada en el próximo capítulo.
- Herramientas de asistencia a la documentación: Como la notación gráfica no es suficiente para definir las reglas de instanciación, el diseñador debe completar con especificaciones textuales esa definición. En los próximos capítulos se muestra cómo pueden utilizarse herramientas para asistir en la creación de esta especificación textual.

VI Técnicas de Apoyo a SmartBooks

En los capítulos anteriores se ha descrito cómo el proceso de instanciación de *frameworks* orientados a objetos puede ser asistido a través de la utilización de técnicas de planificación. La utilización de estas técnicas facilita en gran medida la utilización de *frameworks* para producir aplicaciones, especialmente en el caso de usuarios novatos. Sin embargo, para que estas propuestas puedan ser aplicadas de forma práctica, es necesario disponer de mecanismos que asistan en su materialización. En primer lugar, es necesario facilitar el proceso de documentación, utilizando notaciones gráficas que simplifiquen la representación de las reglas de instanciación. También es necesario utilizar mecanismos para administrar la consistencia prescrita por las reglas de consistencia. Finalmente, se necesita algún mecanismo para administrar y orientar la ejecución de las tareas de instanciación creadas a partir del plan.

En este capítulo se describen los mecanismos que se han diseñado para apoyar la aplicación de *SmartBooks* para la documentación e instanciación de *frameworks*. En primer lugar se explican las relaciones existentes entre las distintas porciones de la documentación. Luego se presenta el modelo de tareas utilizado para representar el plan de instanciación presentado al usuario, así como el administrador de tareas diseñado para administrar la ejecución de esas tareas. Posteriormente se describe la notación gráfica desarrollada para representar las reglas de instanciación y, finalmente, se describe el diseño de un administrador de tareas, utilizado para cubrir aspectos complementarios del enfoque propuesto.

1. Representación de la documentación de los *frameworks*

Antes de estudiar los mecanismos de apoyo a *SmartBooks*, serán descritas las relaciones entre la documentación, su distintas representaciones y los mecanismos que los manipulan. *SmartBooks* propone documentar los *frameworks* utilizando, además de técnicas tradicionales de documentación, reglas funcionales y de consistencia (§IV.3.2). Ambos tipos de reglas son representadas gráficamente utilizando la notación gráfica *TOON*: para las reglas funcionales, esta representación es parcial, mientras que en el caso de las reglas de consistencia es total. La notación TOON es explicada en §VI.3.

A partir de las reglas de instanciación, dos representaciones intermedias son generadas: las acciones de instanciación, utilizadas por el planificador (§V.2), y reglas *Prolog*, utilizadas por el Administrador de Consistencia (§VI.4). Las acciones de instanciación son generadas, en su mayor parte, en base a la información provista por las reglas funcionales, siendo las reglas de consistencia utilizadas para complementar esa información. Por otro lado, el Administrador de Consistencia utiliza exclusivamente la información derivada de las reglas de consistencia.

Finalmente, tanto el planificador como el Administrador de Consistencia producen como resultado tareas de instanciación (§VI.2). En primer lugar, el planificador genera la lista de tareas que deberían ser ejecutadas, el plan de instanciación. Luego, a medida que el usuario ejecuta las tareas contempladas en el plan así como otras actividades no previstas, el Administrador de Consistencia creará nuevas tareas cuya necesidad de ejecución se derive de la documentación del *framework*.

La Figura 6.1 muestra gráficamente las relaciones entre estos conceptos.

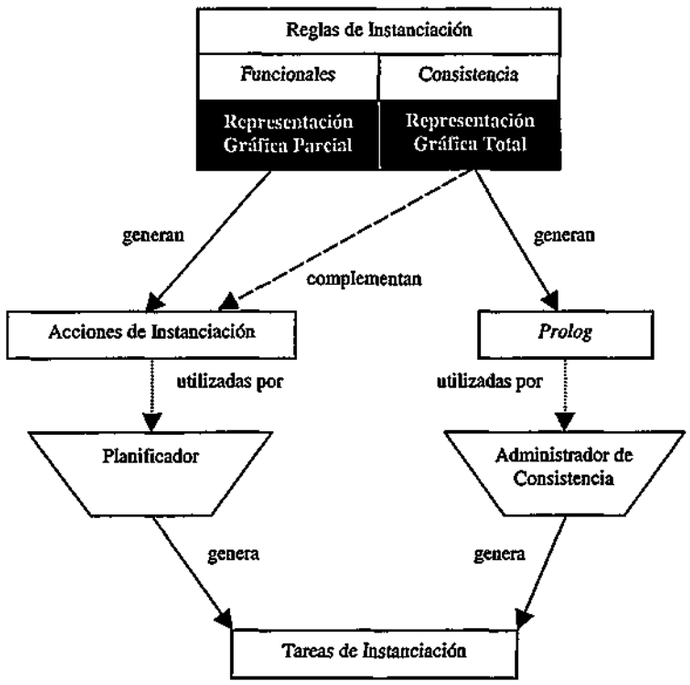


Figura 6.1 Relaciones entre las distintas porciones de documentación y sus representaciones

2. Tareas de Instanciación

El plan de instanciación producido por el planificador es presentado al usuario del *framework* utilizando el concepto de tarea. Es decir, desde el punto de vista del usuario, el proceso de crear una aplicación es el resultado de ejecutar una secuencia de actividades más o menos simples, denominadas tareas de instanciación. Este concepto de tarea de instanciación está basado en los modelos de tareas de usuario utilizados en el diseño de Interfaces de Usuario.

2.1. Modelos de Tareas en las Interfaces de Usuario

Al analizar el problema de documentación de las actividades necesarias para instanciar un *framework*, resulta natural trazar un paralelismo con los modelos de tareas de usuarios utilizados en las Interfaces a Usuario. La interpretación de las acciones del usuario en términos de la semántica de la aplicación es un problema clásico dentro de este área. Cualquier programa interactivo que utilice como entrada las manipulaciones del usuario con un ratón tiene que transformar los eventos básicos de movimiento y pulsación de los botones del ratón en comandos que ejecuten acciones asociadas a los mismos. El desarrollo de mecanismos que permitan un grado de abstracción más alto en la especificación dada por el programador de estas interpretaciones y la reutilización de partes de esta especificación en distintas aplicaciones ha tenido un gran auge en los últimos años [Pue97, SSC95, JWMP93, JJW95, Pat97].

Paralelamente a este proceso, se ha planteado desde distintas perspectivas la necesidad de una interpretación global de las acciones del usuario en términos de tareas complejas que la aplicación permite llevar a cabo. En términos generales, esta interpretación se hace mediante la definición de una jerarquía de tareas, de manera que las tareas de más bajo nivel corresponden a la interpretación semántica de las acciones más elementales, mientras que las de nivel superior están formadas por la concatenación de varias de nivel más bajo. La modelización de tareas de usuario en aplicaciones complejas permite una interacción más rica entre el usuario y el sistema. Gracias a ello el sistema es capaz de comprender con mayor precisión los objetivos del usuario, pudiendo ofrecerle apoyo para su consecución.

Los lenguajes y sistemas de especificación de tareas de usuario se han utilizado con finalidades muy diferentes, como las siguientes:

- Desarrollo de tecnología de diseño y desarrollo de interfaces basado en modelos, incorporando a dicha tecnología la utilización de modelos de tareas de usuario. En general, la información que se almacena en los modelos de interfaces permite agregar a las aplicaciones herramientas externas que razonan acerca del estado de la aplicación durante su ejecución, modificando el comportamiento de la interfaz. En esta línea, entre otros trabajos, se han construido *Mastermind* [SSC95], una herramienta para el diseño y desarrollo de interfaces de usuario sensibles al contexto, y Mecano [Pue97], una herramienta para el desarrollo de interfaces genéricas.
- Creación rápida de prototipos de la interfaz de la aplicación y ayuda a la detección de errores de diseño de la misma [JWMP93, JJW95].
- Creación de una metodología para la construcción de interfaces de usuario basadas en formularios [Pat98]. Esta metodología tiene como pieza fundamental la especificación de un conjunto de tareas que el usuario puede realizar en la interfaz.
- Desarrollo de sistemas para proporcionar ayuda al usuario basada en las tareas definidas para la aplicación [PP95], incluyendo algunos trabajos orientados a la generación semiautomática de sistemas de ayuda basados en modelos de tareas más avanzados [Con98]. Uno de estos trabajos, llamado *ATOMS* [GCRM98], ha sido la base del modelo de tareas de instanciación aquí propuesto.
- Construcción de sistemas capaces de detectar tareas realizadas por un usuario utilizando varias aplicaciones de forma consecutiva [ZK97].
- Desarrollo de cursos multimedia que se adaptan al perfil del usuario [CPR99]. Este trabajo está basado en la utilización de tareas docentes que dependen del perfil detectado en el usuario. Las tareas docentes son tareas interactivas realizadas por el sistema, incluyendo diálogos con el usuario a través de los cuales éste efectúa tareas que se le encomiendan.

2.1.1. Una Aplicación de los Modelos de Tareas de Usuario

A continuación se mostrará un modelo de tareas de usuario, propuesto para la generación automática de ayuda al usuario propuesto por [GCRM98]. Algunos de los conceptos de este modelo han sido tomados como base para el modelamiento de las tareas de instanciación que se presentan en esta tesis.

El sistema propuesto, denominado *HATS*, es un sistema de generación de ayudas sensibles al contexto para aplicaciones interactivas, el cual provee asistencia basada en una jerarquía de tareas que el usuario puede desarrollar. A medida que el usuario interacciona con la aplicación y lleva a cabo las tareas, los mensajes de ayuda son automáticamente actualizados y le ofrecen información sensible al contexto sobre las acciones necesarias para completar la tarea en ejecución, incluyendo referencias gráficas a los lugares de la ventana de la aplicación donde las acciones pueden ser ejecutadas.

El funcionamiento de *HATS* está basado en el sistema de administración de tareas *ATOMS*. El sistema *ATOMS* (*Advanced Task Oriented Management System*) es un sistema orientado a tratar con tareas de usuario complejas. Incluye información de contexto que es dinámicamente actualizada a medida que el usuario desarrolla diferentes actividades y la jerarquía de tareas es definida por reglas. *ATOMS* utiliza un analizador sintáctico, capaz de identificar las tareas globales que están siendo ejecutadas, así como también el contexto determinado de la información parcial extraída de las acciones del usuario.

ATOMS permite definir tanto tareas simples como complejas, pudiendo ambos tipos de tareas tener asociados parámetros. La definición se complementa con reglas que establecen la

descomposición de tareas en subtareas, restricciones sobre el valor de los parámetros, precondiciones de aplicación de las tareas y el ordenamiento de las tareas, que puede ser secuencial, paralelo o alternativo.

No toda la información relacionada con las tareas puede ser especificada declarativamente. *ATOMS* permite la definición de precondiciones por medio de patrones de tareas, que pueden ser definidos de forma declarativa, pero precondiciones generales también son posibles y estas deben ser programadas. Lo mismo ocurre con los métodos para el paso de parámetros de unas tareas a otras.

Para aquellas aplicaciones que incorporan un modelo de tareas de *ATOMS*, *HATS* es capaz de generar ayuda basada en tareas, proveyendo dos tipos de información:

- Explica tareas complejas en términos de tareas más simples
- Ofrece información gráfica sensible al contexto sobre los puntos de la ventana a través de los cuales el usuario puede interaccionar para conseguir los objetivos descritos en las diferentes tareas y cómo esta interacción debe tener lugar.

El usuario puede solicitar información de dos maneras distintas: o bien seleccionando una tarea de la lista jerárquicamente organizada de tareas de aplicación o bien seleccionando un sector de la ventana y preguntando qué acciones pueden ser ejecutadas allí.

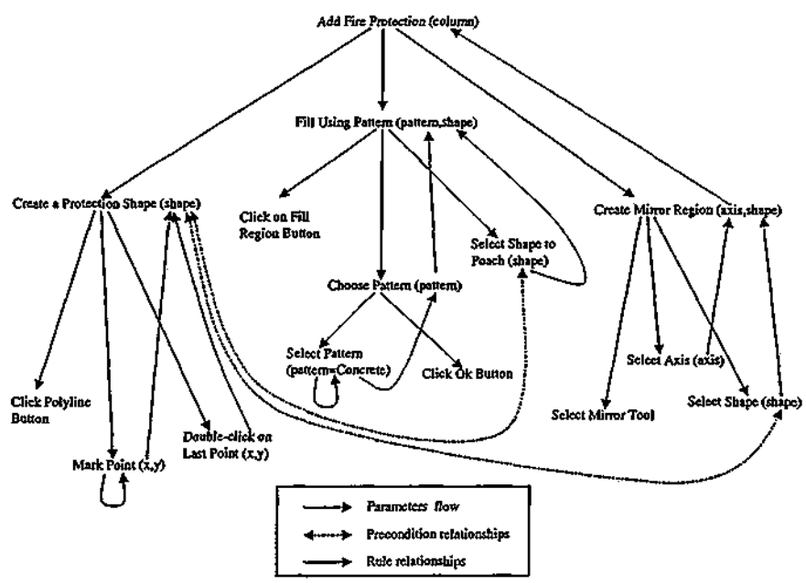


Figura 6.2 Una descomposición óptima para la tarea *Add Fire Protection to a Column.*

La Figura 6.2 muestra un ejemplo de modelo de tareas utilizado por *HATS* para brindar asistencia al usuario. Este ejemplo modela una de las posibles tareas de un sistema de diseño asistido por ordenador (CAD), concretamente el agregado de protecciones contra incendio a las columnas de un edificio.

Si bien tanto en los modelos de interfaces de usuario como en el proceso de instanciación de un *framework* las tareas son utilizadas para representar las actividades del usuario, la diferencia fundamental está dada por la información de la que disponen las respectivas herramientas de asistencia. Así, en el caso de las interfaces de usuario, la herramienta conoce las actividades que debe o puede ejecutar el usuario y su trabajo consiste en determinar cuál de estas actividades está siendo ejecutada en un momento dado. Para esto es suficiente contar con reglas que describan los tipos de actividades posibles y un analizador que permita determinar en base a interacciones básicas, qué actividad está ejecutando el usuario en cada momento. Este

tipo de analizador es similar a un analizador sintáctico, que partiendo de las reglas que definen un lenguaje, debe ser capaz de reconocer textos escritos en ese lenguaje.

En cambio, cuando se trata del proceso de instanciación de *frameworks*, es necesario generar una combinación de tareas básicas, que permitan obtener una funcionalidad dada. Comparando con el procesamiento de lenguajes, en este caso se trata de generar, a partir de reglas de formación, oraciones válidas (o textos más complejos) que describan una situación específica. Es este el principal motivo de la decisión tomada en *SmartBooks* de utilizar planificación para asistir en la instanciación de *frameworks*.

2.2. Representación de Planes de Instanciación en *SmartBooks*

Una tarea de instanciación es una actividad de programación básica, ejecutada como parte del proceso de instanciación de un *framework*, como por ejemplo la implementación de un método. Instanciar una aplicación implica llevar a cabo un conjunto de tareas de instanciación. En este contexto, darle información al usuario sobre qué debe hacer para crear una aplicación determinada, significa mostrarle la lista de tareas de instanciación que debería ejecutar.

Las tareas de instanciación que pueden ser creadas, tanto a través del planificador como del administrador de consistencia, pertenecen a clases predefinidas. Estas clases forman un conjunto acotado, el cual no puede ser extendido por los diseñadores y/o documentadores de los *frameworks*, como así tampoco por los usuarios. Tal como se explicó en el capítulo anterior (§V.2.2.2), las tareas son clasificadas en dos categorías: tareas de espera y tareas pendientes. Mientras que las tareas pendientes corresponden a las actividades que debería ejecutar el usuario como parte del plan de instanciación, las tareas de espera corresponden a ingresos de datos que el usuario debe realizar a los efectos de permitir elaborar el plan de instanciación. Se las representa como tareas de instanciación porque constituyen decisiones que el usuario debe tomar durante el proceso de instanciación, pero no forman parte del plan sugerido por el planificador. Otra particularidad de las tareas de espera es que, por ser utilizadas para la construcción del plan, sólo son creadas por el planificador y no por el administrador de consistencia.

Los tipos de tareas de espera existentes son cuatro: selección de método, selección de opción (de una lista), pedido de información y asignación de argumentos. Por otro lado, las tareas pendientes que pueden ser ejecutadas por el usuario corresponden a uno de estos tipos: creación y modificación de clases, creación y modificación de métodos y creación y modificación de atributos (variables de instancia).

En algunas situaciones, puede ser conveniente definir tareas compuestas o especializaciones de tareas. Por ejemplo, sería posible definir una tarea que involucre tanto la creación de un atributo como su inicialización y luego reutilizar esa definición. Sin embargo, este tipo de situaciones es fácilmente representable utilizando las acciones de instanciación del capítulo V.

2.2.1. Parámetros

Cada clase de tarea tiene asociado un conjunto de parámetros que ajustan su comportamiento. Por ejemplo, las tareas del tipo *DefineClass* tienen como parámetros el nombre de la clase y el nombre de la/s superclase/s. Dependiendo de la clase de tarea y del parámetro específico, estos parámetros no necesariamente tienen asignados valores específicos antes de ejecutar la tarea. Por ejemplo, en las tareas para modificar un método, el nombre del método y la clase siempre deben ser conocidos con anterioridad a la ejecución de la tarea. Esto es así porque se presume que no se creará una tarea para modificar *cualquier* método de *cualquier* clase. En cambio, las tareas del tipo *DefineClass* pueden no tener especificado antes de su ejecución el nombre de la clase que será creada. Es posible que el plan de instanciación requiera la creación

de una subclase de, por ejemplo, *Figure*[1], siendo el nombre de la clase provisto por el usuario durante la ejecución de la tarea. En otros casos, este mismo parámetro puede tener un valor fijo previo a la ejecución, como es el caso de las tareas de creación de clases vistas en el ejemplo de §V.2.5.4.

También existe la posibilidad de que un parámetro cuyo valor no es conocido tenga sin embargo una restricción sobre el conjunto de valores que puede tener. El típico caso es el nombre de la superclase en las tareas *DefineClass*; si la tarea especifica que la clase creada debe ser subclase de *Figure*, esto en realidad representa que debe ser subclase de *Figure* o de una clase que sea, directa o indirectamente, subclase de *Figure*.

Existen otros tipos de parámetros para los cuales su valor sólo es significativo al finalizar la ejecución de la tarea. Son parámetros cuya finalidad es representar información acerca de la tarea que pueda ser utilizada por tareas ejecutadas en forma posterior durante el proceso de instanciación.

De esta forma, los parámetros podrían ser clasificados en parámetros de entrada o parámetros de salida, de acuerdo a si su valor es significativo antes o después de la ejecución de la tarea, es decir, si son utilizados para transmitir información a o desde la tarea. En la práctica la situación común es que un parámetro puede ser tanto de entrada como de salida, siendo esto dependiente del contexto de aplicación de la tarea.

En consecuencia, los valores de los parámetros estarán siempre disponibles después de ejecutada la tarea, pero sólo algunos lo estarán antes. Entonces se hace necesario que cada clase especifique los parámetros que obligatoriamente deben tener un valor asignado antes de ejecutar la tarea. Un ejemplo de este tipo de parámetros, denominados obligatorios (*required*), es el caso del nombre del método en la tarea *UpdateMethod*. Para estos parámetros obligatorios, el mecanismo encargado de crear la tarea, ya sea el planificador o el administrador de consistencia (ver §VI.1), debe siempre proveer un valor. En caso contrario, la tarea no podrá ser ejecutada.

El siguiente es un ejemplo de creación de una tarea a partir de una acción de instanciación (las acciones de instanciación son explicadas en §V.2.1.1):

```
pendingTask ("DefineClass", [NewClass, SuperClass], Description])
        → defineSubclass (NewClass, SuperClass, Description)
```

En este ejemplo se especifica la creación de una tarea del tipo *DefineClass*, con dos parámetros, el nombre de la clase y el de la superclase. Se interpreta que el resto de los parámetros, no presentes en esta acción, no tienen un valor asociado previamente a la ejecución.

2.2.2. Clases de Tareas de Instanciación

En esta sección se describen brevemente los distintos tipos de tareas de instanciación disponibles. En primer lugar, se explican las tareas de espera.

- Selección de método (*SelectMethod*): el objetivo de este tipo de tareas es permitirle al usuario elegir el método de una clase dada que será utilizado en determinada porción del plan. Por ejemplo, si existen varios métodos para crear una instancia de la clase, este tipo de tareas permite que el usuario decida cuál será el que se utilice en cada caso. Los parámetros de esta tarea son la clase, el tipo de método (clase o instancia), la categoría a la que pertenece y una descripción textual para ofrecer información contextual al usuario. Como parámetro de salida tiene asociado el retorno, es decir, la elección del usuario.
- Selección de opción (*AskSelection*): Este tipo de tarea es utilizado para solicitar al usuario que elija una opción de una lista provista. Los parámetros de entrada utilizados en este caso son las opciones, una descripción textual y cómo salida el retorno de la tarea.

[1] Los ejemplos concretos ofrecidos aquí corresponden, como casi todos en este trabajo, al *framework* para editores gráficos *HotDraw* [Joh92], explicado en detalle en §II.2.3

- Petición de información (*GetInput*): son utilizadas para solicitar información textual al usuario. Los parámetros son un texto describiendo la información requerida y la respuesta del usuario.
- Asignación de argumentos (*AssignArguments*): Este tipo de tarea es utilizado cuando, durante el proceso de planificación, es necesario determinar los valores que serán utilizados como argumentos en la invocación de un método. Los parámetros de entrada asociados son el nombre del método, el nombre de la clase y el tipo de método (clase o instancia). Como salida, esta tarea tiene tres listas:
  - Lista de atributos: representa los atributos de la instancia que serán utilizados como argumentos. Deben ser accesibles dentro del entorno donde el método está siendo invocado.
  - Lista de variables locales: representa las variables locales al método que invoca que se utilizarán como argumentos.
  - El texto con que será invocado el método: surge de combinar el nombre del método con los argumentos que se utilizarán en la invocación.

Respecto a las tareas pendientes, ya se explicó más arriba que los tipos existentes representan la creación y modificación de clases, atributos y métodos. Las actividades de creación y modificación, si bien son similares, son consideradas como clases de tareas distintas porque las tareas de creación ignoran la existencia previa del elemento que están definiendo, mientras que las tareas de actualización necesitan del elemento para su ejecución. A continuación se explica cada grupo de tareas, con los parámetros y restricciones que pueden tener asociados.

- Creación / actualización de clase: Las tareas que crean y actualizan clases tienen como parámetro el nombre de la clase a crear, la superclase, la categoría a la que pertenece la clase, un comentario sobre su propósito, además de parámetros que indican si la clase es abstracta, activa y si es hoja o raíz de la jerarquía. Estos parámetros corresponden a las atributos que caracterizan a una clase de acuerdo al metamodelo propuesto por *frameworks* [Rat97]. De estos parámetros ninguno es obligatorio, porque su presencia depende del contexto en el que se utilice la tarea.
- Creación / actualización de atributo: Los parámetros asociados a este tipo de tareas son el nombre del atributo y de la clase, un comentario sobre el propósito del atributo y dos propiedades llamadas *ownerScope* y *visibility*, ambas tomadas de las definiciones de *frameworks*. La primera de estas propiedades describe el entorno en el cual es conocido el valor del atributo. Si el valor es *instance*, significa que su valor es conocido sólo desde una instancia, es decir, cada instancia de la clase tiene su propia copia; en cambio si su valor es *class*, el valor es compartido por todas las instancias de la clase. Un atributo cuyo *ownerScope* es *class* es equivalente a las variables de clase de *Smalltalk* o variables estáticas de C++, por ejemplo. La propiedad *visibility* describe la forma en que el atributo es visible desde fuera de la clase. Los valores posibles para esta propiedad son *public*, *protected* y *private*.
  En la práctica, los nombres del atributo y la clase son conocidos antes de la ejecución de la tarea, pero no son clasificados como obligatorios para ofrecer mayor flexibilidad de utilización. En el caso de la actualización de la definición de un atributo, se exige que tanto el nombre del atributo como de la clase sean parámetros obligatorios, porque carece de sentido el disponer la modificación de un atributo arbitrario de una clase indefinida.
- Creación / actualización de método: En este caso, los parámetros asociados son el nombre del método, de la clase a la que pertenece, el *ownerScope*, la visibilidad e indicadores que dicen si el método representa una consulta (*query*) o es abstracto. También se ha decidido

incluir como parámetro un *template* de código, que puede ser utilizado para dar una referencia clara de lo que se espera del código del método a modificar.

El caso de estas tareas es similar al de la creación y modificación de atributos en lo que respecta a la obligatoriedad de los parámetros. Sin embargo, en algunas implementaciones, como es el caso de *Smalltalk*, el nombre del método puede estar restringido por los argumentos utilizados en la invocación del método. Para las tareas de modificación de métodos se ha seguido el mismo criterio utilizado para la modificación de atributos.

Además, las tareas pendientes pueden ser clasificadas como optativas u obligatorias. Las obligatorias son aquellas que, de acuerdo al plan de instanciación elaborado, el usuario debería ejecutar para implementar la funcionalidad requerida. Las optativas, en cambio, son aquellas que pueden ser ejecutadas, pero que no son indispensables para el cumplimiento del plan.

2.3. Administrador de Tareas

Para administrar la ejecución de las tareas presentadas al usuario, se ha diseñado un módulo administrador de tareas. Este módulo administrador actúa, básicamente, como una agenda de asuntos pendientes, proveyendo al usuario la lista de tareas ya ejecutadas y las que están pendientes de ejecución. Utilizándolo, el usuario puede elegir tareas para, entre otras cosas, ejecutarlas, inspeccionarlas o deshacerlas (*undoing*). Además, puede rechazar una tarea, es decir, rechazar el plan del cual forma parte. En este caso, el planificador es utilizado para generar un plan alternativo, que no incluya la tarea rechazada. Existen algunas tareas que el mismo planificador cataloga de opcionales, es decir, que según la documentación no es indispensable ejecutar para obtener la funcionalidad especificada. Estas tareas *opcionales* ofrecen información sobre posibles actividades de programación destinadas a complementar aquellas que sí deberían ser ejecutadas, las tareas *obligatorias*.

El administrador de tareas también permite superar algunas limitaciones impuestas por el planificador. En concreto, permite administrar situaciones donde las tareas del plan no son la única causa de cambio del estado del mundo (el software desarrollado, en este caso) y la ejecución no atómica de tareas. La primera situación ocurre cuando el usuario quiere ejecutar tareas no previstas en el plan. El segundo caso es cuando el usuario comienza una tarea, la suspende, ejecuta una segunda tarea y luego vuelve a la primera.

En ambas situaciones, los conflictos pueden surgir sólo entre aquellas tareas que, formando parte del plan o no, manipulen el mismo objeto de software, como por ejemplo un método. El administrador de tareas previene estos conflictos, serializando la ejecución de tareas que actúen sobre el mismo componente.

El administrador de tareas también desempeña un papel importante en el mantenimiento de la consistencia entre el software desarrollado y la documentación del *framework*. En §VI.4 se explica el funcionamiento del Administrador de Consistencia, el cual utiliza reglas de consistencia para verificar la correctitud del software. Para que este mecanismo funcione, el administrador de tareas es encargado de avisarle al administrador de consistencia cada vez que el usuario finaliza una tarea, para que las reglas de consistencia puedan ser verificadas. Como resultado, el administrador de consistencia puede crear nuevas tareas pendientes, las cuales son transferidas al administrador de tareas.

En el capítulo VIII se presenta una implementación de este módulo en el entorno de programación *Smalltalk*. Allí se ofrece una descripción más amplia de la funcionalidad provista y también se presenta información sobre la implementación.

3. Representación Gráfica de las Reglas de Instanciación

Tal como se explicó en el capítulo IV, *SmartBooks* propone extender la documentación normal de un *framework* utilizando reglas de instanciación, las cuales son clasificadas en dos

categorías: reglas funcionales y reglas de consistencia. En el capítulo V se mostró cómo estas reglas son representadas a través de acciones de instanciación para ser utilizadas en la elaboración de planes de instanciación. En la próxima sección se explicará cómo las reglas de consistencia pueden ser usadas para guiar la instanciación del *framework* más allá de lo previsto por los planes de instanciación.

Independientemente de su utilización, es necesario proveer una notación en la que estas reglas puedan ser descritas por el autor de la documentación. Además, esta representación debería ser lo más sencilla posible a fin de no complicar excesivamente la labor del diseñador. Con este objetivo, facilitar el proceso de documentación del *framework*, se ha diseñado una representación gráfica para las reglas, denominada *TOON* (*Task and Object Oriented Notation*).

Debido a que las reglas de instanciación y las de consistencia tienen distintos requisitos, la notación *TOON* provee elementos específicos para cada tipo de regla. Si bien ambos tipos comparten muchos de los elementos básicos utilizados para construir las descripciones, la estructura global de cada una es distinta.

TOON no tiene la suficiente potencia expresiva para representar toda la información necesaria para el algoritmo de planificación. Por este motivo, el diseñador debe especificar parte de las reglas funcionales en forma textual, describiendo directamente las acciones de instanciación. Por otra parte, las reglas de consistencia sí pueden ser expresadas utilizando únicamente la notación gráfica. En esta sección se describirá la notación gráfica utilizada para representar las reglas funcionales, aunque siempre que no se indique lo contrario, la descripción es aplicable también a las de consistencia. En la próxima sección se explicará las particularidades de la notación para representar reglas de consistencia.

La notación *TOON* cubre los aspectos de la especificación de reglas de instanciación que consisten en patrones de componentes de software representables en *frameworks* [RJB99]. Estas descripciones consisten de componentes de software relacionadas entre sí; algunos de sus atributos contienen variables que pueden tomar valores en principio arbitrarios. En estas descripciones, las relaciones habituales en *frameworks* juegan el papel de restricciones que tienen que cumplir los componentes, tanto si forman parte de la precondición de aplicación de una regla, como si describen las condiciones que deben satisfacerse después de su aplicación.

Debido a esto, la notación *TOON* está basada en la notación propuesta por *frameworks* para el modelo de clases. El uso de esta notación como base permite obtener mayor consistencia con el resto de la documentación del *framework*. De todas formas, la representación de las reglas no depende de ninguna característica particular de la notación *frameworks*.

TOON utiliza básicamente dos tipos de objetos gráficos: clases y tareas de instanciación. Las clases pueden tener un nombre y un conjunto de atributos y métodos. Las tareas pueden tener un nombre (clase de tarea) y un conjunto de parámetros, de acuerdo a la representación de las tareas explicada en §VI.2. Las representaciones para clases y tareas son mostradas en la Figura 6.3.

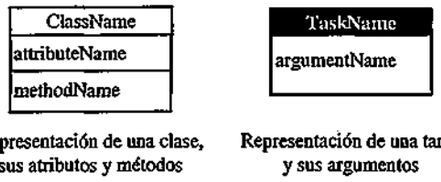


Figura 6.3 Representación de clases y tareas en *TOON*

Al definir una regla, algunas veces es necesario hacer referencia a un elemento genérico, que cumpla alguna condición, pero cuya identidad precisa no se conoce en el momento de la definición. Este es el caso cuando se quiere representar, por ejemplo, *cualquier* subclase de la clase *Figure*. *TOON* provee dos formas de representar esta situación. Una posibilidad es colocar en lugar del nombre del elemento (clase, método o atributo) la expresión *[Any]*. Por ejemplo, la

Figura 6.4-A muestra la representación de una subclase cualquiera de *Figure*. Otras veces, lo que se quiere representar es un elemento genérico, pero cuyo nombre resulta necesario para futuras referencias. En este caso, se utiliza una variable para hacer referencia al valor desconocido, representándolo a través de un nombre entre corchetes. Por ejemplo, si se quiere representar una situación donde un atributo y un método de una clase tengan el mismo nombre, se debe usar la notación de la Figura 6.4-B.

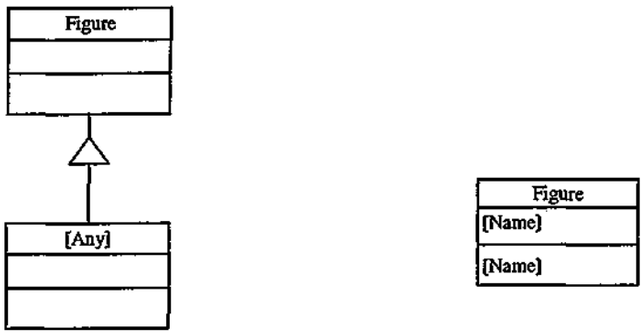


Figura 6.4 Utilización de variables en *TOON*.

TOON además provee medios para representar distintas relaciones entre estos elementos. Las clases se vinculan unas con otras a través de relaciones de herencia y composición. Los métodos se relacionan unos con otros a través de relaciones de invocación. Las tareas, por su parte, pueden estar relacionadas con otras tareas a través de relaciones de precedencia.

También es posible crear relaciones entre tareas y componentes de software. Existe una relación que representa el hecho de que la tarea produzca como resultado de su ejecución el elemento relacionado, sea este una clase, un método o un atributo. Este tipo de relación se denomina *creación* y se representa por una flecha a trazos, dibujada desde la tarea hasta el elemento creado. Por otro lado, existe un tipo de relación que permite representar aquellos elementos utilizados como datos de entrada para una tarea (*input*). En este caso también se utiliza una flecha a trazos. Como siempre se dibuja desde el elemento de entrada hacia la tarea, no existe ambigüedad con la relación de creación. Las representaciones de todas las relaciones se muestran en la Figura 6.5.

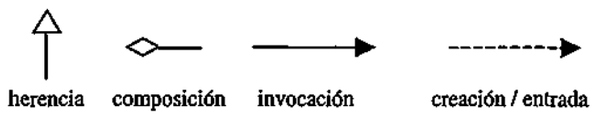


Figura 6.5 Representación de relaciones en *TOON*

Cuando se representan los datos de entrada para una tarea, se requiere que también se especifique a qué parámetro de la tarea corresponde el elemento de entrada. Por ejemplo, la especificación de la Figura 6.6 establece que debe ejecutarse una tarea de tipo *DefineMethod* y que el parámetro *methodName* tendrá un valor igual al nombre del método asociado de la clase *ClassA*. Esto quiere decir que en el momento de crearse la tarea, el nombre del método a ser creado será fijado de acuerdo con el nombre del método de la clase *ClassA* asociado. En el caso del vínculo que relaciona a la tarea con el método de la clase *ClassB*, no necesita tener asociado un nombre, porque se entiende que está haciendo referencia al producto de la ejecución de la tarea, un método en este caso.

Según se ha explicado al comienzo de este capítulo (§VI.1 y, especialmente, figura 6.1), las acciones de instanciación utilizadas por el planificador se generan a partir de las reglas de instanciación funcionales y algunas de las de consistencia, mientras que muchas acciones de consistencia únicamente se utilizan para la generación de código Prolog utilizado a su vez por el

Administrador de Consistencia. Por este motivo, existe un elemento que debe estar presente en la definición de una regla funcional: el de la acción de instanciación asociada. Para describir el efecto se utiliza un rectángulo con el texto, en cuyo interior debe describirse el efecto, aquí denominado funcionalidad. Este rectángulo se ubica, en general, en el extremo superior derecho de los diagramas.

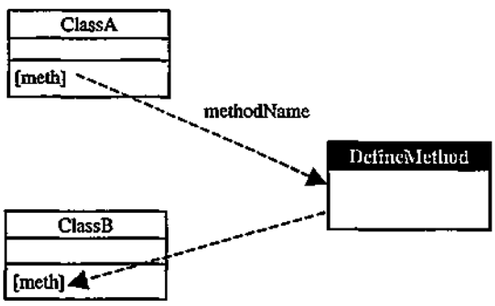


Figura 6.6 Ejemplo de utilización de parámetros en las tareas

Como resultado, la descripción gráfica de una regla funcional está formada por dos componentes básicos: la precondición y el efecto de la acción. La sección gráfica del diagrama describe la precondición, en tanto que el efecto es descrito textualmente. En el prototipo que ha sido implementado, esta descripción textual sigue el formato utilizado para la descripción de efectos en las acciones de instanciación (§V.2.2.1), de forma de poder traducir la regla a una acción comprensible por el planificador.

Un ejemplo de definición de regla funcional utilizando la notación *TOON* es presentado en la Figura 6.7. En este ejemplo se describen las tareas de instanciación que sería necesario ejecutar para implementar la funcionalidad descrita como "*Make Animated Drawing*". Informalmente, la regla de la Figura 6.7 define que para implementar la funcionalidad de diagramas animados, deberían ser ejecutadas dos tareas de instanciación. En primer lugar, se debe definir una nueva clase, subclase de *Drawing*. El nombre de esta clase no está especificado, aunque sí se asocia con la variable [*NewDrawing*], para su futura referencia en otras reglas. Es decir, si otras reglas describen condiciones relacionadas con una clase [*NewDrawing*], el motor de reglas tratará ambos casos como la misma clase. Una vez que la clase sea creada, esta variable quedará ligada con su nombre y en adelante las reglas harán referencia a esa clase concreta y no a una clase genérica. Luego de definirse la nueva clase, debe definirse un método para esta clase, llamado *step*.

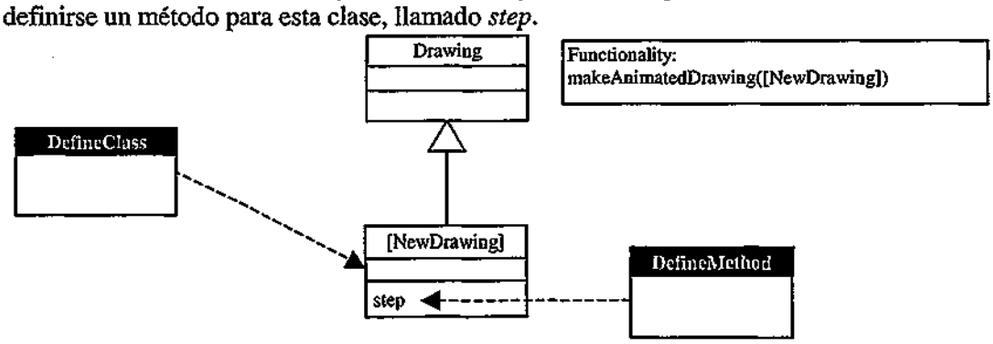


Figura 6.7 Definición de una regla funcional en *TOON*.

Los atributos, métodos y relaciones de una clase mostrados en una regla no constituyen una descripción completa de las propiedades de la clase, sino sólo de aquellas que son relevantes a la regla. En la Figura 6.7, por ejemplo, no se dice nada de las eventuales subclases de la clase representada por la variable [*NewDrawing*], ni de sus atributos o métodos, además del mencionado método *step*. Sólo se describen aquellos elementos que constituyen parte de la precondición o de los efectos de la regla.

La especificación de la Figura 6.7 es traducida a una acción de instanciación de la siguiente forma:

```
pendingTask("DefineClass", [NewDrawing, "Drawing"], Description),
        pendingTask("DefineMethod", [NewDrawing, "step"], Description2)
        → makeAnimatedDrawing([NewDrawing])
```

Algunas veces, la descripción de propiedades de los elementos manipulados por una tarea es utilizada como precondiciones o postcondiciones de la tarea, dependiendo si son elementos de entrada a la tarea o producidos por ésta. Por ejemplo, considérese la regla definida en la Figura 6.8. Esta regla describe las tareas necesarias para dotar a la interfaz de usuario de un editor con animación de la funcionalidad de arrancar y detener la animación. La especificación muestra que para implementar la funcionalidad descrita es necesario ejecutar tres tareas: crear una subclase de *Tool*, y para esta nueva clase definir dos métodos, *activate* y *deactivate*. Además, la especificación impone restricciones adicionales sobre los nuevos métodos. Así, para el método *activate*, prescribe que debe invocar al método *startAnimation* del nuevo diagrama que está siendo definido, mientras que en el caso del método *deactivate* establece que debe invocar a *stopAnimation*. Estas restricciones son asociadas a las tareas correspondientes y cuando son ejecutadas por el usuario, se comprueba si las condiciones se cumplen. Si no es así, el usuario es informado sobre el error contenido en la tarea ejecutada.

Debe notarse que en la Figura 6.8 no hay ninguna tarea de creación asociada con la subclase de *Drawing* ni con sus métodos. Esto significa que la regla está haciendo referencia a alguna subclase existente de *Drawing*. Esta subclase puede haber sido creada por tareas derivadas de otras reglas, aunque la regla de la Figura 6.8 no hace ninguna suposición al respecto. De la misma forma, no se establece nada acerca de la creación de los métodos *startAnimation* y *stopAnimation*, los cuales pueden estar implementados en la clase o heredados de implementaciones estándares existentes en *Drawing*.

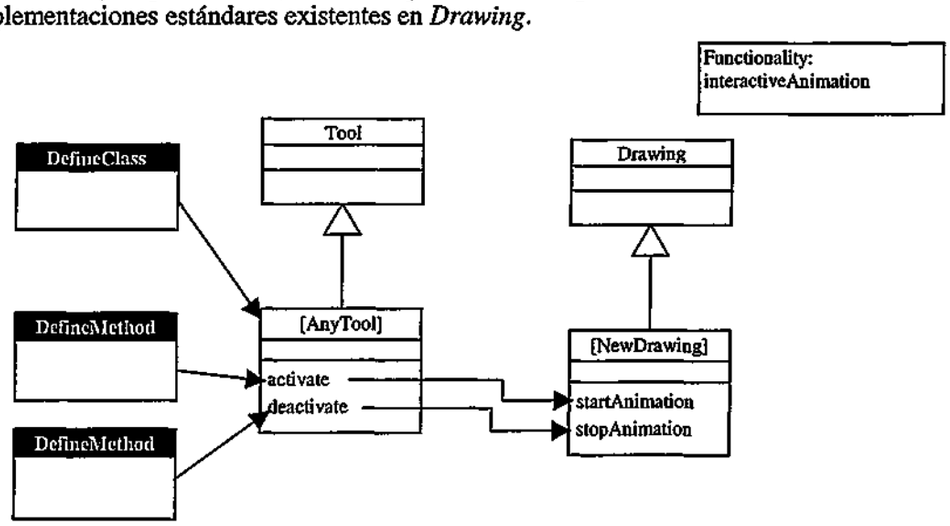


Figura 6.8 Ejemplo de definición de regla funcional con restricciones

Por último, existe una notación para indicar que una tarea es opcional. Las tareas opcionales son representadas con su nombre entre corchetes.

Teniendo en cuenta los elementos que pueden formar parte de una acción de instanciación, algunos de ellos no poseen actualmente representación en *TOON*. La mayor limitación actual es la falta de capacidad para representar precondiciones no relacionadas con la ejecución de tareas. Específicamente, sólo es posible describir aquellas precondiciones que únicamente utilizan los operadores *waitingTask* y *pendingTask*. En todos los otros casos la funcionalidad del *framework* debe ser documentada directamente a través de las acciones de instanciación descritas en el capítulo anterior.

En el caso de los efectos, estos deben ser representados textualmente, si bien este texto está integrado a la descripción gráfica de las precondiciones. Como ya se dijo, esta descripción textual debe respetar el formato utilizado por las acciones de instanciación. Esta necesidad de representar textualmente los efectos siguiendo un formato fijo dificulta su especificación.

En la próxima sección, al describir en detalle las reglas de consistencia, se explican características adicionales de *TOON* específicas para las reglas de consistencia.

4. Administrador de Consistencia

El objetivo de un administrador de consistencia en el contexto de *SmartBooks* es complementar la asistencia provista por el algoritmo de planificación al usuario del *framework*. Su función es ayudar a mantener el software desarrollado por los usuarios dentro de los protocolos y leyes que gobierna el diseño del *framework* y que muchas veces ofrecen información útil sobre cómo está previsto que ese *framework* sea especializado.

Estos protocolos en general materializan criterios de diseño que tienen por objetivo dotar a las aplicaciones basadas en el *framework* de ciertas características no funcionales, como reusabilidad, comprensibilidad, portabilidad, modificabilidad, etc. Por ejemplo, en los diseños que materializan un modelo arquitectónico de niveles (*layers*), una norma que se debe respetar es que las componentes de un nivel dado sólo se comuniquen con componentes de niveles adyacentes. Este tipo de arquitectura dota al sistema, entre otras cosas, de portabilidad. Un ejemplo más sencillo sobre información de diseño que resulta útil para guiar el proceso de instanciación son los métodos *abstractos*, que indican los puntos a través de los cuales está previsto que el *framework* sea especializado (*hot-spots*).

A partir de estas leyes de diseño, dada una tarea de instanciación ejecutada por el usuario, la porción de software resultante (ya sea completamente nueva o resultado de una modificación) puede dar lugar a una de tres situaciones:

- Respeta las leyes de diseño del *framework*
- Contradice inevitablemente alguna de las leyes del diseño
- Las leyes no son respetadas, pero la ejecución de otras tareas puede evitar que se contradigan.

De acuerdo a esto, un administrador de consistencia debe ser capaz de observar las actividades del usuario, detectar situaciones en las que el diseño no fue mantenido y reaccionar ofreciendo las indicaciones necesarias para que el software desarrollado no viole las leyes subyacentes. Para implementar este comportamiento, es necesario disponer de información que describa explícitamente estas leyes de diseño. Este es el papel de las reglas de consistencia, introducidas en §IV.3.2.2.

Utilizando estas reglas, un administrador de consistencia actúa básicamente como un motor de reglas, el cual es activado cada vez que el usuario finaliza una tarea. En ese momento se deben comprobar aquellas reglas que como acción disparadora tienen una tarea del tipo de la que ha finalizado y para cada una de estas reglas se debe evaluar la condición de inconsistencia. Finalmente, se debe crear las tareas prescritas por aquellas reglas cuyo estado inconsistente corresponda al estado actual del software que está siendo desarrollado, o informar al usuario del error correspondiente.

Este sistema es independiente del planificador. Por estar basado en las reglas de consistencia, su funcionamiento no depende de los objetivos funcionales, sino sólo del diseño del *framework*. Esto permite, como ya se explicó, ofrecer asistencia aún cuando la funcionalidad implementada no esté contemplada en la documentación incluida en el sistema de reglas o no se estén siguiendo los pasos previstos por esta documentación.

4.1. Representación Gráfica de las Reglas de Consistencia

Las reglas de consistencia también son descritas utilizando la notación *TOON*. En este caso, las reglas pueden ser especificadas en su totalidad utilizando la notación gráfica, sin ser necesaria la utilización de especificaciones textuales para complementarla. Esto es debido a que las reglas de consistencia no deben hacer referencia a condiciones del plan de instanciación ni a efectos complejos. Tanto las precondiciones como los efectos pueden ser descritos en términos de tareas y componentes de software. Otra consecuencia de esto es que la estructura de la especificación gráfica de una regla de consistencia corresponde a la estructura descrita para estas reglas en §IV.3.2.2: acción o acciones de origen, estado de inconsistencia y acciones reparadoras.

TOON provee algunas características específicamente diseñadas para la definición de reglas de consistencia. La primera de ellas es la posibilidad de representar la no existencia de un elemento. Por ejemplo, la Figura 6.9-A muestra la representación de una clase que no tiene definido un método *forbidden*. En cambio, la Figura 6.9-B representa la condición de que no exista una clase que posea el método *forbidden*.

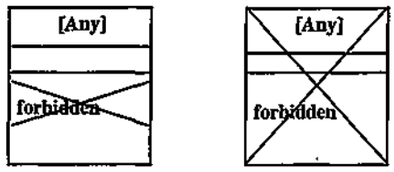


Figura 6.9 Extensiones de TOON para la representaciones de reglas de consistencia

Otra extensión provista por *TOON* para las reglas de consistencia es la posibilidad de definir relaciones generales entre clases. La Figura 6.10, por ejemplo, describe la condición de que la clase *C1* sea distinta de la clase *C2*. Las condiciones que pueden especificarse en este tipo de relaciones corresponden a aquellas que pueda interpretar el motor de reglas. Por este motivo, en su forma general la relación tiene una dirección definida, aunque dependiendo del tipo específico de condición, la dirección de la relación puede ser indistinto.

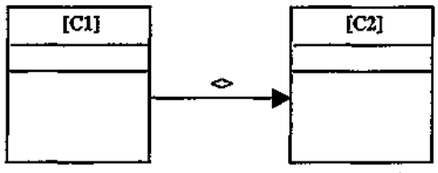


Figura 6.10 Representación de relaciones especiales entre clases

Por último, en las reglas de consistencia es necesario distinguir entre el antecedente y el consecuente de la regla, es decir, qué componentes forman parte de la precondición y cuáles del resultado de la regla. Para facilitar esta distinción se representa explícitamente cuál es el componente cuyo estado se quiere mantener consistente en cada regla, el *sujeto* de la regla. Por ejemplo, si se quiere especificar que cierta acción debe ejecutarse si un determinado método *M* es invocado por *M1*, *M2* y *M3*, el sujeto de esa regla será el método *M*, formando los otros parte de la precondición de la regla. Esto debe ser especificado porque no siempre es posible deducirlo a partir de la definición de la regla. La representación gráfica utilizada para esto es una doble línea alrededor del componente, sea clase, método o atributo.

La Figura 6.11 muestra un ejemplo de la definición de una regla de consistencia utilizando estas extensiones. Esta regla describe la situación que surge del hecho de que el método *displayOn:* esté definido en la clase *Figure* como un método abstracto. Básicamente, la regla define que cuando se crea una subclase de *Figure* (el sujeto de la regla), al menos una de las siguientes condiciones debe ser cierta:

- La nueva clase es abstracta, es decir, no se crean instancias de la misma
- La nueva clase u otra clase intermedia, distinta de *Figure*, implementan el método *displayOn:*

En el caso de que no se cumpla ninguna de estas condiciones, se debe ejecutar una tarea para implementar dicho método en la nueva clase.

Aquí puede verse en un ejemplo concreto cómo se materializan las tres componentes que integran toda regla de consistencia:

- Acciones que originan el estado inconsistente: en este caso puede ser tanto la ejecución de una tarea del tipo *DefineClass*, que produzca una nueva clase, subclase de *Figure*, como una tarea del tipo *DefineMethod*, que produzca un método que contenga una invocación al *new* de una subclase de *Figure* ya existente.
- Determinación del estado de consistencia: en este ejemplo, para que se produzca un estado inconsistente, debe cumplirse que la nueva clase herede (directa o indirectamente) de *Figure*, que no tenga definido el método *displayOn:*, que ninguna superclase suya tenga definido el método *displayOn:* y que se creen instancias de la clase (en *Smalltalk* la forma de verificar esta última condición es comprobar la invocación al método *new* de la clase).
- Acciones reparadoras: la regla establece que para respetar el diseño del *framework*, el usuario debe implementar el método *displayOn:* para la nueva clase. Es decir, ejecutar una tarea del tipo *DefineMethod* que produzca el método referido.

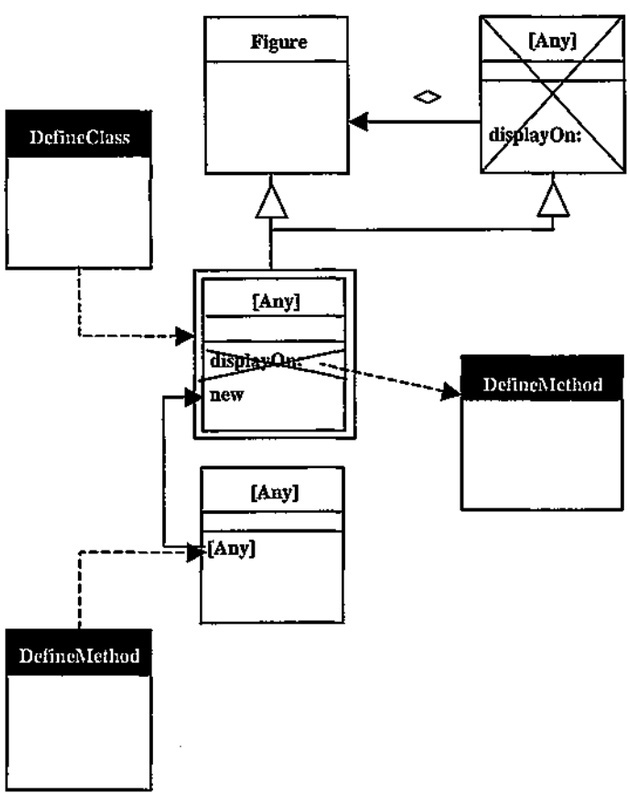


Figura 6.11 Ejemplo de definición de regla de consistencia

Como ya se ha explicado en el capítulo IV, una propiedad muy útil de las reglas de consistencia es que muchas veces es posibles generalizarlas, para describir una propiedad que debe ser cumplida por distintos componentes del *framework*.

Por esto motivo, *TOON* provee una notación especial que permite especificar que una determinada regla no especifica propiedades de un componente en particular, sino propiedades que deben cumplir todos los componentes que desempeñen determinado papel en el sistema. Por ejemplo, la propiedad que deben cumplir todas las clases que hereden de una clase con métodos abstractos.

La generalización de la Figura 6.11, la cual ya ha sido descrita textualmente en IV.3.2.2, es mostrada en la Figura 6.12.

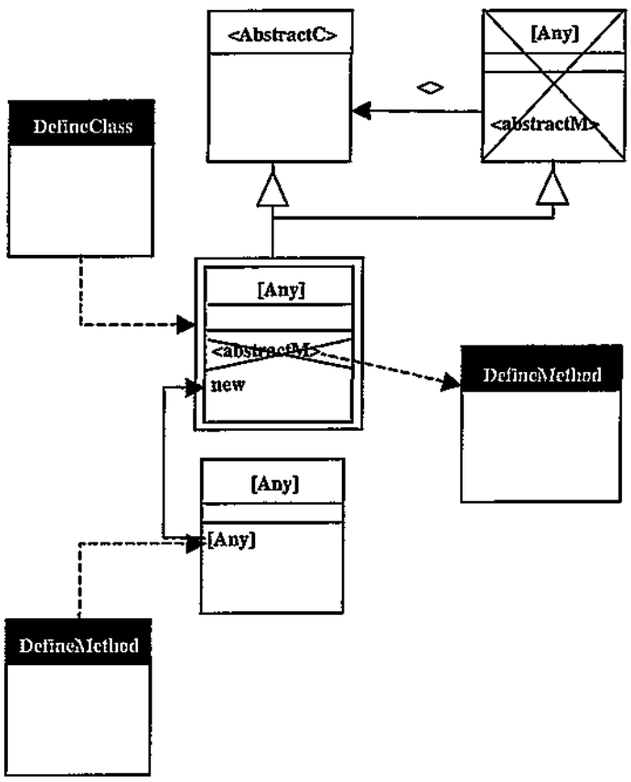


Figura 6.12 Generalización de una regla de consistencia

Esta figura describe la situación derivada de que un método sea definido como abstracto. En esta definición existen dos papeles: el de clase abstracta (*AbstractC*) y el de método abstracto (*abstractM*). Los corchetes angulares (< y >) se utilizan para indicar esta situación. En base a esta definición, el diseñador sólo necesita decir que un método dado de una clase determinada es abstracto, y la regla será automáticamente asociada con los componentes concretos.

Este tipo de generalizaciones son utilizadas especialmente con los patrones de diseño, tal como se explica en el capítulo VIII.

VII Ejemplo de Utilización de SmartBooks

En este capítulo se describirá el funcionamiento de los módulos principales de la herramienta durante el proceso de instanciación del *framework*. Para ello se utilizarán como base las acciones de instanciación para el *framework HotDraw* descritas en el Anexo B. El ejemplo aquí presentado es básicamente una extensión del ejemplo presentado en §IV.4 y §V.2.6, esta vez utilizando todas las reglas de instanciación del framework y mostrando la forma en que interaccionan los distintos módulos del sistema.

Con este objetivo, se parte de una especificación funcional, mostrando los pasos dados por el planificador para generar el plan de instanciación. Luego se mostrará como se asiste al usuario en la ejecución de este plan, a través del Administrador de Tareas y el Administrador de Consistencia.

Cabe destacar que el contenido de este capítulo puede resultar, en ciertos aspectos, demasiado detallado. El objetivo es brindar una idea clara de cómo funcionan los mecanismos de soporte a *SmartBooks*. Para ayudar a la lectura del ejemplo y resaltar los aspectos más significativos, se han subrayados los pasajes del texto que describen las características más importantes del proceso de instanciación.

1. Descripción de funcionalidad

El primer paso en la utilización de la herramienta por parte del encargado de crear una aplicación es definir la funcionalidad requerida para dicha aplicación. Por motivos de claridad, una vez más se reproduce aquí la descripción funcional presentada en §IV.4 para la creación de un editor de diagramas *PERT* utilizando *HotDraw*:

> *"Debe ser posible crear de forma interactiva objetos gráficos que representen tareas, y relacionar estas tareas a través de vínculos de precedencia. Las tareas deben tener una representación visual para sus atributos y dos de los atributos serán editados a través de la interfaz gráfica de usuario: para editar Duration se utilizará un tool, mientras que con EndDate se usará un menú. Además, cada atributo puede estar relacionado con otros, tanto de la misma tarea como de tareas relacionadas."*

Tal como se describió en §V.2.6.3, a partir de esta especificación se genera la siguiente lista de objetivos para el algoritmo de planificación:

I.	*useFigure("PertTask", "")*
II.	*relateAttributes ("PertTask")*
III.	*relateAttributes("PertTask", "PertTask")*
IV.	*editAttribute("Tool", "PertTask", "Duration")*
V.	*editAttribute("Menu", "PertTask", "EndDate")*
VI.	*makeRelationship ("PertTask", "PertTask")*

La funcionalidad representada por estos objetivos es la siguiente:

- representar un tipo de objeto llamado *PertTask*, los cuales deben poder ser editados individualmente
- establecer relaciones entre atributos de una misma *PertTask*
- establecer relaciones entre atributos de distintas *PertTasks*

- editar a través de la interfaz de usuario el atributo *Duration* de las *PertTask*, utilizando un *Tool*
- editar el atributo *EndDate*, en este caso utilizando un menú
- establecer relaciones entre las *PertTask*

La lista de objetivos funcionales será la *agenda* con que se invoque al algoritmo *PIT* (§V.2.5). Concretamente, se hará:

agenda = { ⟨useFigure ("PertTask",""), A_∞⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"),A_∞⟩ }.

Ejecutando el plan de instanciación resultante a partir de estos requisitos, debe ser posible obtener, entre otras cosas, el método de inicialización *initializeAt*: descrito en §IV.1, Figura 4.1.

2. Planificación

Las acciones de instanciación utilizadas para construir un plan de instanciación para los objetivos descritos son las presentadas en el anexo B (*framework HotDraw*) y A (acciones primitivas). Para facilitar la lectura de esta sección, la notación utilizada para representar las acciones que forman parte del plan es la siguiente:

- $A_{m,n}$ representa la *n*-ésima aplicación de la acción *m* del framework *HotDraw*.
- $P_{m,n}$ representa la *n*-ésima aplicación de la acción primitiva *m*.
- A_m y P_n representan, respectivamente, la acción *m* del *framework HotDraw* y la acción primitiva *n* (su definición, no una aplicación específica).

Además, si bien a los nombres de las variables se les agrega un subíndice para distinguir aquellas variables que tengan el mismo nombre, en este ejemplo el subíndice sólo será utilizado cuando exista la posibilidad de confundir variables.

En primer lugar, *PIT* debe considerar aquellas acciones que están definidas obligatorias, independientemente de la funcionalidad requerida. Estas acciones son identificadas por el objetivo *$required*, siendo que las acciones de *HotDraw* incluye una de estas acciones (A_1). Entonces la condición de esta acción es incorporada a la agenda, junto con los objetivos iniciales determinados por la funcionalidad descrita:

agenda = *{ ⟨defineEditor, A_1⟩, ⟨useFigure ("PertTask",""), A_∞⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩ }.*

Para resolver el primer objetivo, se puede aplicar la acción A_7. En consecuencia, la lista de acciones aplicadas (*A* en §V.2.5) será

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩}*

y la agenda será

agenda = *{ ⟨not(moreThanOneDrawing), $A_{7,1}$⟩, ⟨defineClass(Editor, "Editor", "The class that implements the editor itself"), $A_{7,1}$⟩, ⟨defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor") , $A_{7,1}$⟩, ⟨useFigure ("PertTask",""), A_∞⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool",*

"PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩ }.

El primer objetivo de la agenda está relacionado con un operador, *not*. Por lo tanto, el planificador verificará si la condición asociada no está presente en el estado actual del mundo. Como esto es verdadero, la condición se cumple y la planificación puede continuar, colocando en el plan la información necesaria para que esta condición no sea alterada por posteriores acciones. Es decir, se agrega el siguiente elemento a la lista de vínculos causales *L*:

$\{ A_0 \rightarrow^{not(moreThanOneDrawing)} A_{7,1} \}$

La idea de incorporar esta relación es evitar que una acción incorporada posteriormente en el plan, se ubique antes de $A_{7,1}$ y deshaga la condición necesaria para A_7.

A continuación, se debe satisfacer el objetivo ⟨*defineClass(Editor, "Editor", "The class that implements the editor itself")*, $A_{7,1}$⟩, pudiendo utilizarse para ello la acción primitiva P_1. Esta acción se satisface inmediatamente, creándose en el proceso una tarea pendiente de definición de clase.

El siguiente objetivo de la agenda es ⟨*defineMethod(...)* , $A_{7,1}$⟩, el cual puede ser resuelto utilizando la primitiva P_9. Esta primitiva también deja como resultado la creación de una tarea pendiente, en este caso para la implementación de un método. Como consecuencia, la lista de tareas pendientes creada hasta el momento es

tareas = *{DefineClass(Editor, "Editor", "The class that implements the editor itself", "Required"), DefineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor"), "Required")}.*

El próximo objetivo es el primero derivado directamente de la funcionalidad requerida por el usuario; *PIT* buscará de satisfacer el objetivo ⟨*useFigure ("PertTask","")*, A_∞⟩. La acción de instanciación 30) de *HotDraw*, A_{30}, describe cómo satisfacer este objetivo, donde *ClassName = PertTask, Description = ""*. Entonces el conjunto de acciones aplicadas ahora será

A = *{⟨$A_{30,1}$, (useFigure("PertTask","")*, *changed("PertTask")⟩)}*,

agregando las precondiciones *defineClass("PertTask", "Figure", "")* y *defineFigureLayout("PertTask")* a la agenda. Por lo tanto, las principales variables del planificador tendrán los siguientes valores:

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask","")*, *changed("PertTask")⟩ }*

agenda = *{ ⟨defineClass("PertTask", "Figure", Description_1), $A_{30,1}$⟩, ⟨defineFigureLayout("PertTask"), $A_{30,1}$⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩ }.*

A continuación es considerado el objetivo ⟨ *defineClass("PertTask", "Figure", "")*, $A_{30,1}$ ⟩, el cual es resuelto a través de la acción de instanciación primitiva 1 (P_1). Por esta acción, se incluye en la lista de tareas a ser ejecutadas la tarea *DefineClass("PertTask", "Figure", "")*. En este momento, la lista de tareas pendientes es

Tareas = *{DefineClass(Editor, "Editor", "The class that implements the editor itself", "Required"), DefineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection*

new) yourself"), [] "This method should be updated every time a tool is added to the editor"), "Required"), DefineClass("PertTask", "Figure", "", "Required")}

El próximo objetivo a satisfacer es ⟨*defineFigureLayout("PertTask")*, $A_{30,1}$ ⟩, para el cual la primera acción encontrada es la A_8, con *ClassName="PertTask"*. Una precondición de esta acción es que ninguna acción aplicada con anterioridad haya hecho verdadero *composite("PertTask")*, lo cual se cumple. En consecuencia, la segunda precondición (*optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")*) es agregada a la agenda. En este momento, la lista de acciones aplicadas es

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask",""), changed("PertTask")⟩, ⟨$P_{1,2}$, defineClass("PertTask", "Figure", Description_1)⟩ , ⟨$A_{8,1}$, defineFigureLayout("PertTask"), simple("PertTask")⟩ }*

agenda = *{ ⟨optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], ""), $A_{8,1}$⟩, ⟨relateAttributes ("PertTask"), A_∞⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩ }*.

Siguiendo con los objetivos de la agenda, el próximo que se intenta satisfacer será ⟨*optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")*, $A_{8,1}$⟩. Para esto se utiliza la acción P_{10}. Por efecto de esta primitiva se agrega a la lista de tareas pendientes *DefineMethod("PertTask", "displayOn:", ...; "Optional")*. El estado de la lista de tareas pendientes es ahora

tareas = *{DefineClass(Editor, "Editor", "The class that implements the editor itself", "Required"), DefineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor"), "Required"), DefineClass("PertTask", "Figure", "", "Required"), DefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "", "Optional")}*

El siguiente objetivo de la agenda es ⟨*relateAttributes ("PertTask")*, A_∞⟩. Este objetivo puede ser satisfecho aplicando A_{26}. En consecuencia, la lista de acciones aplicadas será

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask",""), changed("PertTask")⟩, ⟨$P_{1,2}$, defineClass("PertTask", "Figure", Description_1)⟩ , ⟨$A_{8,1}$, defineFigureLayout("PertTask"), simple("PertTask")⟩, ⟨$P_{10,1}$, optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")⟩, ⟨$A_{26,1}$, relateAttributes("PertTask"), changed("PertTask"), useConstraints ⟩ }*

en tanto que

agenda = *{ ⟨selectMethod("HotDrawConstraint", "class", "instance creation", Description, Message), $A_{26,1}$⟩, ⟨assignArguments(Message, "initializeAt:", "PertTask", Attributes, LocalVars, MessageString), $A_{26,1}$⟩, ⟨defineAttributeList ("PertTask", Attributes), $A_{26,1}$⟩, ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), LocalVars, Description), $A_{26,1}$⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩ }*.

Continuando el proceso de planificación, el objetivo que debe ser considerado ahora es ⟨*selectMethod("HotDrawConstraint", "class", "instance creation", Description, Method),* $A_{26,1}$⟩, para lo cual debe ser aplicada la primitiva P_{18}. Esta acción crea una tarea de espera, por lo cual el proceso de planificación debe detenerse y esperar que el usuario elija un método de la clase dada, dentro de la categoría especificada. Con este objetivo, la tarea de espera es transferida al Administrador de Tareas, el cual presenta la tarea al usuario para su ejecución. El Administrador de Tareas provee, además, asistencia para la ejecución de cada tarea. En el caso de las tareas de selección de un método, el Administrador presenta al usuario la lista de métodos pertenecientes a la categoría correspondiente y espera su elección.

En el ejemplo aquí presentado, la categoría *"instance creation"* de los métodos de clase de la clase *"HotDrawConstraint"*, contiene, principalmente, las siguientes opciones:

editOn: aVariable
equalityBetween: var1 and: var2
equalityBetween: var1 and: var2 plusOffset: aConstantOffset
greaterThanEqualBetween: var1 and: var2
greaterThanEqualBetween: var1 constant: c
left: l right: u percent: p offset: offset equal: aVariable
maximize: aVariable with: aVariableCollection default: aConstant
multiply: aVar1 and: aVar2 equal: aVar3
origin: origin corner: corner xPercent: x yPercent: y offset: offset equal: aPoint
plus: aVar1 and: aVar2 equal: aVar3
set: aVar equalTo: aValue

Una vez que el usuario ha elegido, el Administrador de Tareas retorna la opción seleccionada al planificador, para que continúe con la creación del plan.

En el caso de las restricciones entre las variables de cada *PertTask*, el método escogido para relacionarlas es *plus:and:equal:*, el cual necesita de tres parámetros. Entonces en el plan, la variable *Method* de $A_{26,1}$ se asocia con el nombre del método, *plus:and:equal:*.

El próximo objetivo de la agenda es ⟨*assignArguments("plus:and:equal:", "initializeAt:", "PertTask", Attributes, LocalVars, MessageString),* $A_{26,1}$⟩, que también se satisface utilizando una acción primitiva, P_{21}. Aquí nuevamente se recurre a una tarea de espera para solicitar información al usuario, en este caso, cuáles serán los valores utilizadas como parámetros del método seleccionado en el paso anterior. En el ejemplo del diagrama *Pert*, el primer argumento será un atributo del objeto (variable de instancia en la terminología *Smalltalk*), llamado "startDate", mientras que los otros dos corresponden a variables locales del método *initializeAt:*, concretamente "*duration*" y "*endDate*".

En este punto del proceso, la lista de acciones instanciadas es:

A = {⟨$A_{1,1}$, *$required*⟩, ⟨$A_{7,1}$, *defineEditor, beingDefined(Editor), changed("PertTask")*⟩, ⟨$P_{1,1}$, *defineClass(Editor, "Editor", "The class that implements the editor itself")*⟩, ⟨$P_{9,1}$, *defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")*⟩, ⟨$A_{30,1}$, *(useFigure("PertTask",""), changed("PertTask")*⟩, ⟨$P_{1,2}$, *defineClass("PertTask", "Figure", Description_1)*⟩, ⟨$A_{8,1}$, *defineFigureLayout("PertTask"), simple("PertTask")*⟩, ⟨$P_{10,1}$, *optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")*⟩, ⟨$A_{26,1}$, *relateAttributes("PertTask"), changed("PertTask"), useConstraints* ⟩, ⟨$P_{18,1}$, *selectMethod("HotDrawConstraint", "class", "instance creation", "", "plus:and:equal:")*⟩, ⟨$P_{21,1}$, *assignArguments("plus:and:equal:", "initializeAt:", "PertTask", ["startDate"], ["duration", "endDate"], "HotDrawConstraint plus:startDate and:duration equal:endDate")*⟩}

y la agenda es:

agenda = *{ ⟨defineAttributeList ("PertTask", ["startDate"]), $A_{26,1}$⟩, ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), ["duration", "endDate"], Description_2), $A_{26,1}$⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_{∞}⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_{∞}⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_{∞}⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_{∞}⟩ }.*

A continuación debe ser satisfecho *⟨defineAttributeList ("PertTask", ["startDate"]), $A_{26,1}$⟩*, lo que es conseguido utilizando A_6. Esta acción utiliza el operador *forEach* y como, en este caso, la lista contiene un solo elemento, el siguiente elemento es agregado a la agenda: *⟨defineAttribute ("PertTask", "startDate", ""), $A_{6,1}$⟩*.

Es este último objetivo el que debe ser satisfecho a continuación. Para esto se aplica la acción A_5, dando como resultado la siguiente configuración de acciones aplicadas y agenda:

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask",""), changed("PertTask")⟩, ⟨$P_{1,2}$, defineClass("PertTask", "Figure", Description_1)⟩ , ⟨$A_{8,1}$, defineFigureLayout("PertTask"), simple("PertTask")⟩, ⟨$P_{10,1}$, optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")⟩, ⟨$A_{26,1}$, relateAttributes("PertTask"), changed("PertTask"), useConstraints), ⟨$P_{18,1}$, selectMethod("HotDrawConstraint", "class", "instance creation", "", "plus:and:equal:")⟩, ⟨$P_{21,1}$, assignArguments("plus:and:equal:", "initializeAt:", "PertTask", ["startDate"], ["duration", "endDate"], "HotDrawConstraint plus:startDate and:duration equal:endDate")⟩, ⟨$A_{6,1}$, defineAtributeList("PertTask", ["startDate"])⟩, ⟨$A_{5,1}$, defineAttribute ("PertTask", "startDate", ""), changed("PertTask")⟩}*

agenda = *{ ⟨defineVariable("PertTask", "startDate", "undefined", "instance", "public", ""), $A_{5,1}$⟩, ⟨initializeAttribute("PertTask", "startDate", ""), $A_{5,1}$⟩, ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), ["duration", "endDate"], Description_2), $A_{26,1}$⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_{∞}⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_{∞}⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_{∞}⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_{∞}⟩ }.*

El próximo paso es satisfacer el objetivo *defineVariable*, para lo cual se puede de aplicar la primitiva P_5. Esto origina que una nueva tarea pendiente sea creada y que se agregue otro objetivo a la agenda. El nuevo objetivo, *optionalDefineMethod*(...), es resuelto aplicando P_{10}. Luego, el estado de las principales variables del planificador es:

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask",""), changed("PertTask")⟩, ⟨$P_{1,2}$, defineClass("PertTask", "Figure", Description_1)⟩ , ⟨$A_{8,1}$, defineFigureLayout("PertTask"), simple("PertTask")⟩, ⟨$P_{10,1}$, optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")⟩, ⟨$A_{26,1}$, relateAttributes("PertTask"), changed("PertTask"), useConstraints), ⟨$P_{18,1}$, selectMethod("HotDrawConstraint", "class", "instance creation", "", "plus:and:equal:")⟩, ⟨$P_{21,1}$, assignArguments("plus:and:equal:", "initializeAt:", "PertTask", ["startDate"], ["duration", "endDate"], "HotDrawConstraint plus:startDate and:duration equal:endDate")⟩, ⟨$A_{6,1}$, defineAtributeList("PertTask", ["startDate"])⟩, ⟨$A_{5,1}$, defineAttribute ("PertTask", "startDate", ""), changed("PertTask")⟩, ⟨$P_{5,1}$, defineVariable("PertTask", "startDate",*

"undefined", "instance", "public", "")⟩, ⟨$P_{10,2}$, optionalDefineMethod("PertTask", "startDate", "instance", "public", string("^ startDate"), [], "")⟩}

agenda = *{⟨initializeAttribute("PertTask", "startDate", ""), $A_{5,1}$⟩, ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), ["duration", "endDate"], Description_2), $A_{26,1}$⟩, ⟨relateAttributes ("PertTask", "PertTask"), A_∞⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_∞⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_∞⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_∞⟩ }*.

En tanto que la lista de tareas pendientes creada hasta el momento es

Tareas = *{DefineClass(Editor, "Editor", "The class that implements the editor itself", "Required"), DefineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor"), "Required"), DefineClass("PertTask", "Figure", "", "Required"), DefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "", "Optional"), DefineAttribute("PertTask", "startDate", "undefined", "instance", "public", "", "Required"), DefineMethod("PertTask", "startDate", "instance", "public", string("^ endDate"), [], "access method for the attribute", "Optional")}*

A continuación, el objetivo a satisfacer es ⟨*initializeAttribute ("PertTask", "startDate", ""*), $A_{5,1}$⟩, para lo cual se aplica la acción A_{16}. Con esto, dos nuevas condiciones son agregadas a la agenda: ⟨*selection(["edit", "noEdit"], "startDate", Ret), $A_{16,1}$*⟩, ⟨*specificInitialization(Ret, "PertTask", "startDate", ""), $A_{16,1}$*⟩.

El primero de estos dos objetivos es satisfecho utilizando la acción primitiva P_{19}, para lo cual debe ejecutarse la tarea de espera *AskSelection*. Suponiendo que el usuario no quiere editar este atributo, el retorno de esta tarea será "*noEdit*".

Para el segundo objetivo, ⟨*specificInitialization("noEdit", "PertTask", "startDate", ""), $A_{16,1}$*⟩, puede ser aplicada la acción A_{29}. En consecuencia, el estado de las variables del planificador será

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask","")⟩, changed("PertTask")⟩, ⟨$P_{1,2}$, defineClass("PertTask", "Figure", Description_1)⟩ , ⟨$A_{8,1}$, defineFigureLayout("PertTask"), simple("PertTask")⟩, ⟨$P_{10,1}$, optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")⟩, ⟨$A_{26,1}$, relateAttributes("PertTask"), changed("PertTask"), useConstraints ⟩, ⟨$P_{18,1}$, selectMethod("HotDrawConstraint", "class", "instance creation", "", "plus:and:equal:")⟩, ⟨$P_{21,1}$, assignArguments("plus:and:equal:", "initializeAt:", "PertTask", ["startDate"], ["duration", "endDate"], "HotDrawConstraint plus:startDate and:duration equal:endDate")⟩, ⟨$A_{6,1}$, defineAtributeList("PertTask", ["startDate"])⟩, ⟨$A_{5,1}$, defineAttribute ("PertTask", "startDate", ""), changed("PertTask")⟩, ⟨$P_{5,1}$, defineVariable("PertTask", "startDate", "undefined", "instance", "public", "")⟩, ⟨$P_{10,2}$, optionalDefineMethod("PertTask", "startDate", "instance", "public", string("^ startDate"), [], "")⟩, ⟨$A_{16,1}$, initializeAttribute("PertTask", "startDate", "")⟩, ⟨$P_{19,1}$, selection(["edit", "noEdit"], "startDate", "notEdit")⟩, ⟨$A_{29,1}$, specificInitialization("noEdit", "PertTask", "startDate", ""), changed("PertTask")⟩ },*

agenda = *{ ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string(" startDate := HotDrawVariable with:0 owner: self"), [], ""), $A_{29,1}$⟩, ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint",*

MessageString), ["duration", "endDate"], Description_2), $A_{26,1}$*⟩, ⟨relateAttributes ("PertTask", "PertTask"),* A_∞*⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"),* A_∞*⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"),* A_∞*⟩, ⟨makeRelationship ("PertTask", "PertTask"),* A_∞*⟩ }.*

El siguiente objetivo, *updateMethod("PertTAsk, "initializeAt:", ...)*, causa la aplicación de la primitiva P_{11}. Como consecuencia, es creada una nueva tarea pendiente, *UpdateMethod.* El siguiente objetivo es de la misma forma, por lo que será también resuelto aplicando la primitiva P_{11}. En este momento, la lista de tareas pendientes, con un formato que facilita su lectura, es

Tareas = *{*

- *DefineClass(Editor, "Editor", "The class that implements the editor itself", "Required"),*
- *DefineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor"), "Required"),*
- *DefineClass("PertTask", "Figure", "", "Required"),*
- *DefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "", "Optional"),*
- *DefineAttribute("PertTask", "startDate", "undefined", "instance", "public", "", "Required"),*
- *DefineMethod("PertTask", "startDate", "instance", "public", string("^ endDate"), [], "access method for the attribute", "Optional"),*
- *UpdateMethod("PertTask", "initializeAt:", "instance", "protected", string(" startDate:= HotDrawVariable with:0 owner: self"), [], "", "Required"),*
- *UpdateMethod("PertTask"; "initializeAt:", "instance", "protected", string("HotDrawConstraint plus:startDate and:duration equal:endDate"), ["duration", "endDate"], "", "Required")}.*

Aquí debe notarse que las dos últimas tareas tratan sobre la modificación de un mismo método, el método *initializeAt:* de la clase *PertTask*. Estas tareas serán mantenidas por separado hasta tanto se finalice el proceso de planificación. Sin embargo, antes de presentar la lista de tareas al usuario, el Administrador de Tareas comprueba si existen este tipo de situaciones y presenta las tareas de forma unificada. Es decir, que el usuario sólo verá una tarea de modificación del mencionado método.

El siguiente elemento de la agenda que debe ser considerado es ⟨*relateAttributes ("PertTask", "PertTask"),* A_∞⟩. La acción que se utiliza para satisfacer este objetivo es A_{27}. Esta acción agrega varios objetivos a la agenda: *selectMethod("HotDrawConstraint", "class", "instance creation", Description, Message), assignArguments(Message, "initializeAt:", "PertTask", Attributes, LocalVars, MessageString), defineAttributeList ("PertTask", Attributes), updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), LocalVars, Description), updateMethod("PertTask", "addDest:", "instance", "public", string(""), [], ""), updateMethod("PertTask", "addSource:", "instance", "public", string(""), [], "").*

De esta forma, el próximo paso será satisfacer el objetivo ⟨*selectMethod("HotDrawConstraint", "class", "instance creation", Description, Message),* $A_{27,1}$⟩, para lo cual se utiliza la acción primitiva P_{18}. Esta acción provocará que se ejecute una tarea de espera, que dará como resultado el método a utilizar para crear la restricción. Continuando con el ejemplo de creación del editor *PERT*, el método elegido para relacionar fechas de distintas tareas será *maximize:with:default:* (ver el código del método de la Figura 4.1).

En este momento, el estado de las variables del planificador será el siguiente:

A = *{⟨$A_{1,1}$, $required⟩, ⟨$A_{7,1}$, defineEditor, beingDefined(Editor), changed("PertTask")⟩, ⟨$P_{1,1}$, defineClass(Editor, "Editor", "The class that implements the editor itself")⟩, ⟨$P_{9,1}$, defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor")⟩, ⟨$A_{30,1}$, (useFigure("PertTask","")), changed("PertTask")⟩, ⟨$P_{1,2}$, defineClass("PertTask", "Figure", Description_1)⟩ , ⟨$A_{8,1}$, defineFigureLayout("PertTask"), simple("PertTask")⟩, ⟨$P_{10,1}$, optionalDefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "")⟩, ⟨$A_{26,1}$, relateAttributes("PertTask"), changed("PertTask"), useConstraints ⟩, ⟨$P_{18,1}$, selectMethod("HotDrawConstraint", "class", "instance creation", "", "plus:and:equal:")⟩, ⟨$P_{21,1}$, assignArguments("plus:and:equal:", "initializeAt:", "PertTask", ["startDate"], ["duration", "endDate"], "HotDrawConstraint plus:startDate and:duration equal:endDate")⟩, ⟨$A_{6,1}$, defineAtributeList("PertTask", ["startDate"])⟩, ⟨$A_{5,1}$, defineAttribute ("PertTask", "startDate", ""), changed("PertTask")⟩, ⟨$P_{5,1}$, defineVariable("PertTask", "startDate", "undefined", "instance", "public", "")⟩, ⟨$P_{10,2}$, optionalDefineMethod("PertTask", "startDate", "instance", "public", string("^ startDate"), [], "")⟩, ⟨$A_{16,1}$, initializeAttribute("PertTask", "startDate", "")⟩, ⟨$P_{19,1}$, selection(["edit", "noEdit"], "startDate", "notEdit")⟩, ⟨$A_{29,1}$, specificInitialization("noEdit", "PertTask", "startDate", "")⟩, ⟨$P_{11,1}$, updateMethod("PertTask", "initializeAt:", "instance", "protected", string(" startDate := HotDrawVariable with:0 owner: self"), [], "")⟩, ⟨$P_{11,2}$, updateMethod"PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), ["duration", "endDate"], "")⟩, ⟨$A_{27,1}$, relateAttributes ("PertTask", "PertTask"), changed("PertTask"), changed("PertTask"), useConstraints ⟩, ⟨$P_{18,2}$, selectMethod("HotDrawConstraint", "class", "instance creation", "", "maximize:with:default:")⟩}*,

agenda = *{ ⟨assignArguments("maximize:with:default:", "initializeAt:", "PertTask", Attributes, LocalVars, MessageString), $A_{27,1}$⟩, ⟨defineAttributeList ("PertTask", Attributes), $A_{27,1}$⟩, ⟨updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), LocalVars, Description), $A_{27,1}$⟩, ⟨updateMethod("PertTask", "addDest:", "instance", "public", string(""), [], ""), $A_{27,1}$⟩, ⟨updateMethod("PertTask", "addSource:", "instance", "public", string(""), [], ""), $A_{27,1}$⟩, ⟨editAttribute ("Tool", "PertTask", "Duration"), A_{∞}⟩, ⟨editAttribute("Menu", "PertTask", "EndDate"), A_{∞}⟩, ⟨makeRelationship ("PertTask", "PertTask"), A_{∞}⟩ }*.

El siguiente objetivo a considerar es ⟨*assignArguments(...), $A_{27,1}$*⟩, para lo cual es posible aplicar la primitiva P_{21}. De acuerdo al ejemplo, el primer parámetro será el atributo *starDate*, mientras que los otros dos serán expresiones constantes: *Array new* (creación de un vector) y 0.

A continuación, el objetivo que ha de satisfacerse es ⟨*defineAttributeList ("PertTask", ["startDate"]), $A_{27,1}$*⟩. Como es posible ver en la última representación del estado de la variable *A*, esta condición ya es hecha verdadera por una acción aplicada con anterioridad, $A_{6,1}$. Debido a esto, únicamente es necesario agregar al plan información que indique que dicha acción también soporta la nueva condición (vínculos causales).

Los tres siguientes objetivos, ⟨*updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", "maximize:startDate with: Array new default:0"), [], Description), $A_{27,1}$*⟩, ⟨*updateMethod("PertTask", "addDest:", "instance", "public", string(""), [], ""), $A_{27,1}$*⟩, ⟨*updateMethod("PertTask", "addSource:", "instance", "public", string(""), [], ""), $A_{27,1}$*⟩, se resuelven con sucesivas aplicaciones de la acción primitiva P_{11}.

En este punto, la porción final del conjunto de acciones aplicadas, *A*, será la siguiente:

A = *{ ... ⟨$A_{27,1}$, relateAttributes ("PertTask", "PertTask"), changed("PertTask"), changed("PertTask"), useConstraints ⟩, ⟨$P_{18,2}$, selectMethod("HotDrawConstraint", "class", "instance creation", "", "maximize:with:default:") ⟩, ⟨$P_{21,2}$,*

assignArguments("maximize:with:default:", "initializeAt:", "PertTask", ["startDate"], [], "maximize:startDate with: Array new default:0")⟩, ⟨$P_{11,3}$, *updateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint", "maximize:startDate with: Array new default:0"), [], "")*⟩, ⟨$P_{11,4}$, *updateMethod("PertTask", "addDest:", "instance", "public", string(""), [], "")*⟩, ⟨$P_{11,5}$, *updateMethod("PertTask", "addSource:", "instance", "public", string(""), [], "")*⟩}.

Así mismo, la agenda tendrá el siguiente contenido:

agenda = { ⟨*editAttribute ("Tool", "PertTask", "Duration")*, A_∞⟩, ⟨*editAttribute("Menu", "PertTask", "endDate")*, A_∞⟩, ⟨*makeRelationship ("PertTask", "PertTask")*, A_∞⟩ }.

Finalmente, la lista de tareas a ser ejecutadas es, en este momento:

Tareas = {

- *DefineClass(Editor, "Editor", "The class that implements the editor itself", "Required"),*
- *DefineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [] "This method should be updated every time a tool is added to the editor"), "Required"),*
- *DefineClass("PertTask", "Figure", "", "Required"),*
- *DefineMethod("PertTask", "displayOn:", "instance", "public", string(""), [], "", "Optional"),*
- *DefineAttribute("PertTask", "startDate", "undefined", "instance", "public", "", "Required"),*
- *DefineMethod("PertTask", "startDate", "instance", "public", string("^ endDate"), [], "access method for the attribute", "Optional"),*
- *UpdateMethod("PertTask", "initializeAt:", "instance", "protected", string(" startDate := HotDrawVariable with:0 owner: self"), [], "", "Required"),*
- *UpdateMethod("PertTask", "initializeAt:", "instance", "protected", string("HotDrawConstraint maximize:startDate with: Array new default:0"), [], ""),*
- *UpdateMethod("PertTask", "addDest:", "instance", "public", string(""), [], "")*
- *UpdateMethod("PertTask", "addSource:", "instance", "public", string(""), [], "")}*

De acuerdo a estos valores, el próximo objetivo a considerar es ⟨*editAttribute ("Tool",...)*, A_∞⟩. Para resolverlo, es posible aplicar la acción A_{12}. Esta acción tiene, entre otras precondiciones, tres condiciones basadas en el operador *not*. Las dos primeras verifican que no haya sido aplicada otra acción que utilice un medio distinto para editar atributos. La tercera de estas condiciones comprueba si no existe una acción que tenga como efecto *simple("PertTask")*. Esta condición no se cumple porque efectivamente existe en el plan actual una acción que produzca ese efecto, $A_{8,1}$.

Como no existe otra acción que permita satisfacer el objetivo actual, el planificador debe aplicar *backtracking* sobre anteriores decisiones, en la búsqueda de un plan alternativo. La opción es retroceder hasta la aplicación de la acción A_8, utilizando en vez de ello la acción A_9. A partir de allí, el resto del plan es vuelto a construir. Aquí es posible comprobar otra de las características de la asistencia provista por la herramienta. Todas las tareas ejecutadas por el usuario para tomar decisiones durante el proceso de planificación son mantenidas, de forma tal de poder reutilizar esa información. Con esto se consigue que el usuario no deba, por ejemplo, elegir más de una vez el método que se utilizará para implementar determinada restricción entre atributos.

Siguiendo con el proceso de planificación, se produce una situación que ya ha sido explicada en §V.2.6.4. Esta situación se deriva del hecho de que en la lista de objetivos iniciales existen dos objetivos que, de acuerdo a la documentación del *framework*, son contradictorios. Así, no es posible construir un plan donde un atributo sea editado utilizando un *tool* y otro lo sea utilizando un menú. Como resultado, *PIT* retornará un plan de instanciación que no contemple la implementación del penúltimo objetivo, $\langle editAttribute(\text{"Menu"}, \text{"PertTask"}, \text{"endDate"}), A_\infty \rangle$. A partir de este punto, existen dos alternativas: el usuario puede decidir seguir adelante con el plan obtenido, implementando él por su cuenta la funcionalidad faltante, o puede reconsiderar la lista inicial de objetivos funcionales, modificando alguno de los objetivos conflictivos.

3. Ejecutando el Plan de Instanciación

Otro aspecto que debe ser comentado es la asistencia provista durante la ejecución del plan resultante. Una vez que se le presenta al usuario la lista de tareas que debería ejecutar para implementar la funcionalidad especificada, los Administradores de Tareas y de Consistencia interaccionan para controlar la actividad del usuario, asistiendo siempre que la documentación disponible lo permita.

Por ejemplo, si en el ejemplo de la sección anterior se hubiera utilizado la acción A_8 en el plan final, se le presentaría al usuario, entre otras tareas, una tarea opcional para implementar el método *displayOn:* de la clase *PertTask*. Sin embargo, existe una regla de consistencia que describe que el método *displayOn:* definido en la clase *Figure* (superclase de *PertTask*) es abstracto. Como consecuencia, toda subclase concreta de *Figure* debe redefinir dicho método. A raíz de esta regla de consistencia, se creará una tarea **obligatoria** para definir el método *displayOn:*, la cual tendrá prioridad sobre la tarea optativa existente con anterioridad.

En el capítulo siguiente se describe cómo son implementados estos mecanismos para complementar la asistencia provista a través del planificador.

VIII Entorno de Desarrollo HiFi

Para demostrar la viabilidad de las ideas propuestas en los capítulos anteriores, fue construido el prototipo de un entorno para asistir la instanciación de *frameworks*, llamado ***HiFi*** (*Helping in Framework Instantiation*). Este entorno, que ha sido implementando en *Smalltalk*, concretamente utilizando *VisualWorks* 2.0, soporta el desarrollo de aplicaciones basado en *frameworks* siguiendo el enfoque propuesto por *SmartBooks*. Para ello cumple dos funciones: en primer lugar, posibilita la documentación del *framework*, utilizando *UML* más las extensiones de *SmartBooks*, es decir, las reglas de funcionalidad y consistencia. Luego, en base a esta documentación, guía la labor de un usuario en la creación de una aplicación a partir del *framework*.

En este capítulo se describirá brevemente la funcionalidad provista por este sistema, así como su diseño e implementación.

1. Utilización de *HiFi*

El ambiente *HiFi* distingue entre dos tipos de usuarios: el diseñador del *framework* (o autor de la documentación) y el usuario del *framework* (programador de la nueva aplicación).

El diseñador puede describir el diseño utilizando técnicas normales de documentación (específicamente *UML*), patrones de diseño [GHJV94], ejemplos, etc. Para soportar *SmartBooks*, además de esa documentación, el entorno permite al diseñador:

- describir la funcionalidad provista por el *framework*;
- describir cómo esta funcionalidad es implementada por los diferentes componentes del *framework;* y
- proveer reglas que limiten las posibles especializaciones del *framework*.

Los dos últimos puntos son necesarios, sobre todo, para las actividades de planificación.

Utilizando esta documentación, el sistema provee información que guía al usuario del *framework* en el proceso de creación de la aplicación. Esta guía está enfocada en la funcionalidad requerida de la nueva aplicación, por lo que el usuario es orientado a definir qué es lo que debe hacer la aplicación. En base a esta información sobre la funcionalidad, es elaborado un plan para la instanciación del *framework* y se le indica al usuario que lleve a cabo las tareas listadas en este plan. Cada tarea generada es vinculada con la documentación asociada. De esta forma, cuando está ejecutando una tarea dada, el usuario puede navegar a la información relacionada, como por ejemplo jerarquías de clases, diagramas de estado, diagramas de colaboración, patrones de diseño, ejemplos, reglas, código asociado, etc.

Como ya se ha destacado, para que la asistencia sea de mayor utilidad, el soporte a la ejecución del plan no debe ser rígido: debe permitir al usuario revisar la funcionalidad requerida en cualquier momento del proceso de instanciación. Es decir, no se requiere que el usuario describa toda la funcionalidad de la aplicación en un solo paso. El proceso de describir esta funcionalidad, generar el plan de instanciación y ejecutar las tareas correspondientes responde, en cierta medida, a un modelo en espiral. El usuario provee una primera descripción de la funcionalidad requerida, el plan es generado y el usuario comienza a trabajar de acuerdo a este plan. Ya sea cuando el plan ha sido completamente ejecutado (y la aplicación implementada) o mientras se está ejecutando el plan, el usuario puede retroceder al paso de descripción de funcionalidad, cambiar los requisitos y comenzar a trabajar con el plan resultante de la nueva funcionalidad. Para esto, uno de los módulos del sistema, el Administrador de Tareas, mantiene el registro de las tareas ejecutadas y si el nuevo plan incluye algunas de las tareas que el usuario

ya ha ejecutado como parte del plan anterior, el resultado de estas tareas será reutilizado siempre que sea posible.

El entorno necesita flexibilidad adicional cuando el usuario trata de utilizar el *framework* para implementar funcionalidad no anticipada por el diseñador. Como se vio, en este caso sólo un plan parcial puede ser generado, o incluso en algunos casos no se podrá generar plan alguno. Aún en esas situaciones, el ambiente puede proveer cierta ayuda a partir de la información ofrecida por las reglas de consistencia, especialmente aquellas derivadas de los patrones de diseño aplicadas en el diseño.

En las próximas secciones se describe el diseño e implementación de este sistema, ampliando también la descripción de su funcionalidad.

2. Diseño e Implementación de *HiFi*

La arquitectura del sistema está formada por ocho componentes principales (Figura 8.1):

Interfaz a Usuario: está implementada en base al concepto de libro electrónico. Es decir, la documentación del *framework* es organizada en forma de libro, donde las porciones de información relacionadas están asociadas por medio de vínculos de hipertexto.

Editor de Documentación: esta herramienta es utilizada principalmente por el diseñador del *framework* para escribir la documentación del mismo. Además, es utilizada por el usuario del *framework* cuando crea nuevos componentes o le agrega funcionalidad a los ya existentes.

Asistente de Funcionalidad: utilizando la información sobre la funcionalidad provista por el *framework*, este módulo ayuda al usuario a describir la funcionalidad requerida para la nueva aplicación que está siendo desarrollada.

Generador de Acciones de Instanciación: este componente, basado en la documentación del *framework*, genera las acciones de instanciación, es decir, las acciones, precondiciones y efectos que son utilizados para la generación de los planes de instanciación.

Planificador: este módulo implementa el algoritmo de planificación *PIT* descrito en el capítulo V. En base a los requisitos funcionales y la información producida a partir de las reglas, genera un plan parcial para crear la aplicación a partir del *framework*.

Generador de Reglas Prolog: es el encargado de generar la información necesaria para el Administrador de Consistencia. Esta información es generada a partir de las reglas de consistencia (capítulo VI) y es codificada en forma de reglas *Prolog*.

Administrador de Tareas: controla la actividad del programador de la aplicación. Este módulo administra la lista de tareas pendientes e informa al Administrador de Consistencia cuando una tarea ha sido finalizada, de forma que éste último pueda comprobar si alguna regla de consistencia debe ser aplicada. Además, el Administrador de Tareas interacciona con la editor de documentación cuando, como consecuencia de la ejecución de una tarea, componentes de software nuevos o modificados deben ser documentados. Finalmente, es el responsable de permitir la reutilización de tareas ya ejecutadas cuando el plan de instanciación es regenerado.

Administrador de Consistencia: trabaja de forma muy relacionada con el Administrador de Tareas. Este módulo tiene la responsabilidad de verificar si las acciones del usuario producen alguna configuración del software inconsistente con la descripción del *framework*. En este caso, debe crear y pasar al Administrador de Tareas las tareas que el usuario debería que ejecutar para retornar el software a un estado consistente.

El objetivo de la Figura 8.1 es ofrecer una idea general de las partes que componen el sistema y no es una descripción detallada de su arquitectura. En particular, no todos los componentes de *HiFi* son mostrados en esta figura. Además, si bien el usuario interacciona siempre a través de los libros electrónicos, el diagrama muestra cuales son los componentes encargados de proporcionar y tomar información a y del usuario, utilizando para ellos las líneas

a trazos. Las líneas sólidas, por su parte, representan algunas de las relaciones entre los componentes del sistema. Específicamente, estas representan las relaciones de flujo de datos, describiendo qué componente produce los datos que son utilizado por otro(s).

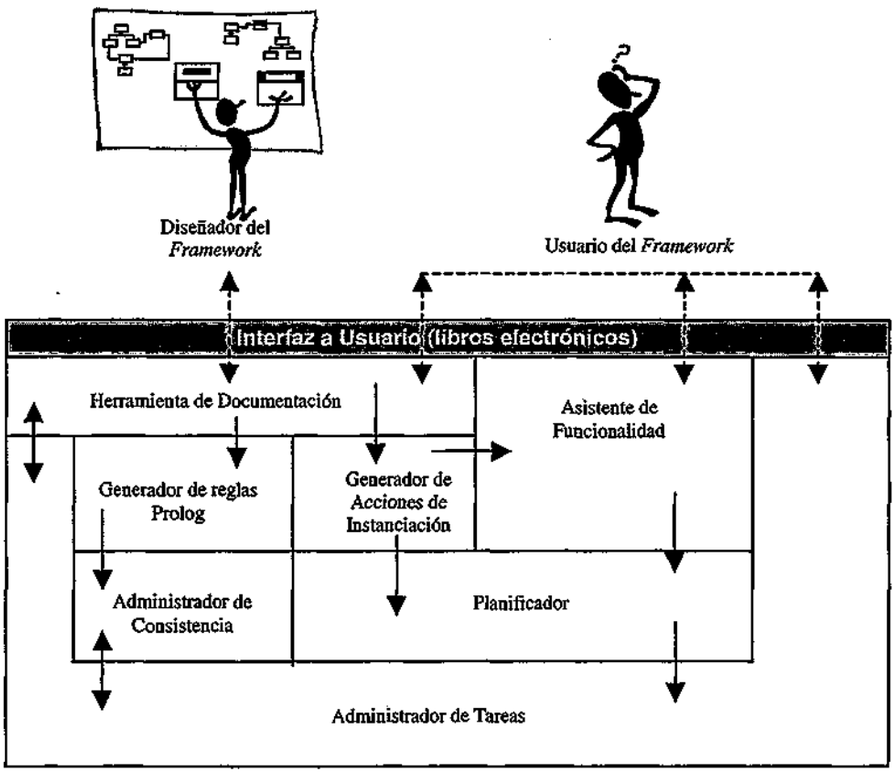


Figura 8.1. Arquitectura parcial de *HiFi*

Existe un componente adicional no mostrado hasta el momento: un **repositorio central**, a través del cual se realiza gran parte del paso de datos mostrado en la Figura 8.1. Este componente y las otras interacciones existentes entre los módulos se describen con mayor detalle a continuación.

2.1.1. Estilo general de la Arquitectura del Sistema

Para un estudio más detallado de la arquitectura del sistema, se la describirá en términos de estilos arquitectónicos [SG96]. En este sentido, el sistema, de forma general, responde a las características de un sistema basado en repositorio. En este estilo de sistemas, los componentes son divididos en dos grandes grupos: una estructura central de datos representa el estado actual y una colección de componentes independientes operan sobre el repositorio central. Una representación de este tipo de sistemas es mostrada en la Figura 8.2 (adaptada de [SG96]).

El diagrama de la Figura 8.2 no muestra una representación explícita del componente a cargo del control del sistema, el cual puede estar distribuido en los otros componentes. Esta estructura, que ha sido utilizada frecuentemente para implementar entornos de desarrollo de software, fue escogida porque permite que una colección de herramientas, con bajo acoplamiento entre sí, puedan compartir datos comunes. De esta forma, es posible experimentar con distintas formas de asistir al usuario, a partir de la representación de la documentación y el código del *framework*. En el caso de *HiFi*, las *fuentes de conocimiento* son los distintos módulos mostrados en la Figura 8.1, como por ejemplo el Administrador de Tareas y el Generador de Acciones de Instanciación.

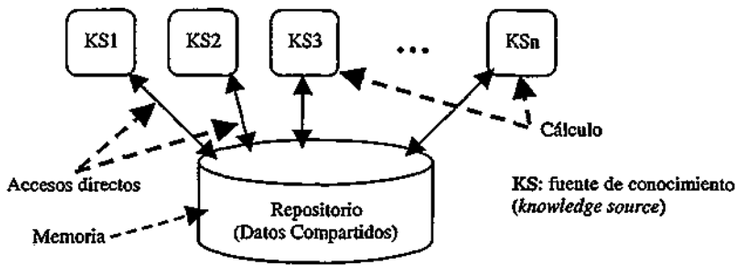


Figura 8.2. Estructura de un Sistema basado en Repositorio

Si bien el estilo de sistemas basados en un repositorio central describe la estructura general de *HiFi*, la interacción entre los distintos subconjuntos de componentes puede ser descrita a través de otros estilos o patrones arquitectónicos. En las siguientes secciones se describen los distintos componentes y se indican otros estilos arquitectónicos que pueden ser encontrados en el diseño del sistema.

El prototipo fue implementado principalmente en el lenguaje *Smalltalk* [Gol83, Par94], estando algunas porciones implementadas en *Prolog*. Para esto se utilizó una biblioteca que provee la capacidad de implementar en *Prolog* métodos de clases *Smalltalk* [Ama97]. De esta forma, cuando se envía un mensaje a un objeto de estas clases, el código ejecutado como consecuencia puede ser *Prolog*.

2.2. Repositorio Central

Además de posibilitar el intercambio de información entre las distintas componentes del sistema, el repositorio central de *HiFi* está encargado de la persistencia de la información almacenada. Esta información posee distintas representaciones. Por ejemplo, el código es representado con una estructura de objetos, pero también se mantienen representaciones paralelas en lenguaje *Prolog* y *Smalltalk*. En general, el repositorio de *HiFi* contiene la siguiente información:

- Documentación de diseño: son los diagramas prescritos por la notación *UML*, la información de patrones de diseño, la representación de las reglas funcionales y de consistencia y descripción de funcionalidad. Las reglas son almacenadas en dos formatos: su representación a través de una estructura de objetos (la descripción construida por el usuario) y una representación en *Prolog*, utilizada por el Administrador de Consistencia para disparar la creación de tareas necesarias para reparar inconsistencias.
- Representación del plan de instanciación, con toda la información generada por el Planificador y no sólo la lista de tareas.
- Representación del proceso de instanciación, es decir, información del estado de aplicación del plan de instanciación. Entre otras cosas, comprende información sobre las tareas que fueron ejecutas y las que restan ejecutar.
- Representación del código del *framework*, de la aplicación que está siendo desarrollada y de otras aplicaciones construidas anteriormente a partir del *framework*.

La Figura 8.3 muestra las clases básicas que implementan este repositorio[1]. Los componentes almacenados o son simples (subclases de *EntityComponent*), compuestos

[1] Para facilitar su comprensión, las clases utilizadas para representar la información en el repositorio serán mostradas de forma gradual, a medida que se expliquen los componentes que las manipulan. En los diagramas mostrados en este capítulo, los rectángulos sombrados representan clases

(subclases de *ContextComponent*) o son relaciones entre otros componentes (subclases de *LinkComponent*). Esta clasificación responde a un modelo desarrollado para almacenar la información producida durante el proceso de desarrollo de software, llamado *ProMeto* [Ort95].

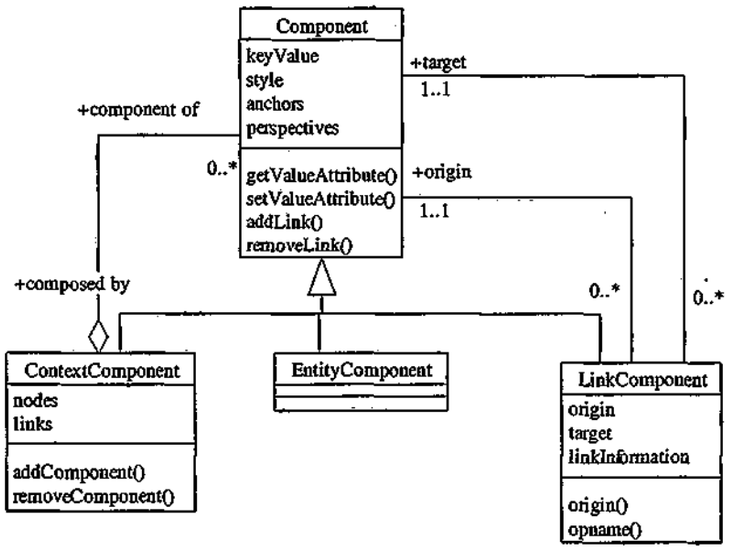


Figura 8.3. Representación de las clases básicas que conforman el repositorio

En aquellos casos en que la misma información se almacena con distintas representaciones, es el repositorio el encargado de mantener la consistencia entre ellas. Por ejemplo, cuando alguna herramienta modifica el código *Smalltalk*, informa de ello al repositorio, para que éste actualice la información correspondiente.

Aunque las herramientas ven al repositorio como un depósito único donde guardar y recuperar información, en realidad esta información se halla distribuida en distintos lugares. Por ejemplo, la persistencia de la información es lograda almacenándola en ficheros, aunque el acceso a estos es transparente para las herramientas que utilizan el repositorio. Otro ejemplo es el código *Smalltalk*, que desde el punto de vista de las herramientas es almacenado en el repositorio. Sin embargo, este código es mantenido por los mecanismos normales del entorno *Smalltalk* (a través de la imagen de memoria), lo que posibilita que las herramientas provistas por el entorno, como por ejemplo el compilador y los distintos inspectores, puedan ser utilizadas para manipularlo y ejecutarlo. El repositorio no sólo mantiene las representaciones consistentes, sino que permite a las herramientas operar cómo si el código estuviera sólo en el repositorio. En términos de patrones de diseño, el repositorio actúa como un *proxy* [GHJV94] del código *Smalltalk* (ver, por ejemplo, el atributo *realClass* de la clase *RealClassRepresentation* mostrada en la Figura 8.6).

2.3. Interfaz a Usuario

El módulo de interfaz al usuario utiliza el concepto de libro electrónico para organizar y presentar la documentación del *framework* al usuario. Esta documentación es divida en libros, cada uno de los cuales está compuesto por capítulos. El diseñador puede utilizar esta estructura para organizar la información como lo considere conveniente. De todas formas, siguiendo la propuesta de Johnson [Joh92] y Lajoie y Kelly [LK94], en cada libro se incluye una sección de

que no se relacionan directamente con el módulo que se está describiendo, pero que son dibujadas para mostrar la ubicación en la jerarquía de las otras clases

introducción, con referencias a las distintas porciones de la documentación incluida en ese libro. La Figura 8.4 muestra un ejemplo de la interfaz mostrada al usuario.

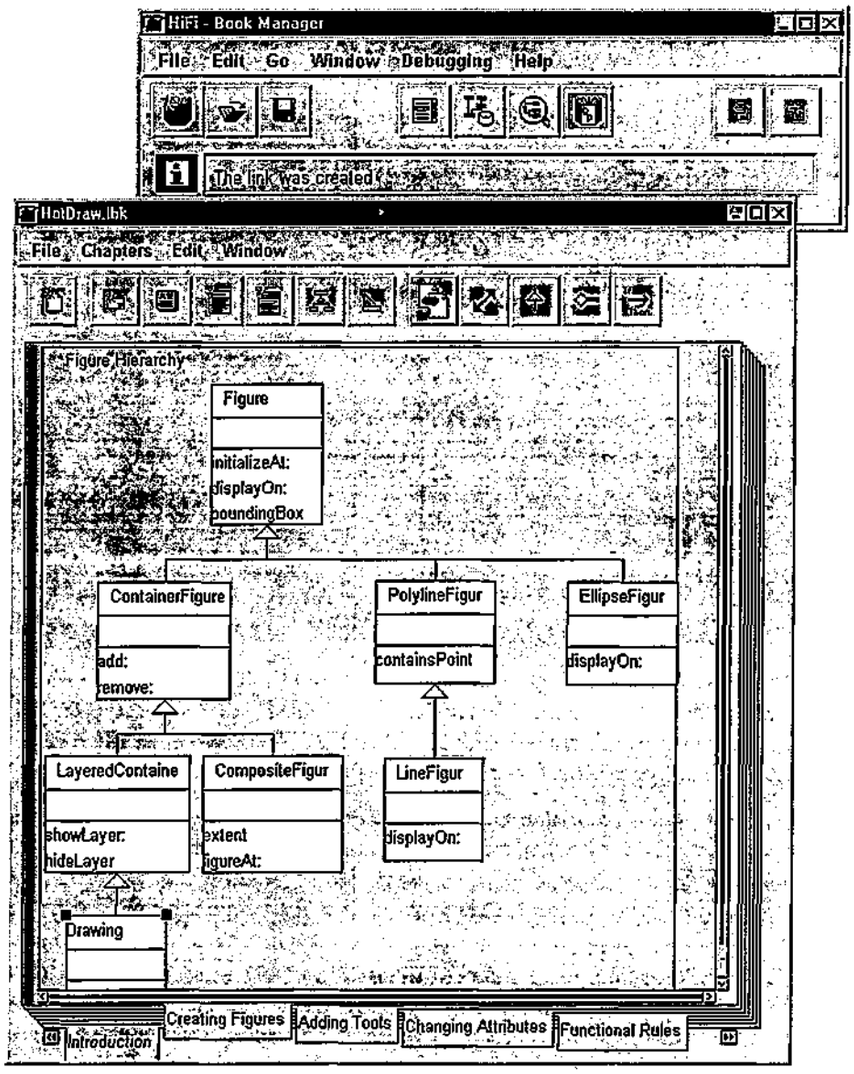


Figura 8.4. Interfaz del editor / inspector de documentación

Una de las principales características de los libros electrónicos es permitir relacionar las distintas porciones de documentación a través de vínculos (*links*) de hipertexto. De esta forma, el usuario puede, por ejemplo, navegar desde la definición de un componente del *framework* al código asociado, a ejemplos de especialización y a las reglas de consistencia en las que está involucrado.

También la información dinámica generada por el entorno es presentada a través de los libros electrónicos. Por ejemplo, para visualizar la información de las tareas ejecutadas y pendientes, se inserta una vista del Administrador de Tareas en un capítulo de un libro y luego se puede interaccionar a través de ella. Más aún, pueden crearse tantas de estas vistas como se desee, y el módulo de interfaz se encargará de mantenerlas actualizadas a todas. Una de las ventajas de incorporar el administrador de tareas dentro de los libros es mantener la coherencia de la presentación que se muestra al usuario. Además, de esta forma es posible utilizar los mecanismos de navegación para ver la información relacionada con las tareas, como por ejemplo las reglas y los requisitos que dieron origen a una tarea dada.

La interacción de este módulo con el resto del sistema responde a una variante del patrón arquitectónico MVC (*Model-View-Controller*) [Par94, BMR+96]. Este patrón es utilizado para el diseño de sistemas interactivos y tiene como objetivo separar la funcionalidad de la aplicación de su presentación al usuario. En este caso, los libros electrónicos implementan la vista y el controlador prescritos por este patrón, mientras que el repositorio y las herramientas actúan como modelos. El mantenimiento de consistencia entre el modelo y su presentación es hecho a través de anuncios indirectos de cambio. Esto permite implementar el componente modelo independientemente de las diferentes presentaciones que vayan a ser utilizadas para visualizarlo y manipularlo.

Además, el diseño interno del módulo también puede ser descrito en términos de otra aplicación del mismo patrón arquitectónico, en este caso la funcionalidad de las clases utilizadas para almacenar la información (comprendidas dentro del repositorio) y aquellas que implementan las vistas, es decir, que efectivamente dibujan en la pantalla. En la Figura 8.5 se representan las clases que implementan el "modelo" en esta aplicación del patrón; de esta clases heredan todas aquellas que representan objetos que deben mostrarse en los libros electrónicos.

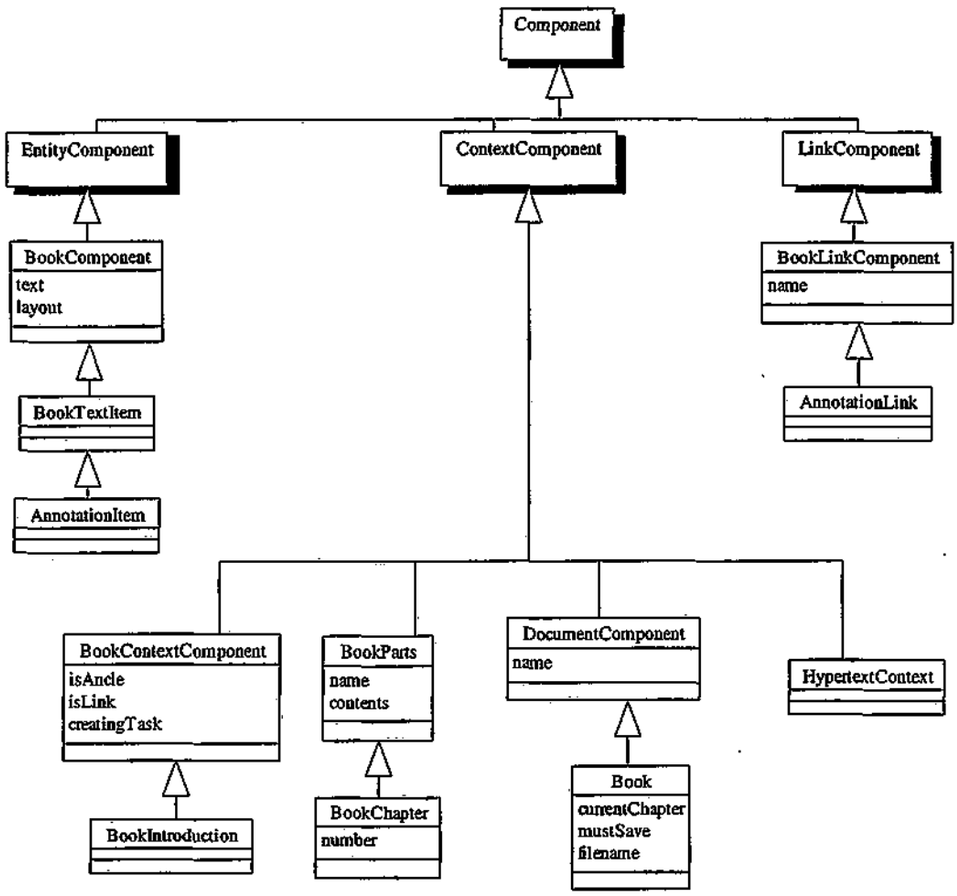


Figura 8.5. Jerarquía de clases utilizadas para representar los libros electrónicos en el repositorio. Las clases sombreadas pertenecen a la estructura básica del repositorio

2.4. Editor de Documentación

La documentación de un *framework* utilizando *HiFi* generalmente consiste en una descripción tradicional de diseño, realizada a través de la notación *UML*, documentación de los

patrones de diseños aplicados [GHJV94], ejemplos y vínculos al código relacionado. También es posible vincular la documentación con ejemplos de otras aplicaciones construidas utilizando el *framework.*

Los aspectos más destacable de la documentación de un *framework* en *HiFi* son la descripción de la funcionalidad ofrecida por el *framework* (tanto a través de reglas funcionales gráficas como acciones de instanciación) y los estados de consistencia que deben ser mantenidos (reglas de consistencia). Las acciones de instanciación han sido explicadas en el capítulo V, en tanto que las reglas son descritas en el capítulo VI.

Gran parte de las clases del repositorio son usadas directamente por la Editor de Documentación, debido a que es a través de esta herramienta que toda la documentación es producida. El modelamiento del software se hizo de acuerdo al meta modelo propuesto para *UML* [Rat97], el cual puede ser visto en el Anexo C.

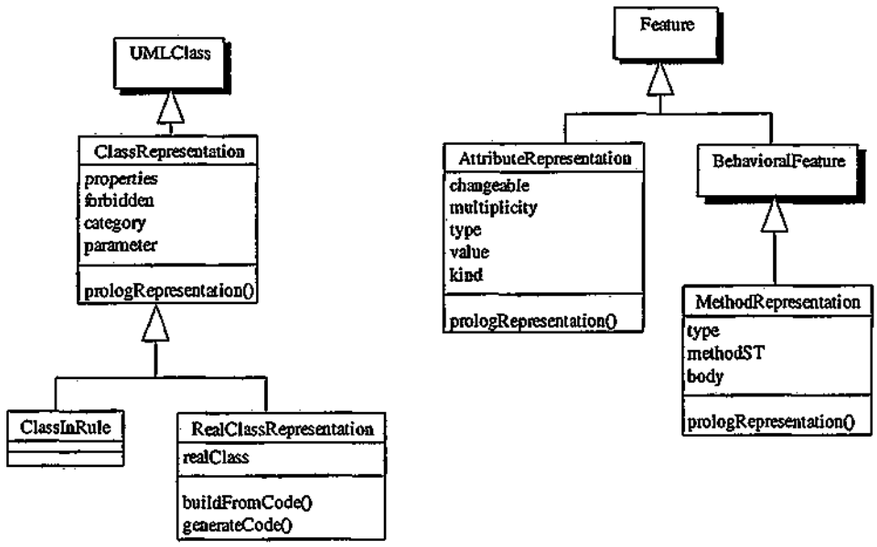


Figura 8.6. Estructura utilizada para representar las clases, atributos y métodos.

En la Figura 8.6 se muestra, por ejemplo, la estructura de representación de las clases, sus atributos y operaciones. Las clases *UMLClass*, *Feature* y *BehavioralFeature* corresponden a las clases prescritas por el meta modelo de *UML*. En la Figura 8.7, por su parte, es mostrada la representación de las relaciones entre clases, mientras que en la Figura 8.8 se muestran algunas de las clases utilizadas para representar los *frameworks* y los distintos diagramas.

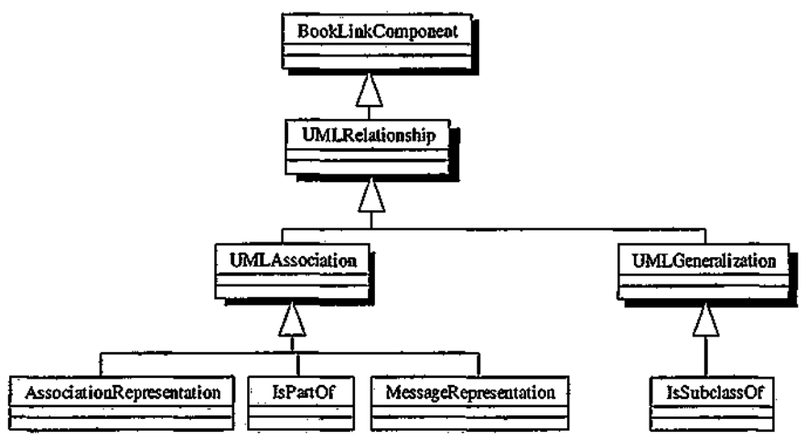


Figura 8.7. Representación de las relaciones entre clases

En el momento de documentar su *framework*, el diseñador dispone de una biblioteca de libros ya existentes, los cuales pueden ser utilizados en la documentación del nuevo *framework*. De esta forma, por ejemplo, la documentación de un *framework* puede hacer referencia a la documentación de un subsistema del mismo, a *frameworks* similares que puedan brindar información adicional, otras versiones (tal vez en diferente lenguaje de programación) del mismo *framework*, etc.

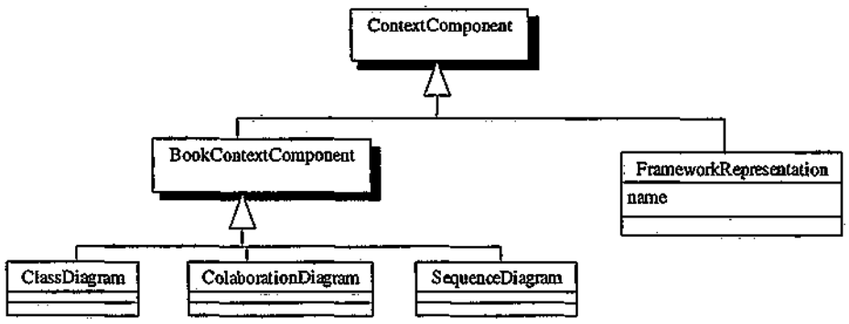


Figura 8.8. Representación de los *frameworks* y algunos de los diagramas

En particular, existe una colección de libros dedicados a la documentación de patrones de diseño; la Figura 8.9 muestra una vista del libro que contiene la documentación de los patrones. En la definición de patrones, los nombres entre corchetes angulares ("<" y ">") no representan componentes genéricos ni variables comunes de *TOON*. Estos nombres representan los papeles que pueden desempeñar los componentes que toman parte de cada aplicación del patrón. Es decir, cuando el diseñador describa una aplicación del patrón *Strategy* en su diseño, deberá especificar qué clase cumple el papel de *Strategy* y qué método de esa clase el papel de *algorithm*. En la Figura 8.9 la definición del patrón es asociada con una regla de consistencia, cuya utilización es explicada en §VIII.2.6.

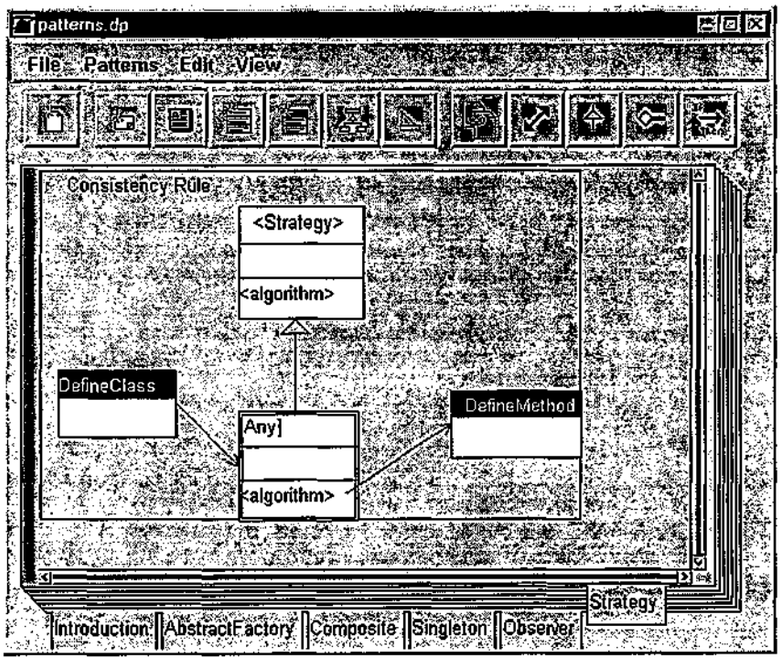


Figura 8.9. Documentación de patrones de diseño en *HiFi*.

Algunas de las clases utilizadas para representar la información de diseño han sido especializadas para permitir la representación de los patrones de diseño. Estas clases específicas son mostradas en la Figura 8.10.

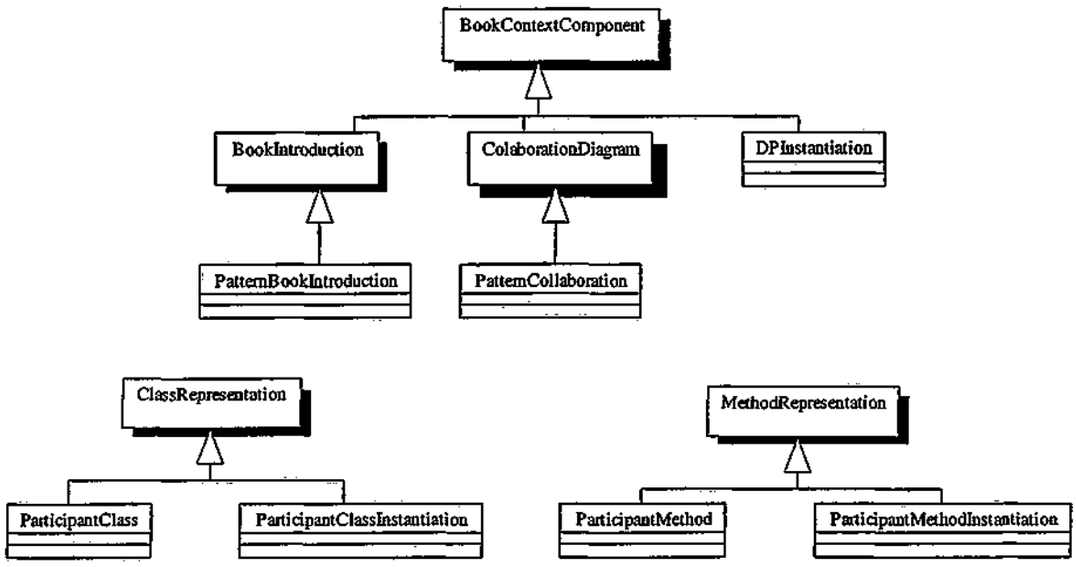


Figura 8.10. Especializaciones para la representación de patrones de diseño

Para facilitar la producción de la documentación, la herramienta permite utilizar notaciones gráficas siempre que sea posible. Sin embargo, como se explicó en el capítulo anterior, no toda la información necesaria para los mecanismos de asistencia puede ser expresadas a través de *TOON* (ver §VI.3). Parte de esta información adicional puede ser descrita a través del editor de funcionalidad, mostrado en en la Figura 8.11. Este editor, que forma parte del Editor de Documentación, es capaz de producir acciones de instanciación simples. Aquella funcionalidad que tampoco puede ser descrita utilizando el editor de funcionalidad debe ser especificada directamente mediante acciones de instanciación, en la forma descrita en el capítulo V.

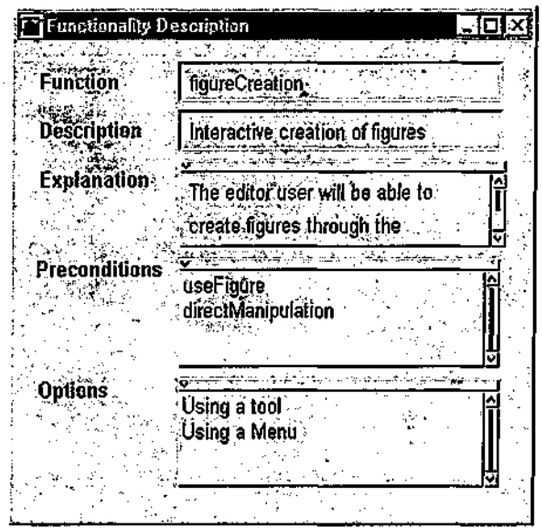


Figura 8.11. Interfaz del Editor de Funcionalidad

2.5. Generador de Acciones de Instanciación

A partir de la documentación provista por el diseñador, el Generador de Acciones de Instanciación produce las acciones de instanciación (ver el capítulo V para una descripción detallada de estas acciones), las cuales serán utilizadas por el Planificador y el Asistente de Funcionalidad. En el estado actual de desarrollo de *HiFi*, este módulo utiliza dos tipos de entradas:

- Descripciones gráficas de reglas de consistencia y funcionales las cuales son traducidas en acciones de instanciación
- Descripciones textuales de acciones de instanciación: estas descripciones pueden ser generadas por el editor de funcionalidad de la Figura 8.11 o bien acciones de instanciación escritas directamente por el diseñador, siguiendo el formato descrito para las mismas en el capítulo V. La descripciones textuales no son modificadas, sino que son directamente enviadas al planificador.

2.6. Generador de reglas *Prolog*

Este módulo genera código *Prolog* a partir de las reglas de consistencia. El código *Prolog* generado es utilizado por el Administrador de Consistencia para supervisar la actividad del usuario, tal como se explica en §VIII.2.9. Por ejemplo, volviendo a considerar la regla de consistencia relacionada con un método abstracto *displayOn:* (§VI.4.1, Figura 6.11), la Figura 8.12 muestra el código *Prolog* producido a partir de esa regla. Debe notarse que el antecedente de la regla es traducido como un evento que debe producirse para que la regla sea evaluada. En §VI.2.10 se explica cómo estos eventos son generados.

```
displayOnMethod-1 (C0,M2) :- hasMethod (C0,M2),
        nameOf(M2,'displayOn:').

displayOnMethod-5 (M4,M6) :- calls (M4,M6).

displayOnMethod-3 (C0,M4) :- hasClassMethod (C0,M4),
        nameOf(M4,'new'), displayOnMethod-5 (M4,M6).

displayOnMethod-11 (C0,C10) :- isSubclass(C0,C10).

displayOnMethod-9 (C8,C10) :- nameOf(C10,'Figure'),
        \=(C8,C10), displayOnMethod-11 (C0,C10).

displayOnMethod-7 (C0,C8) :- isSubclass(C0,C8),
        displayOnMethod-9 (C8,C10), hasMethod(C8,'displayOn:').

displayOnMethod-14 (C0,C8) :- isSubclass(C0,C8).

displayOnMethod-13 (C8,C10) :- \=(C8,C10),
        displayOnMethod-14 (C0,C8), hasMethod(C8,'displayOn:').

displayOnMethod-12 (C0,C10) :- nameOf(C10,'Figure'),
        isSubclass(C0,C10), not(displayOnMethod-13 (C8,C10)).

displayOnMethod-15 (M2) :- nameOf(M2,'displayOn:').

displayOnMethod (C0) :- not(displayOnMethod-1 (C0,M2)),
        displayOnMethod-3 (C0,M4), not(displayOnMethod-7 (C0,C8)),
        displayOnMethod-12 (C0,C10),
        =(X16,'displayOn:'), nameOf(C0,X23), =(X18,'instance'),
        send ({DefineMethod}, newFromRule:args:post:, [displayOnMethod, [required, 'displayOn:', X17, X18, X19,
        X20, X21 ],'displayOnMethod-15 (M2)'] ), fail.

displayOnMethod (C0) :- true.

event('DefineClass',C0 ) :- displayOnMethod (C0).

event('DefineMethod',M6 ) :- displayOnMethod (C0).
```

Figura 8.12. Representación *Prolog* de la regla de consistencia del método abstracto *displayOn:*

Debido a que el código mostrado es generado automáticamente, es posible ver que no está optimizado; en particular, existen comprobaciones redundantes. Mientras que esta redundancia no cambia la semántica de la regla, sí tiene efectos negativos sobre la eficiencia del sistema; este aspecto, sin embargo, no ha sido tenido en cuenta en el desarrollo de *HiFi*, por no ser uno de sus objetivos.

En los capítulos IV y VI se vio que estas reglas son especialmente útiles cuando se combinan con situaciones generales, que se encuentran frecuentemente en distintos *frameworks*. La regla descrita para el método *displayOn:* puede ser aplicada a cualquier método abstracto. De esta forma, resulta útil definir las reglas de consistencia asociadas con los métodos abstractos y luego permitir que el diseñador describa cuáles son los método abstractos de su *framework*.

Dentro de estas descripciones de casos generales es posible catalogar a los patrones de diseño. Como ha sido visto en §VIII.2.4, durante el proceso de documentación el diseñador puede describir qué patrones fueron aplicados en el diseño del *framework*, pudiendo asociar cada aplicación de un patrón con los correspondientes componentes del *framework*. Además de explicar el diseño de los componentes, algunos patrones tienen asociadas reglas de consistencia, que describen propiedades del diseño que deben ser respetadas por aquellos componentes que implementen el patrón.

Para analizar un ejemplo de regla de consistencia asociado a un patrón de diseño, consideramos el patrón *Singleton*. Este patrón describe el diseño e implementación de una clase de la cual existirá sólo una instancia. En consecuencia, es posible inferir que la creación de instancias de esa clase es un error potencial. Por lo tanto, si se especifica que una clase implementa el patrón *Singleton*, el sistema puede marcar una advertencia (*warning*) cada vez que una mensaje de creación sea enviado a la clase o alguna de sus subclases. La representación *TOON* de esta regla es mostrada en la Figura 8.13.

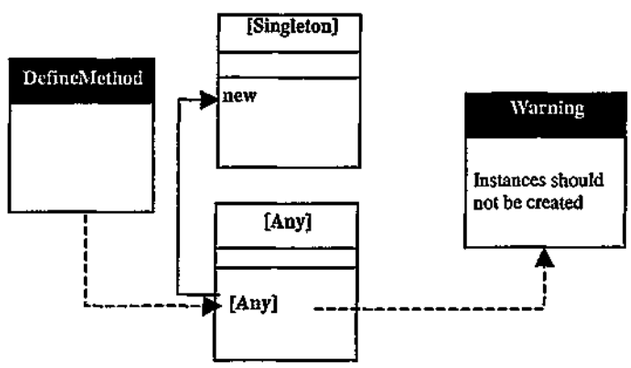


Figura 8.13. Regla de consistencia asociada con el patrón de diseño *Singleton*

La clase *Singleton* de la no representa una clase específica, sino un papel que puede ser desempeñado por algunas clases del sistema. Cuando el diseñador establezca que una clase determinada responde a este patrón, el correspondiente código *Prolog*, específico para la clase documentada, es generado.

Este mismo mecanismo es aplicable con patrones más complejos y *HiFi* provee una biblioteca con reglas de consistencia asociadas a patrones como *AbstractFactory* y *Composite*, por ejemplo. Sin embargo, es importante destacar que el mecanismo tiene limitaciones. En particular, no es posible asociar reglas de consistencia a todos los patrones del catálogo creado por Gamma y otros [GHJV94]. Tampoco se ha probado su utilización con otros tipos de patrones de diseño.

2.7. Asistente de Funcionalidad

Este módulo es el encargado de ayudar al usuario del *framework* a definir la funcionalidad que se requiere para la aplicación que se está implementando. Para ello, muestra la funcionalidad provista por el *framework* y permite que el usuario marque aquellas opciones que quiere estén presentes en la aplicación. Esta lista de opciones también se muestra como parte de los libros electrónicos (Figura 8.14). Algunas opciones tienen subopciones; en este caso, cuando el usuario elige la opción de mayor nivel, el sistema muestra las subopciones relacionadas, para que el usuario refine su elección. Por ejemplo, en la Figura 8.14 el usuario ha elegido la opción "*Interactive creation of connections*", lo cual lleva a que se muestren las subopciones "*Using a Menu*", "*Using a Tool*" y "*Using a Handle*".

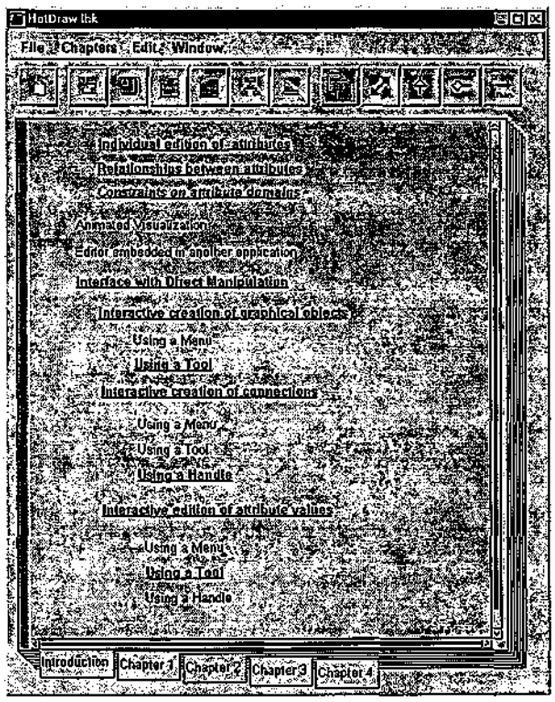


Figura 8.14. Ejemplo de captura de funcionalidad. Las opciones subrayadas y en negrita representan las elecciones del usuario

Tanto la entrada como la salida de esta herramienta están representadas en la notación para las acciones de instanciación presentadas en el capítulo V. De esta forma, las opciones elegidas por el usuario son pasadas directamente al planificador para la elaboración del plan.

2.8. Planificador

Este módulo es el encargado de implementar el algoritmo *PIT* descrito en el capítulo V. Trabaja a partir de las acciones de instanciación y de la funcionalidad provista por el asistente de funcionalidad. Una vez que el usuario da por finalizada la descripción de la funcionalidad requerida, el asistente de funcionalidad invoca al planificador para que este produzca el plan de instanciación. Como parte del plan de instanciación se genera la lista de tareas que el usuario debería ejecutar, la cual es utilizada por el Administrador de Tareas para guiar el proceso de instanciación.

El módulo planificador está escrito principalmente en *Prolog* y el aspecto más relevante de su implementación está relacionado con las tareas de espera (*waiting tasks*). Para proveer tanto la capacidad de definir tareas que detengan el proceso de planificación esperando una respuesta del usuario como la capacidad de rehacer el plan, fue necesario que el planificador se

ejecutara como un proceso independiente del resto del entorno, con el cual se comunica a través de un mecanismo de eventos.

Al iniciar el planificador, este se ejecuta como un proceso nuevo (ver Figura 8.15 y Figura 8.16). Si durante la creación del plan de instanciación no se encuentra ninguna tarea de espera, entonces el proceso terminará y retornará el plan que haya construido. En caso que una de las acciones de instanciación consideradas incluya una tarea de espera, el planificador creará la tarea, se la pasará al administrador de tareas (a través del repositorio) y suspenderá su ejecución a la espera de un evento. Cuando se produce el evento de finalización de la tarea esperada, el planificador solicita el resultado correspondiente al Administrador de Tareas, lo incorpora al plan actual y sigue la ejecución normal.

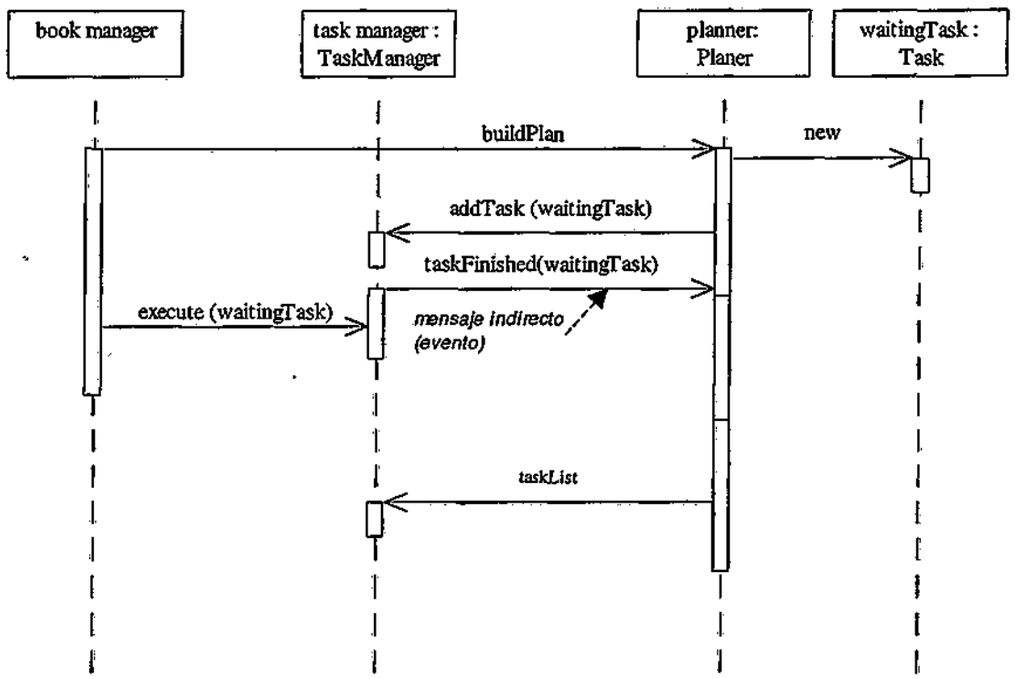


Figura 8.15. Interacción del Planificador con otros componentes del sistema

Esta ejecución como proceso separado es necesaria para que el planificador siga en todo momento dentro del código *Prolog*. De otra forma se perderían los vínculos de variables que se mantienen durante la ejecución y no sería posible reconsiderar decisiones ya tomadas en la planificación.

2.9. Administrador de Tareas

El Administrador de Tareas es una de las principales herramientas con las cuales interacciona el usuario durante el proceso de instanciación. Su principal objetivo es administrar la lista de tareas pendientes de ejecución, así como mantener información de aquellas que ya fueron ejecutadas.

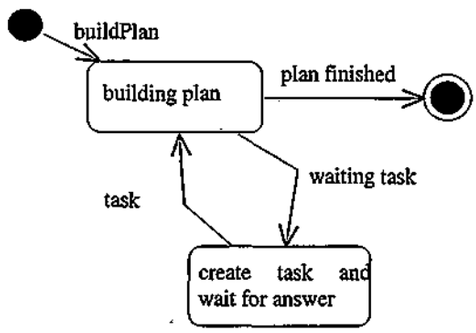


Figura 8.16. Vista parcial del diagrama de estados del Planificador

La mayor parte de la información utilizada por este módulo proviene del planificador. La lista de tareas que el planificador determina deberían ser ejecutadas para crear la aplicación, es presentada al usuario a través de la interfaz mostrada en la Figura 8.17. Esta lista puede ser tanto visualizada dentro de un libro de diseño, como mostrada en una ventana independiente.

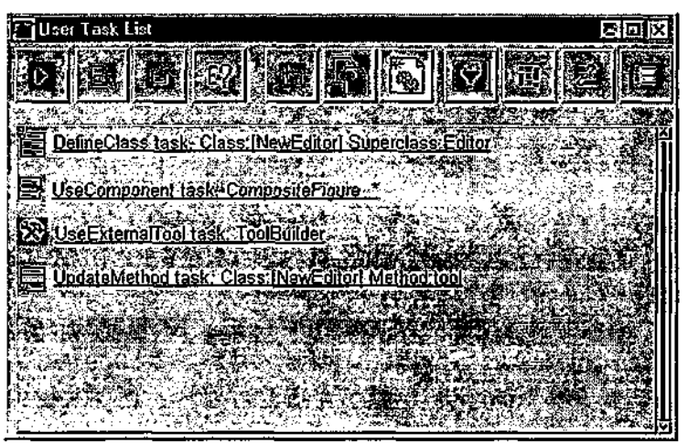


Figura 8.17. Interfaz del Administrador de Tareas. Cada línea representa una tarea

El usuario puede utilizar esta lista para ver tanto las tareas que ya fueron ejecutadas como las que aún están pendientes. Puede utilizar diversos métodos de ordenación y también aplicar distintos filtros, como por ejemplo ver sólo las tareas de creación de métodos o sólo las tareas aún no ejecutadas.

No todas las tareas mostradas al usuario tienen el mismo grado de obligatoriedad. Algunas son consideras *obligatorias* en el sentido que el usuario debería ejecutarlas, según el plan elaborado, para conseguir la funcionalidad requerida. Otras, consideras *opcionales*, sólo indican tareas que el usuario puede ejecutar, pero que no son necesarias para implementar la funcionalidad. En la interfaz del Administrador de Tareas, las tareas subrayadas representan a las tareas obligatorias, siendo el resto opcionales.

Utilizando la interfaz del Administrador de Tareas, el usuario puede seleccionar una tarea y ejecutar distintas acciones, dependiendo de si la tarea ya ha sido ejecutada. Si es una tarea pendiente, el usuario puede, entre otras cosas:

- navegar hacia información relacionada. Por ejemplo puede solicitar ver la regla que le dio origen, el código de la clase asociada o ejemplos de ejecución de la tarea;

- rechazar la tarea. Esto significa que el usuario solicita al sistema un plan alternativo de instanciación, que no incluya a esta tarea. Como consecuencia, el planificador deberá rehacer el plan propuesto;
- ejecutarla. Esto generalmente implica que se abra un formulario asociado con ese tipo de tarea para que el usuario lo complete. Por ejemplo, si es una tarea de creación de clase, se le solicita al usuario que provea nombre de clases y la superclase, en caso de que esta no se conozca. Otro posibilidad es que la tarea deba ser ejecutada utilizando otra herramienta, en cuyo caso la herramienta correspondiente es abierta. Por ejemplo, ver la anteúltima tarea de la Figura 8.18. Esta tarea tiene por objetivo crear un *tool* para *HotDraw*, tal como fue descrito en el capítulo II; la ejecución de la misma provocará que el *ToolBuilder* sea abierto.

En caso de que la tarea ya haya sido ejecutada, podrá navegar hacia la información relacionada (en este caso puede ver, por ejemplo, el código producido por la tarea) y pedir que la tarea sea deshecha. En este caso, todos los efectos de la tarea serán eliminados y la tarea puede ser manipulada como una tarea pendiente normal.

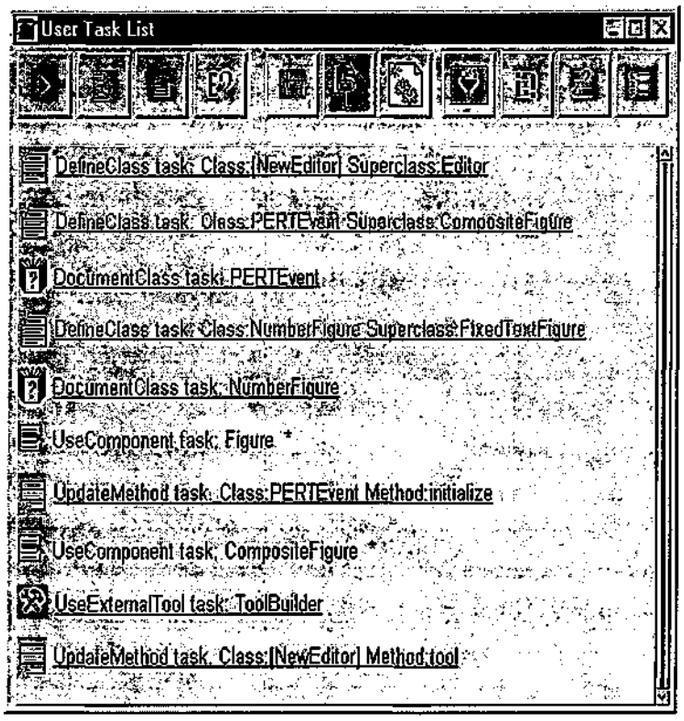


Figura 8.18. Administrador de Tareas durante la ejecución del proceso de instanciación

Cada vez que el usuario finaliza la ejecución de una tarea, el Administrador de Tareas construirá un evento con la información de la tarea finalizada y lo comunicará a las otras herramientas, para que aquellas interesadas tomen las acciones necesarias. Una de las consecuencias podría ser la activación de una regla de consistencia, lo que daría lugar a la creación de nuevas tareas. En este caso, las tareas son creadas por el Administrador de Consistencia y pasadas al Administrador de Tareas.

Para que el software producido durante la creación de una aplicación no quede sin documentar, el Administrador de Tareas crea tareas de documentación cada vez que una parte del software es modificado, como por ejemplo una nueva clase o un método. La herramienta de documentación tomará la información de la tarea ejecutada para ayudar a documentar el nuevo código.

Por ejemplo, la Figura 8.18 muestra el estado de la lista de tareas en un momento del proceso de instanciación. Las tareas acompañadas con un icono representando un libro son las tareas de documentación. La tercer tarea de la lista, por ejemplo, es una tarea de documentación correspondiente a la clase creada en la tarea anterior.

Entre las funciones provistas por el Administrador de Tareas está el controlar que la misma tarea no sea creada dos veces, es decir, que no se le pida al usuario que ejecute dos veces lo mismo. Esto es importante sobre todo cuando se rehacen los planes de instanciación, para que el usuario no tenga que realizar más de una vez el mismo trabajo.

Además, para ofrecer mayor flexibilidad al usuario, se le permite utilizar las herramientas normales del entorno *Smalltalk* para manipular el código. En este caso, el Administrador de Tareas supervisa las actividades del usuario y cuando éste modifica el código a través de cualquier herramienta, crea la tarea equivalente y la coloca como tarea ya ejecutada. En caso de que la tarea ejecutada coincida con alguna que ya estaba pendiente, se le pide al usuario que confirme que las dos tareas son efectivamente la misma.

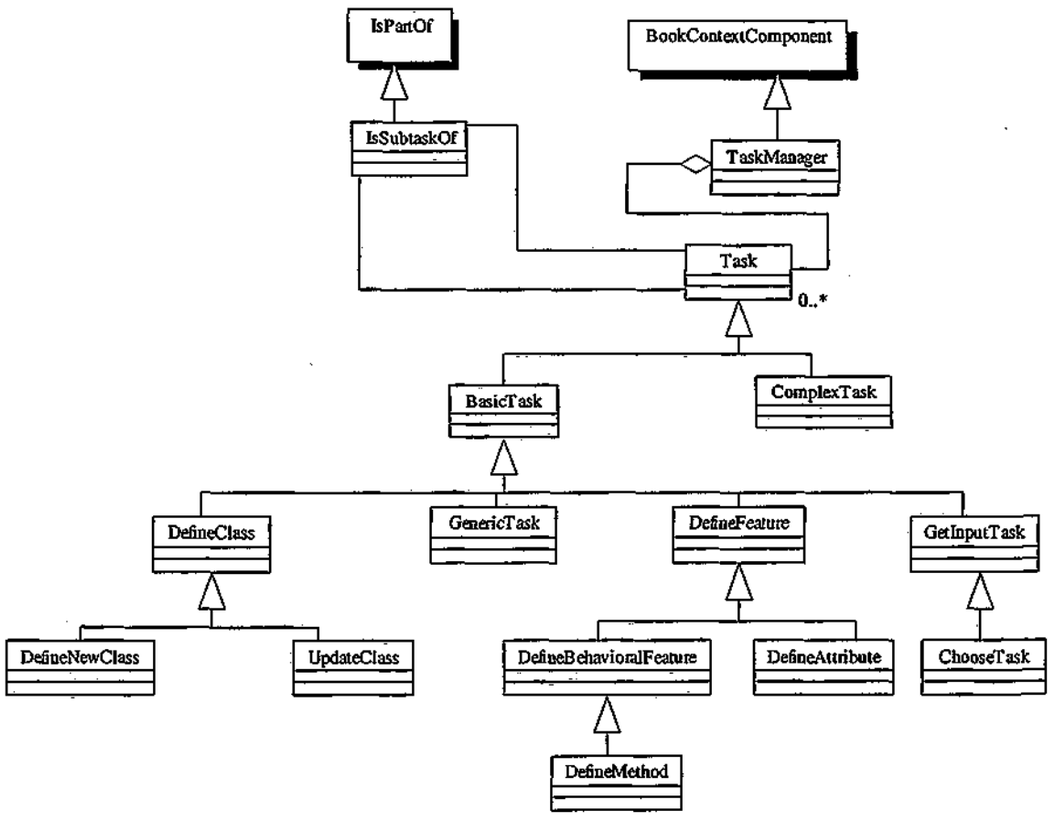


Figura 8.19. Jerarquía de clases del Administrador de Tareas

2.10. Administrador de Consistencia

Este módulo está también implementado mayormente en *Prolog* y es activado cada vez que el usuario da por finalizada una tarea. Como se explicó, al producirse la finalización de una tarea, el Administrador de Tareas genera un evento, el cual es transmitido a todas los componentes interesados. En particular, el Administrador de Consistencia codifica este evento como una consulta *Prolog* y lo transmite al intérprete *Prolog* para que lo procese. El resultado será que se compruebe en la base de datos de reglas *Prolog* si existe alguna regla de

consistencia cuya condición deba verificarse ante el tipo de evento producido. Si esto sucede, se procede a ejecutar el código asociado, creando las tareas necesarias para eliminar la inconsistencia o al menos informar sobre ella.

La Figura 8.20 muestra la organización jerárquica y las relaciones de las clases manipuladas por el Administrador de Consistencia. Estos elementos son utilizados para construir las reglas de consistencia, como por ejemplo la definida en la Figura 6.11 del capítulo VI. Puede verse que el Administrador de Consistencia administra un conjunto de reglas, cada una de las cuales está formada por representaciones de tareas (*TaskRepresentation*), Clases (*ClassRepresentation*) y Métodos (*MethodRepresentation*). Cada tarea contiene un conjunto de 0 o más argumentos obligatorios y 0 o más opcionales. Las clases y los métodos son asociados a las tareas a través de relaciones de causalidad (*CausalityConnection*); estas son las relaciones representadas con líneas a trazos en las definiciones de reglas de consistencia.

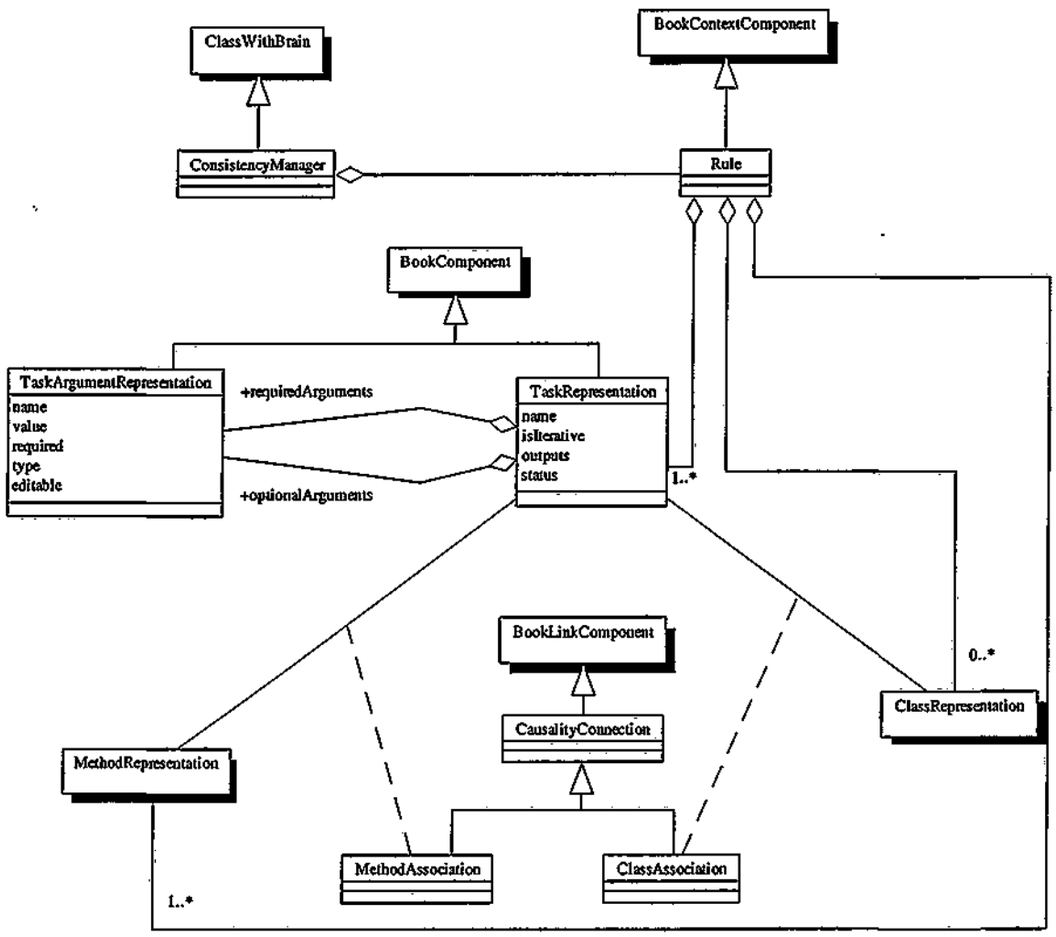


Figura 8.20. Jerarquía de clases manipuladas por el Administrador de Consistencia

IX Conclusiones

En los capítulos precedentes *SmartBooks*, un método para documentar y asistir en la instanciación de *frameworks* de aplicación orientados a objetos, fue presentado. El objetivo principal detrás del desarrollo de *SmartBooks* es brindar a un usuario de un *framework* dado asistencia inteligente, no existente en las herramientas actuales, para construir nuevas aplicaciones a partir de dicho *framework*.

Específicamente, la propuesta de *SmartBooks* es utilizar técnicas de planificación para generar, a partir de los requisitos funcionales de una aplicación, una secuencia de tareas de instanciación que deberían ser ejecutadas por el usuario para implementar esa funcionalidad. En combinación con esta asistencia basada en la funcionalidad, se provee un motor de reglas que permite, en cierta medida, seguir y controlar las acciones del usuario, aún cuando éstas se salen del marco previsto por la documentación basada en la funcionalidad. Esta combinación permite superar limitaciones de otros enfoques destinados a apoyar la instanciación de *frameworks*, ofreciendo flexibilidad y capacidad de adaptación a las distintas necesidades de los usuarios.

La documentación del *framework* debe centrarse en la funcionalidad provista por el mismo y en cómo esta funcionalidad se puede implementar. Estas descripciones funcionales se realizan en base a acciones de instanciación, a partir de las cuales un algoritmo de planificación puede elaborar un plan para implementar la funcionalidad requerida. La asistencia provista es complementada con reglas que permiten supervisar que el software desarrollado cumpla las normas subyacentes en el diseño del *framework*.

Además, en este trabajo se proponen técnicas destinadas a facilitar la utilización de *SmartBooks*, principalmente una notación gráfica para documentar las reglas de instanciación y un modelo de tareas de instanciación para representar las actividades del plan. Finalmente, se muestra el prototipo de un entorno para apoyar la documentación e instanciación de *frameworks* con *SmartBooks*.

A continuación, las principales contribuciones de la tesis son discutidas, así como algunas limitaciones de la propuesta y posibles líneas de investigación que pueden ser derivadas del presente trabajo.

1. Contribuciones

La principal contribución de esta tesis es el uso de técnicas de planificación para asistir la reutilización de software, en especial el desarrollo de aplicaciones basadas en *frameworks* orientados a objetos. La tesis muestra cómo este método, no documentado en la literatura más conocida del tema hasta hoy, puede ayudar a los usuarios de un *framework* a implementar aplicaciones derivadas de él. El método, denominado *SmartBooks*, propone documentar el *framework* en base a la funcionalidad provista, y utilizar el concepto de tarea de instanciación para estructurar el plan de instanciación presentado al usuario.

El uso de *SmartBooks* en la documentación e instanciación de un *framework* reduce la necesidad de adquirir un conocimiento profundo del mismo para instanciar una aplicación. Una primera consecuencia de esto es la disminución del esfuerzo necesario para crear una aplicación, reduciendo por lo tanto el tiempo necesario para su desarrollo. Pero más importante aún, *SmartBooks* estimula el desarrollo de software de mayor calidad, en la medida en que orienta a una correcta utilización del *framework*, al menos desde el punto de vista de sus diseñadores.

Además de esta contribución principal, la tesis ofrece varios aportes relacionados con el apoyo a la instanciación de *frameworks*, los cuales pueden ser resumidos por los siguientes puntos:

- Algoritmo de Planificación: Un algoritmo de planificación, llamado *PIT*, ha sido desarrollado para ser utilizado en una implementación de *SmartBooks*. Este algoritmo es una extensión de *UCPOP*, un algoritmo clásico basado en la técnica de minimización de compromisos. El algoritmo *PIT* posee características que lo hacen especialmente adecuado para su utilización en la instanciación de *frameworks*. Una de estas características es la capacidad de generar planes parciales. Los planes generados por el algoritmo *PIT* son parciales en más de un sentido:
  - Si la documentación que se posee sobre el *framework* es insuficiente para generar un plan que describa cómo implementar toda la funcionalidad requerida por el usuario, el algoritmo genera planes describiendo la implementación de aquellas porciones de funcionalidad contempladas en la documentación.
  - El plan generado, cubra o no todos los requisitos funcionales necesarios para la aplicación que se está desarrollando, puede contener referencias a objetos genéricos, que el usuario individualizará durante el proceso de instanciación. Por ejemplo, puede hacer referencia a una clase que será implementada por el usuario o a un método de una clase, sin especificar exactamente cuál. Esto permite describir situaciones en las cuales la identidad de alguno de los objetos manipulados no puede ser conocida en el momento de generar el plan o es preferible dejar al usuario la elección definitiva.
  - La secuencia de tareas que deberían ser ejecutadas es ordenada sólo parcialmente. De esta forma, se le permite al usuario llevar a cabo el plan con la mayor flexibilidad posible.

  La segunda propiedad distintiva del algoritmo *PIT* es la capacidad de generar los planes de forma incremental. Esto quiere decir que el plan no es generado a partir de una especificación monolítica única, sino que la descripción de funcionalidad requerida puede ser refinada en sucesivas etapas, siendo el plan producido ajustado en consecuencia. Estas modificaciones en las entradas al algoritmo pueden producirse en cualquier punto del proceso de desarrollo, tanto durante la creación del plan como durante la fase de instanciación.

  Además, también es posible rechazar un plan propuesto por el algoritmo, lo que ocasiona que se generen planes alternativos.

- Diseño de un Modelo de Tareas de Instanciación: Los planes de instanciación son presentados al usuario en forma de secuencias de tareas de instanciación que deberían ser ejecutadas. Estas tareas de instanciación están basadas en el concepto de tareas de usuario, utilizado en el dominio de interfaces a usuario. El modelo de tareas utilizado se basa en un conjunto de tareas básicas: creación y modificación de clases, métodos y atributos. En función de estas tareas es posible construir planes que describan todas las actividades necesarias para implementar una aplicación.

  Junto con un modelo de tareas de instanciación, se ha definido un administrador de tareas, que permite llevar un control de las actividades del usuario, proveyéndole asistencia sobre las actividades que ya ejecutó y aquellas que aún debe llevar a cabo.

- Representación de Acciones de Instanciación: Para la generación de los planes de instanciación, es necesario documentar el *framework* con descripciones que permitan determinar cómo puede ser implementada la funcionalidad provista. Con este objetivo, se ha diseñado una notación semiformal que permite especificar, utilizando acciones de instanciación, los pasos necesarios para implementar cada función disponible.

- Mantenimiento de Consistencia: Si bien la generación de planes de instanciación ofrece gran flexibilidad y adaptabilidad para asistir en la utilización de *frameworks*,

esta asistencia puede ser mejorada con la utilización de información complementaria. Específicamente, *SmartBooks* propone utilizar información sobre las reglas subyacentes en el diseño del *framework* para guiar la actividad de instanciación. De esta forma, es posible extender la asistencia provista a algunos casos en los que la funcionalidad requerida para la aplicación no ha sido prevista en la documentación. Para esto se han diseñado reglas de consistencia, que permiten describir situaciones en las que deben ser ejecutadas ciertas tareas para que el software desarrollado respete el diseño del *framework*. El uso de estas reglas es especialmente útil cuando se utilizan para describir situaciones recurrentes en el diseño de un *framework*, como por ejemplo los patrones de diseño.

- Notación Gráfica para las Reglas de Instanciación: Para facilitar el trabajo de documentar el *framework*, se ha diseñado una notación gráfica que permite especificar tanto reglas funcionales como de consistencia, si bien sólo en forma parcial. Esta notación, denominada *TOON*, está basada en la notación propuesta por *UML* para los diagramas de clases, extendida para permitir la representación de tareas.

- Construcción de un Prototipo de Entorno de Instanciación de *Frameworks (HiFi)*: El prototipo de un entorno para asistir en la instanciación de *frameworks* de acuerdo al método *SmartBooks* ha sido diseñado y construido. El prototipo *HiFi* permite construir libros electrónicos conteniendo la documentación del *framework* y en especial permite la definición de reglas de instanciación. En función de esto, una implementación del algoritmo *PIT* genera los planes de instanciación correspondientes. También se ha implementado un módulo para la administración de consistencia, el cuál utiliza una representación *Prolog* de las reglas para controlar la actividad del usuario. El entorno se completa con un administrador de tareas, que guía el trabajo del usuario del *framework* proveyendo una agenda de tareas pendientes.

2. Limitaciones

Se ha mostrado cómo el enfoque propuesto por *SmartBooks* puede ser utilizado para mejorar la asistencia provista por las técnicas actuales para la instanciación de *frameworks*. Sin embargo, este método posee algunas limitaciones.

Una limitación inherente del método propuesto es su dependencia de la documentación disponible. A pesar de que se provean medios para facilitar el trabajo de documentación del *framework*, sigue siendo una carga adicional impuesta a los diseñadores y como tal, susceptible de no ser realizada o de serlo en forma deficiente. De este modo, no es posible garantizar asistencia para instanciar un *framework* determinado, en la medida en que no es posible garantizar la existencia y calidad de la documentación correspondiente.

SmartBooks tiene también limitaciones referidas a la plataforma de implementación. Para proveer toda la asistencia descrita, es necesario disponer de un entorno abierto de desarrollo, que posibilite la implementación de herramientas que controlen la actividad del usuario. En particular, *HiFi* ha sido implementado utilizando el entorno de *VisualWorks* 2.0 de *Smalltak*. En la medida en que el entorno de desarrollo utilizado restrinja la implementación de esas herramientas, la utilidad de *SmartBooks* se verá reducida. Concretamente, siempre será posible construir un plan de instanciación de acuerdo a la funcionalidad requerida, independientemente de la plataforma. Sin embargo, no siempre será posible construir un mecanismo que supervise las actividades del usuario y sea capaz de reaccionar en consecuencia. Por otra parte, la posibilidad de contar con información de tipos, no presentes en *Smalltalk*, aumentaría las capacidades de asistencia de una herramienta de este tipo.

Finalmente, el método presenta una característica que puede ser considerada, desde cierto punto de vista, como una limitación, y está relacionada con el nivel de conocimientos necesarios para utilizar la herramienta. Si bien una de las presunciones iniciales de la propuesta es que el usuario no posee un conocimiento profundo del *framework*, sí en cambio se presupone que posee conocimientos de desarrollo de software orientado a objetos. Esto representa una diferencia respecto de otros enfoques, que están orientados a permitir que usuarios con pocos conocimientos de programación sean capaces de implementar aplicaciones. Este tipo de propuestas, generalmente conocidas como entornos visuales de programación, sólo han tenido algún éxito trabajando con *frameworks* específicos en dominios acotados, como por ejemplo la creación de interfaces de usuario basadas en diálogos. *SmartBooks* tiene como objetivo servir de apoyo a la instanciación de *frameworks* en general, sin restricciones de dominio y, debido a esto, los mecanismos utilizados para instanciar un *framework* requieren de conocimiento de programación en el lenguaje del *framework*.

Existen otras limitaciones que no son inherentes a *SmartBooks*, sino que son propias del estado actual de desarrollo e implementación de la propuesta. En este sentido, la mayor limitación actual está relacionada con la capacidad de describir la funcionalidad requerida para la aplicación, es decir, la utilización de menús para seleccionar dicha funcionalidad.

Este sistema para seleccionar la funcionalidad no sólo provee escasa flexibilidad, sino que fuerza al usuario a interpretar el significado de la descripción provista por el diseñador. Debe tenerse en cuenta que el vocabulario utilizado, las presunciones y visiones del dominio del aplicación pueden variar en gran medida de un usuario a otro y especialmente con respecto al conocimiento del diseñador. Por este motivo, no es posible asegurar que una determinada descripción funcional signifiquen lo mismo para el diseñador y para el usuario.

En la próxima sección se sugieren algunas líneas de investigación tendientes a superar limitaciones relacionadas con el estado actual de desarrollo de la propuesta, entre ellos la descripción de funcionalidad.

3. Trabajo Futuro

Diversos aspectos de *SmartBooks* requieren un mayor estudio, con vistas a poder superar algunas limitaciones que presenta actualmente el método.

Entre las líneas de investigación que pueden derivarse de *SmartBooks*, una de ellas es el desarrollo de extensiones al algoritmo de planificación. El propósito del algoritmo *PIT* es, fundamentalmente, mostrar la viabilidad de utilizar técnicas de planificación para ayudar en la instanciación de *frameworks*. Pese a que este objetivo ha sido cumplido, existen varios aspectos que pueden ser estudiados con mayor detalle en vistas a desarrollar un algoritmo más avanzado.

Entre otras cosas, es posible investigar sobre la utilización de funciones que intenten optimizar, desde distintos puntos de vista, la exploración del espacio de planes. Uno de los principales aspectos a ser considerados es la maximización del número de objetivos funcionales cubiertos por el plan generado.

Al considerar estas optimizaciones, más importante que mejorar aspectos relacionados con la construcción del plan (por ejemplo, optimizar el tiempo empleado en la ejecución del algoritmo), es optimizar el uso de recursos derivado del plan. Por ejemplo, es posible ajustar el algoritmo de planificación para que, dada la funcionalidad requerida, minimice el número de tareas que es necesario ejecutar o la cantidad de clases que deben ser especializadas.

Otra línea de investigación que puede derivarse de *SmartBooks* es el soporte al desarrollo colaborativo de aplicaciones. En la práctica, debe tenerse en cuenta que el desarrollo de software es realizado, casi siempre, por un grupo de personas. En consecuencia, un entorno destinado a proveer apoyo a la instanciación de *frameworks* debe tener capacidad de administrar las actividades llevadas a cabo por varias personas. Por ejemplo, debe ser capaz de distribuir

entre los distintos miembros del equipo de desarrollo las tareas que deban ejecutarse. También debe tener capacidad para coordinar la ejecución de tareas que comparten recursos, como por ejemplo aquellas que realizan modificaciones sobre la misma clase. En este sentido, el módulo administrador de tareas surge como el soporte natural para este tipo de funcionalidad.

El estudio del algoritmo de planificación también puede ser profundizado en relación al soporte provisto para desarrollo colaborativo. Específicamente, puede trabajarse en la búsqueda de métodos de construcción de planes que optimicen el uso de los recursos humanos disponibles.

Con respecto a la descripción de funcionalidad, el problema debe ser cuidadosamente estudiado para encontrar una forma menos ambigua de describir funcionalidad, sin las dificultades inherentes al uso de alguna notación formal. Una posible solución es elaborar, a partir de un análisis de dominio, un lenguaje de dominio, que permita una clara descripción de la funcionalidad provista por el *framework*. También deberían buscarse técnicas alternativas para que el usuario defina los requisitos de su aplicación, que ofrezcan más flexibilidad que un simple menú de opciones.

También se deben estudiar métodos para acompañar la evolución del *framework*. En la medida en que el diseño del *framework* es modificado, la documentación correspondiente debe ser actualizada. En consecuencia, resultaría conveniente proveer métodos que faciliten mantener la documentación al día, por ejemplo, detectando las porciones de la documentación que han sido afectadas por los cambios en el *framework*.

4. Consideraciones Finales

A pesar de los numerosos esfuerzos realizados en la Ingeniería de Software, la producción de software de calidad con costo razonable sigue siendo un problema de difícil solución. Esto se debe, principalmente, a la necesidad de desarrollar sistemas de software cada vez más complejos, con mayores restricciones de tiempo y presupuesto. Por este motivo, se hace más necesario contar con técnicas y herramientas de apoyo al desarrollo. En este sentido, la utilización de herramientas inteligentes surge como una alternativa con buenas perspectivas, que cada vez está recibiendo mayor atención.

En el caso de los *frameworks* orientados a objetos, estos constituyen una tecnología plenamente aceptada hoy en día en el desarrollo industrial de software, pero que todavía requiere de investigación para solucionar problemas relacionados con su utilización. Esta tesis tiene por objetivo aliviar uno de estos problemas: la dificultad de determinar los pasos necesarios para instanciar nuevas aplicaciones, en especial cuando se trata de usuarios inexpertos. Si bien posee limitaciones, constituye un avance concreto en el camino de facilitar la construcción de software.

Considerando el desarrollo de software en general, el uso de herramientas y técnicas de Inteligencia Artificial para soportar distintas etapas de este proceso abre nuevos caminos de investigación. Estos elementos ya han sido usados con éxito para soportar distintos aspectos del proceso, como por ejemplo técnicas de verificación basadas en conocimiento o la utilización de métodos de adquisición de conocimiento para elicitación de requisitos.

Sin embargo, existen otros aspectos que pueden beneficiarse de la utilización de herramientas inteligentes. En especial, resulta interesante su utilización para apoyar aquellas etapas del proceso de desarrollo que no tengan un método sistemático de ejecución, sino que es necesario aplicar conocimiento y experiencia para llevarlos a cabo, siendo el ejemplo más destacado el diseño del sistema.

Así, resulta necesario continuar las investigaciones tendentes a incrementar la utilización de herramientas inteligentes en el desarrollo de software, y de esta forma facilitar el trabajo de los desarrolladores, permitiendo obtener un desarrollo más eficaz y eficiente.

IX – Conclusiones

Anexo A Reglas y Acciones Primitivas

Esta sección presenta la lista de acciones de instanciación y reglas de consistencia primitivas, es decir, que son provistas por *HiFi* para ser utilizadas con cualquier *framework*. Los textos entre /* y */ representan comentarios sobre la acción descrita a continuación de cada texto. Los números que acompañan cada acción sirven sólo a los efectos de facilitar la referencia a cada acción, en especial en el ejemplo mostrado en el capítulo VII.

1. Acciones Primitivas

1.1. Acciones que originan Tareas Pendientes

1. pendingTask ("DefineClass", [NewClass, SuperClass], Description, "Required"),
   → defineClass(NewClass, SuperClass, Description)
2. pendingTask ("DefineClass", [NewClass, SuperClass], Description, "Optional"),
   → optionalDefineClass(NewClass, SuperClass, Description)

/ Esta tarea se utiliza para cambiar una clase existente. Por ejemplo, cambiar la superclase (en este caso sólo permite refinar) o cambiar la visibilidad de la clase. El tipo de cambio debe ser explicado en Description */*

3. pendingTask("UpdateClass", [Class, SuperClass], Description, "Required")
   → updateClass(Class, SuperClass, Description)
4. pendingTask("UpdateClass", [Class, SuperClass], Description, "Optional")
   → optionalUpdateClass(Class, SuperClass, Description)

/* Agrega una nueva variable a la definición de la clase */

5. pendingTask ("DefineAttribute", [Class, Name, Type, OwnerScope, Visibility], Description, "Required"),
   optionalDefineMethod (Class, Name, OwnerScope, Visibility, string("^", Name), [], "Access method for the attribute")
   → defineVariable(Class, Name, Type, OwnerScope, Visibility, Description)
6. pendingTask ("DefineAttribute", [Class, Name, Type, OwnerScope, Visibility], Description, "Optional")
   → optionalDefineVariable(Class, Name, Type, OwnerScope, Visibility, Description)

/ Modifica la definición de una variable existente. Puede variar el tipo, el ownerScope y /o la visibilidad */*

7. pendingTask ("UpdateAttribute", [Class, Name, Type, OwnerScope, Visibility], Description, "Required")
   → updateVariable (Class, Name, Type, OwnerScope, Visibility, Description)
8. pendingTask ("UpdateAttribute", [Class, Name, Type, OwnerScope, Visibility], Description, "Optional")
   → optionalUpdateVariable (Class, Name, Type, OwnerScope, Visibility, Description)

/ Agrega un nuevo método a la definición de la clase. Puede incluir una porción de código que debe contener el método, así como variables locales que deben ser declaradas */*

9. pendingTask ("DefineMethod", [Class, Name, OwnerScope, Visibility, String, LocalVars], Description, "Required")
   → defineMethod (Class, Name, OwnerScope, Visibility, String, LocalVars, Description)

10. pendingTask ("DefineMethod", [Class, Name, OwnerScope, Visibility, String, LocalVars], Description, "Optional")
    → optionalDefineMethod (Class, Name, OwnerScope, Visibility, String, LocalVars, Description)

/ Modifica la definición de una variable existente. Puede variar el ownerScope, la visibilidad y/o el cuerpo del método*/*

11. pendingTask("UpdateMethod", [Class, Method, OwnerScope, Visibility, String, LocalVars], Description, "Required")
    → updateMethod(Class, Method, OwnerScope, Visibility, String, LocalVars, Description)

12. pendingTask("UpdateMethod", [Class, Method, OwnerScope, Visibility, String, LocalVars], Description, "Optional")
    → optionalUpdateMethod(Class, Method, OwnerScope, Visibility, String, LocalVars, Description)

/ Tool = nombre de la herramienta a ser invocada. Message = mensaje que se envía para abrir la herramienta. Arguments = argumentos a incluir en el mensaje de apertura; Description = Información del contexto para el usuario */*

13. pendingTask ("UserExternalTool", [Tool, Message, Arguments], Description, "Required")
    → useExternalTool (Tool, Message, Arguments, Description)

14. pendingTask("UserExternalTool", [Tool, Message, Arguments], Description, "Optional")
    → optionalUseExternalTool (Tool, Message, Arguments, Description)

15. pendingTask("DocumentClass", [Class, Description], "Optional")
    → documentClass (Class, Description)

16. pendingTask ("DocumentAttribute", [AttributeName, Class, Description], "Optional")
    → documentAttribute (AttributeName, Class, Description)

17. pendingTask ("DocumentMethod", [MethodName, Class, Description], "Optional")
    → documentAttribute (MethodName, Class, Description)

1.2. Acciones que originan Tareas en Espera

18. waitingTask ("SelectMethod",[Class, MethodKind, Category, Description], Return)
    → selectMethod(Class, MethodKind, Category, Description, Return)

19. waitingTask ("AskSelection",[Options, Description], Return)→ selection(Options, Description, Return)

20. waitingTask ('GetInputTask', [Description], Return) → getUserInput (Description,Return)

/ Toma el método Method de la clase Class y asocia los argumentos a usar en la invocación del mensaje Message, los que pueden ser variables locales (incluyen parámetros del método que invoca) o atributos del objeto. MessageString es el texto de la invocación resultante. */*

21. waitingTask ("AssignArguments", [Method, Class], [Attributes, LocalVars, MessageString])
    → assignArguments(Message, Method, Class, Attributes, LocalVars, MessageString)

1.3. Otras primitivas

22. exists (class (CompDescription, X)), choose (CompDescription, ["refine", "reuse"], Answer), useComponent (C, CompDescription, Answer)
    → tryUseComponent (C, CompDescription).
23. ¬ exists (class (CompDescription,X)), defineNewComp (Component, CompDescription)
    → tryUseComponent (Component, CompDescription)
24. defineSubclass (NewClass, SuperClass, CompDescription)
    → useComponent (SuperClass, CompDescription, "refine")
25. reuseClass (C, CompDescription)
    → useComponent (C, CompDescription, "reuse")
26. exist (Class, Superclass) → checkIfExists (Class, Superclass)
27. not(exists(Class, Superclass)), defineClass(Class, Superclass) → checkIfExists (Class, Superclass)

2. Reglas de Consistencia

En esta sección se presentan algunas de las reglas de consistencia comunes a todos los *frameworks* documentados utilizando *HiFi*. Las tres primeras reglas establecen que cada vez que una clase, método o atributo sea creado o modificado, se creará una tarea de documentación. De esta forma, si el usuario ejecuta estas tareas, permitirá que el software desarrollado para una aplicación quede documentado. Entre otras cosas, esta documentación ayudará a guiar posteriores instanciaciones. La Figura A.1 describe las acciones a tomar cuando se crea o modifica una clase.

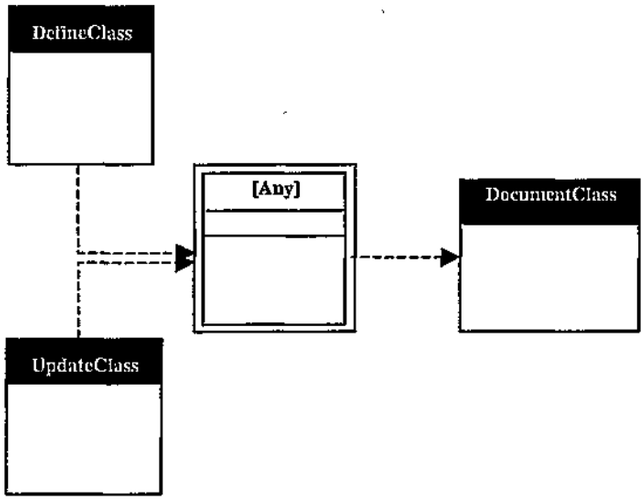


Figura A.1 Regla de documentación de clases

Es importante notar que las tareas de documentación podrían haber sido creadas por el planificador, a partir de las acciones de instanciación, como parte del plan de instanciación original. Por ejemplo, la acción de instanciación A_1 (§A.1.1) podría crear, además de la tarea de definición de clase, una tarea para documentar esta clase. Sin embargo, esto provocaría ambigüedades en las acciones que crean tareas opcionales, donde no sería correcto crear la tarea de documentación hasta tanto el usuario haya decidido ejecutar la tarea principal. Por este motivo se ha preferido crear todas las tareas de documentación como efecto de reglas de consistencia.

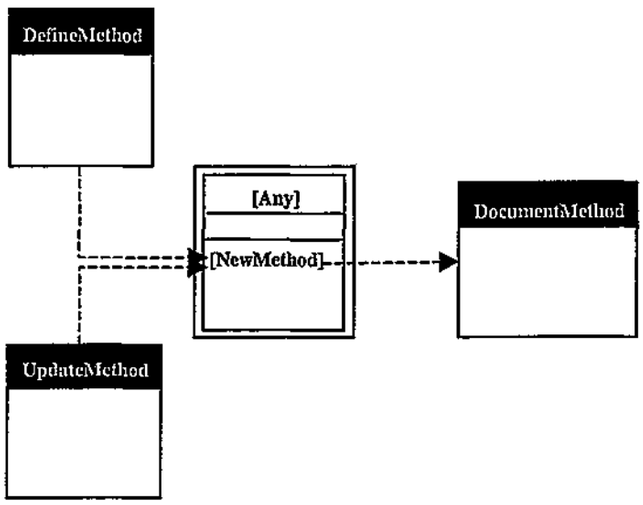


Figura A.2 Regla de documentación de métodos

Las figuras A.2 y A.3 describen las situaciones derivadas de la creación/modificación de métodos y atributos, respectivamente.

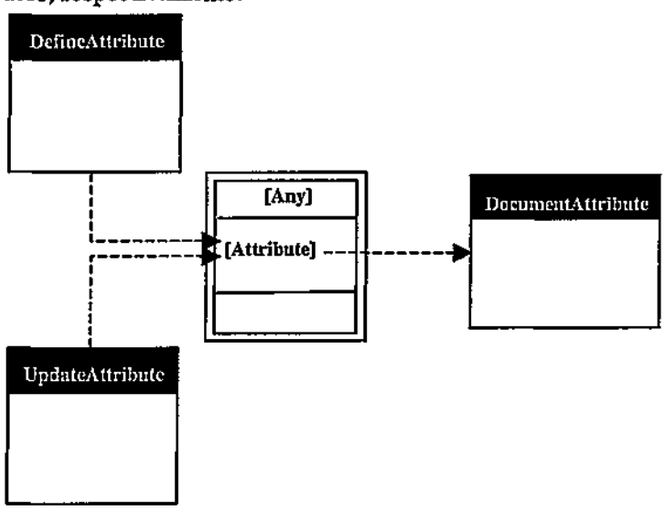


Figura A.3 Regla de documentación de atributos

Además de las reglas de documentación, hay otras reglas que describen situaciones comunes de encontrar en distintos *frameworks*. Un ejemplo de esta reglas es el caso de los métodos *template*. Un método *template* es, por definición, un método que invoca a métodos abstractos y/o *hooks*. Una característica de este tipo de métodos es que está previsto que no sean especializado en subclases. Por lo tanto, es posible definir una regla de consistencia que cree una tarea de advertencia cada vez que un método, que ha sido documentado como *template*, sea especializado. Esta regla es mostrada en la Figura A.4. Otros ejemplos de reglas generales provistas por *HiFi* son la regla para los métodos abstractos, descrita en §VI.4.1 y la regla para el patrón *Singleton*, mostrada en §VIII.2.6

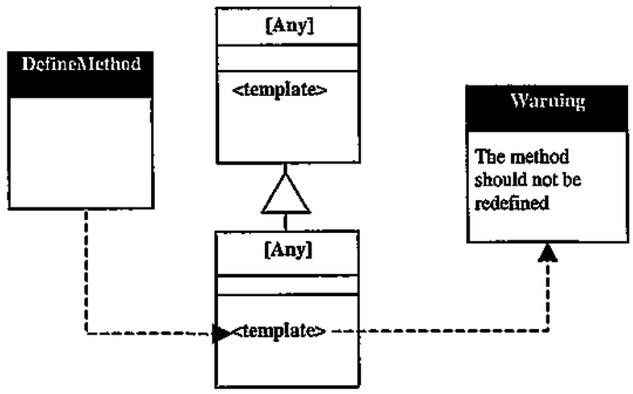


Figura A.4 Regla para la especialización de métodos *template*.

Anexo A - Reglas y Acciones Primitivas

Anexo B Acciones de Instanciación de HotDraw

En este anexo se describen las acciones de instanciación utilizadas para documentar el *framework* para editores gráficos *HotDraw*. En base a estas acciones se ha probado la utilización de la herramienta *HiFi* para asistir la instanciación de *frameworks*.

Es importante destacar que, aunque las acciones de instanciación ha sido diseñadas para documentar la funcionalidad de un framework, en algunas de estas acciones también se ha incluido información que permite determinar ciertas características no funcionales de la aplicación. Así, por ejemplo, todas las acciones cuya aplicación implica la utilización del subsistema de restricciones provisto con *HotDraw* hacen explícita esta información. De esta forma, sería posible que un usuario especifique que la funcionalidad requerida debe ser implementada sin utilizar dicho subsistema.

La lista de acciones está presentada en orden alfabético, teniendo en cuenta el primer efecto de cada acción, excepto cuando el orden sea significativo (porque el planificador debe considerarlos en un orden determinado).

1. Acciones de instanciación para *HotDraw*

/ Todas las acciones que tengan como efecto $required serán aplicadas en cada plan. Se utilizan para representar condiciones que deben ser satisfechas independientemente de los objetivos funcionales de cada aplicación */*

1. defineEditor → $required

/ Especialización de la acción de crear un tool desde un método, para el caso de las tools de creación de figuras. Otros efectos de esta acción son las moficaciones del las clases representadas por las variables Editor y Figure */*

2. beingDefined(Editor)
   updateMethod(Editor, "tools", "class", "public", string(Figure, " creationTool;"), [], "Add the tool to the editor"),
   defineMethod(Figure, "creationTool", "class", "protected", string("^Tool icon: cursor: Cursor origin class:self creationMessage:#createAt:"), [], "The cursor parameter should be an instance of Image")
   → createToolWith("Method", ["CreateFigure", Figure]), changed(Editor), changed(Figure)

/ Acción genérica de creación de un tool desde un método */*

3. beingDefined(Editor),
   selectMethod("Tool", "class", "creation tools", "Choose the method used to create the tool", Method),
   defineMethod(Editor, ToolName, "class", "protected", string("^Tool ", Method), [], ""),
   updateMethod(Editor, "tools", "class", "public", string("self ", ToolName), [], "Add the tool to the editor")
   → createToolWith("Method", Goal, Args), changed(Editor), useTool

/ Acción genérica de creación de un tool utilizando el ToolBuilder */*

4. beingDefined(Editor),
   useExternalTool("ToolBuilder", "open", [Goal, Args], ""),
   updateMethod(Editor, "tools", "class", "public", string("Tool ", Figure, "Tool;"), [], "Add the tool to the editor")
   → createToolWith("ToolBuilder", Goal, Args), changed(Editor), useTool

5. defineVariable(Figure, Name, "undefined", "instance", "public", Description),
   initializeAttribute(Figure, Name, Description)
   → defineAttribute(Figure, Name, Description), changed(Figure)

6. forEach(X, List, defineAttribute(Class, X, "")) → defineAttributeList(Class, List)

7. not(moreThanOneDrawing),
   defineClass(Editor, "Editor", "The class that implements the editor itself"),
   defineMethod(Editor, "tools", "class", "private", string("^(OrderedCollection new) yourself"), [],
   "This method should be updated every time a tool is added to the editor")
   → defineEditor, beingDefined(Editor), changed(Editor)

/ Existen dos formas de definir cómo se verá una figura: si la figura es simple, es posible redefinir el método displayOn:; en cambio si es una figura compuesta, se deben definir las clases de los subcomponentes (si no existen) y crear las respectivas instancias en el método initializeAt: */*

8. not(composite(ClassName)),
   optionalDefineMethod(ClassName, "displayOn:", "instance", "public", string(""), [], "")
   → defineFigureLayout(ClassName), simple (ClassName)

9. getUserInput("Component list", Return),
   forEach(X, Return, checkIfExists(X, "Figure")),
   updateMethod(ClassName, "initializeAt:", "instance", "private", string("self add:", Return), ""),
   → defineFigureLayout(ClassName), changed(ClassName)

/ La siguiente acción comprueba que en el plan actual no haya una acción que produzca como efecto la edición de atributos utilizando menús */*

10. not(editAttribute("Menu", _, _)),
    not(editAttribute("Tool", _, _)),
    selectMethod("ConstraintHandle", "class", "special Instances", "Choose the method used to create the handle", Method),
    updateMethod(ClassName, "handle", "instance", "private", string("add: (ConstraintHandle", Method, ");", [], "Add the handle to the list of the figure"),
    → editAttribute("Handler", ClassName, AttributeName), changed(ClassName), useConstraints, useHandle

11. not(editAttribute("Handler", _, _)),
    not(editAttribute("Tool", _, _)),
    useMenu (ClassName)
    → editAttribute("Menu", ClassName, AttributeName), useMenu

12. not(editAttribute("Handler", _, _)),
    not(editAttribute("Menu", _, _)),
    not(simple(ClassName)),
    defineClass (ClassName, "CompositeFigure", "An attribute has to be edited"),
    specificInitialization ("edit", ClassName, AttributeName, Description),
    useTool("editAttribute", [])
    → editAttribute("Tool", ClassName, AttributeName), composite(ClassName),
    changed(ClassName), useTool

13. updateMethod(Class, _, _, _, "DrawingView on:", [], "The method should create the DrawingView, and assign it a model that understands the DrawingEditor protocol"
    → embedded, changed(Class)

14. updateMethod(Figure, "initializeAt:", "instance", "protected", string(Name, "NumberFigure string:'0' at: aPoint."), [], Description)
    → initEditableAttribute("number", Figure, Name, Description), changed(Figure)

15. updateMethod(Figure, "initializeAt:", "instance", "protected", string(Name, ":= TextFigure string:'' at: aPoint."), [], Description)
    → initEditableAttribute("text", Figure, Name, Description), changed(Figure)

(15.5) if (editAttribute(_, Figure, Name)), specificInitialization("edit", Figure, Name, Description)
    → initializeAttribute(Figure, Name, Description)

16. selection(["edit", "noEdit"], Name, Ret), specificInitialization(Ret, Figure, Name, Description)
    → initializeAttribute(Figure, Name, Description)

17. beingDefined(Editor),
    defineClass(NewTool, Tool, ""),
    updateMethod(Editor, "tools", "class", "public", string(NewTool, " new;"), [], "Add the tool to the editor"),
    defineMethod (NewTool, "activate", "instance", "public", string("self drawing startAnimation"), [], ""),
    defineMethod (NewTool, "deactivate", "instance", "public", string("self drawing stopAnimation"), [], ""),
    drawingDefined(Drawing),
    updateMethod(Drawing, "step", "instance", "public", string(""), [], "This method should verify the state of some condition changed by startAnimation and stopAnimation methods")
    → interactiveAnimation, changed(NewTool), changed(Editor), changed(Drawing), useTool

/ Creación interactiva usando un menú */*

18. beingDefined(Editor),
    not(interactiveCreationWithTool(_)),
    useMenu (Editor)
    → interactiveCreationWithMenu (Figure), useMenu

/ Creación interactiva usando un Tool */*

19. not(interactiveCreationWithMenu(_)),
    useTool ("CreateFigure", [Figure])
    → interactiveCreationWithTool(Figure), useTool

/ Implementación de la Animación */*

20. defineClass(Drawing, "Drawing", ""),
    defineMethod (Drawing, "step", "instance", "public", string(""), [], "This method should update the position of every figure in the diagram"),
    beingDefined(Editor),
    defineMethod (Editor, "drawingClass", "class", "public", string(Drawing), "")
    → makeAnimatedDrawing, drawingDefined(Drawing), changed(Drawing), changed(Editor)

21. selection(["Tool", "Handle"], "Interaction Technique", Return),
    makeRelationshipUsing(Return, Source, Dest)
    → makeRelationship(Source, Dest)

/ La notación {op1|op2|...|opn} se utiliza para indicar al usuario que debe elegir uno de esos valores para al expresión final de código */*

22. getUserInput ("Class of Relationship", RelationClass), checkIfExists (RelationClass, "LineFigure"),
    updateMethod("handles", Source, string("handle := CommandHandle connectionFor:self at: {#center | #left | #right} class:", RelationClass, "action:[:source :dest :aFigure | dest addSource: source. source addDest: dest]. handles add:handle."), ["handle"], "Put this code after assigning handles variable"),
    optionalUpdateMethod(Dest, "addSource:", "instance", "public", string(""), [], ""),
    optionalDefineMethod(Dest, "connectTo:", "instance", "public", string("self {center|top|left|right|bottom}: aVariable"), [], "Return a constraint between aVariable (the argument) and the connection position"),
    optionalUpdateMethod(Source, "addDest:", "instance", "public", string(""), [], "")
    → makeRelationshipUsing("Handle", Source, Dest), changed(Source), useConstraints, useHandles

23. getUserInput("Class of Relationship", RelationClass), checkIfExists (RelationClass, "LineFigure"),
    useTool("MakeRelationship", [Source, Dest]),
    optionalUpdateMethod(Dest, "addSource:", "instance", "public", string(""), [], ""),
    optionalDefineMethod(Dest, "connectTo:", "instance", "public", string("self {center|top|left|right|bottom}: aVariable"), [], "Return a constraint between aVariable (the argument) and the connection position"),
    optionalUpdateMethod(Source, "addDest:", "instance", "public", string(""), [], "")
    → makeRelationshipUsing("Tool", Source, Dest), useTools

24. beingDefined(Editor),
    defineClass(Editor, "Object", "The model of the drawings"),
    optionalDefineMethod(Editor, "currentTool", "instance", "public", "", [], ""),
    optionalDefineMethod(Editor, "menu", "instance", "public", "", [], ""),
    optionalDefineMethod(Editor, "drawing", "instance", "public", "", [], "")
    → moreThanOneDrawing, defineEditor, changed(Editor)

/ Describe la creación de editores con más de una vista */*

25. beingDefined(Editor),
    updateMethod (Editor, "open", "class", "public", "DrawingView on: Editor", [], "The method should create a DrawingView and the other views")
    → moreThanOneView, changed(Editor)

/ Crea relaciones (restricciones) entre atributos de una misma figura */*

26. selectMethod("HotDrawConstraint", "class", "instance creation", Description, Message),
    assignArguments(Message, "initializeAt:", Class, Attributes, LocalVars, MessageString)
    defineAttributeList (Class, Attributes),
    updateMethod(Class, "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), LocalVars, Description)
    → relateAttributes(Class), changed(Class), useConstraints

/ Crea relaciones (restricciones) entre atributos de distintas figuras */*

27. selectMethod("HotDrawConstraint", "class", "instance creation", Description, Message),
    assignArguments(Message, "initializeAt:", Class1, Attributes, LocalVars, MessageString)
    defineAttributeList (Class1, Attributes),
    updateMethod(Class1, "initializeAt:", "instance", "protected", string("HotDrawConstraint", MessageString), LocalVars, Description)
    updateMethod(Class1, "addDest:", "instance", "public", string(""), [], ""),
    updateMethod(Class2, "addSource:", "instance", "public", string(""), [], ""),
    → relateAttributes(Class1, Class2), changed(Class1), changed(Class2), useConstraints

28. selection(["text", "number"], Name, Ret), initEditableAttribute(Ret, Figure, Name, Description)
    → specificInitialization("edit", Figure, Name, Description)

29. updateMethod(Figure, "initializeAt:", "instance", "protected", string(Name, ":= HotDrawVariable with:0 owner:self."), Description)
    → specificInitialization("noEdit", Figure, Name, Description), changed(Figure)

30. defineClass(ClassName, "Figure", Description),
    defineFigureLayout(ClassName)
    → useFigure(ClassName, Description), changed(ClassName)

31. updateMethod(ClassName, "menu", "instance", "public", string(""), [], "Agregar una etiqueta y un valor a la inicialización del menú. El valor debe coincidir con el nombre de un método de la clase")
    → useMenu(ClassName), changed(ClassName)

/ Acciones genéricas para usar una herramienta */*

32. selection(["ToolBuilder", "Method"], string("Tool Definition Technique for", Goal), Return),
    createToolWith(Return, Goal, Args)
    → useTool(Goal, Args)

Anexo C - Metamodelo de UML

En este anexo se presenta resumidamente el metamodelo propuesto por UML para la representación de software [Rat97]. El objetivo de este metamodelo es describir la sintaxis de la notación *UML* y si bien carece de formalidad por estar descrito en el mismo UML, resulta útil para la identificación de los elementos que es necesario representar en la documentación de software orientado a objetos. Por este motivo, ha sido utilizado para la representación del software en el prototipo *HiFi* (capítulo VIII).

La Figura C.1 presenta las principales clases utilizadas para representar el software. Constituyen lo que *UML* denomina la *columna vertebral* del metamodelo (*backbone*).

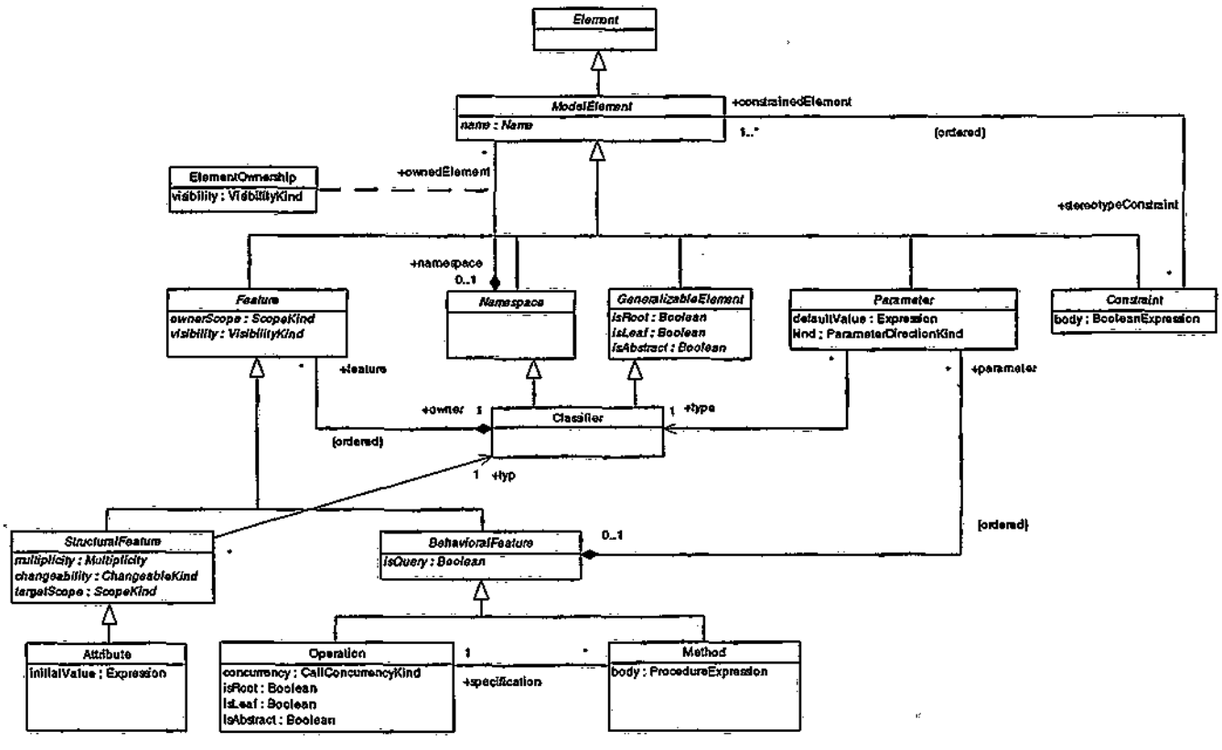


Figura C.1 Diagrama de clases del *backbone* de UML

En la Figura C.2 se presenta el diagrama de las clases utilizadas para la representación de relaciones y en la Figura C.3 las clases de representación de clasificadores.

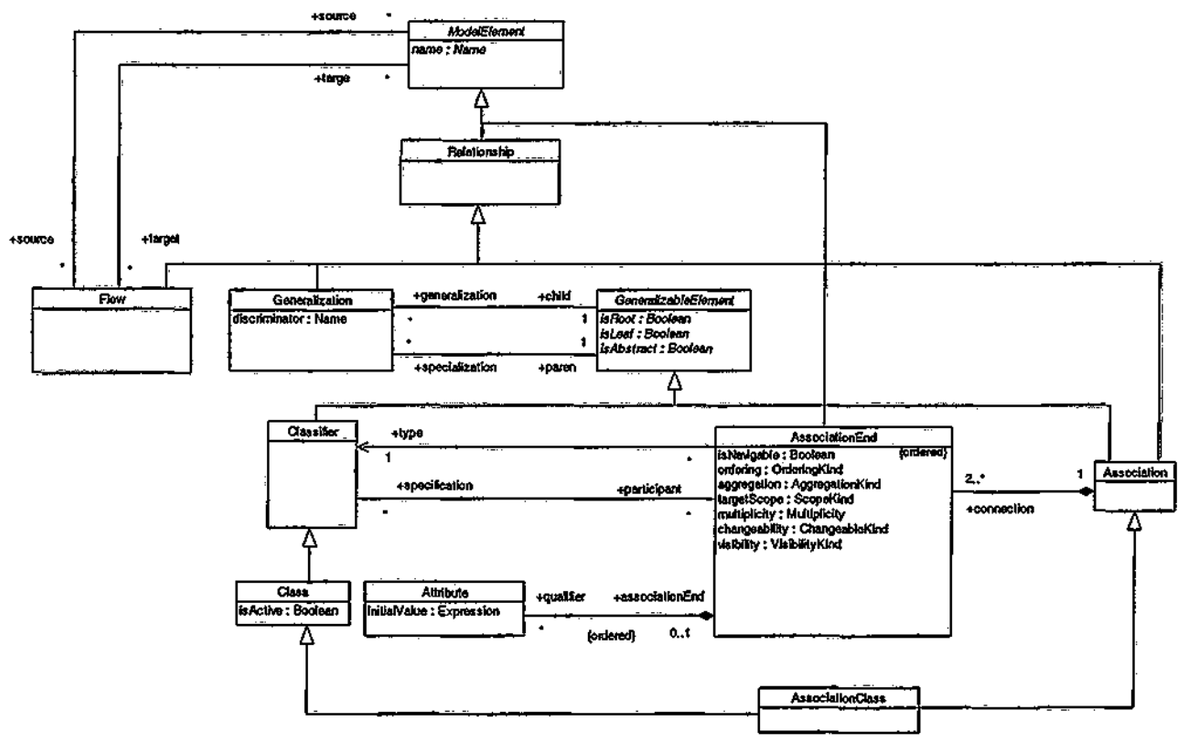


Figura C.2 Diagrama de clases de las relaciones en *UML*

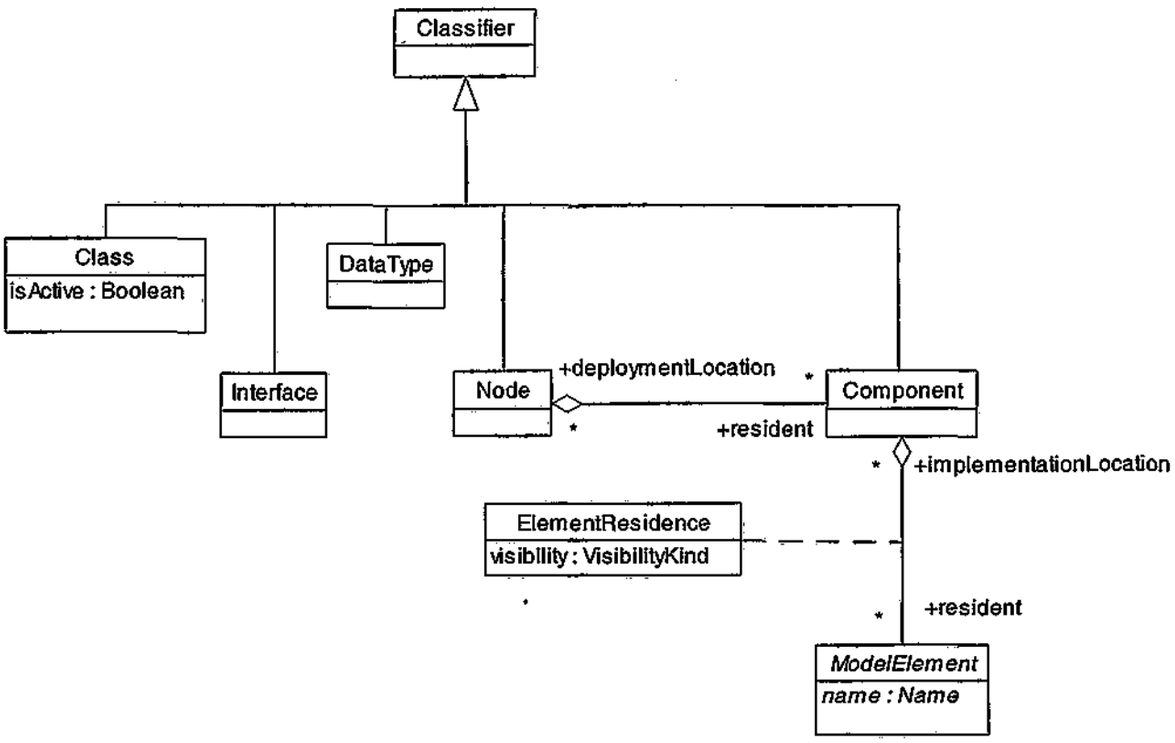


Figura C.3 Diagramas de las clases utilizadas para representar clasificadores

Anexo C – Metamodelo de UML

Anexo D – Índice de Términos Utilizados

En primer lugar se especifica la sección donde el término es explicado o citado por primera vez. Entre paréntesis figura el número de la página correspondiente.

Referencias Bibliográficas

[Ado85] Adobe Systems. **Postscript Language - Tutorial and Cookbook**. Addison-Wesley, 1985

[Ama97] Amandi A. **Programação de Agentes Orientada a Objetos**. Tesis de Doctorado – CPGCC, Universidade Federal do Rio Grande do Sul – 1997. (en portugués)

[ASP93] Arango G., Shoen E., Pettengill R. **A process for consolidating and reusing design knowledge**. En Proceedings of 15th International Conference on Software Engineering. IEEE Computer Press, 1993.

[BD99] Butler G., Dénommée P. **Documenting Frameworks**. En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.). John Wiley and Sons, N.Y, 1999.

[BJ94] Beck K., Johnson R. **Patterns Generate Architectures**. En Proceedings of European Conference on Object Oriented Programming (ECOOP94). Springer-Verlag, Berlin, 1994

[BrJ94] Brant J., Johnson R. **Creating Tools in HotDraw by Composition.** ftp://st.cs.uiuc.edu/pub/papers/HotDraw/HotDrawTools.ps. 1994

[BKM99] Butler G., Keller R., Mili H. **A Framework for Framework Documentation.** A ser publicado en ACM Computing Surveys, special symposium issue on Object-oriented Application Frameworks. Actualmente en http://www.cs.concordia.ca/~faculty/gregb

[BMA97] Brugali D., Menga G., Aarsten A. **The Framework Life Span**. Communications of the ACM. v.40, n.20. Octubre 1997.

[BMM+99] Bosch J., Molin P., Mattsson M., Bengtsson P., Fayad M. **Framework Problems and Experiences.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[BMR+96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. **Pattern-Oriented Software Architecture – A System of Patterns.** John Wiley & Sons Ltd. 1996. ISBN 0-471-95869-7

[Boo99] Boon J. **Harvesting Desing**. En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[Bos98] Bosch, J. **Problems and Issues in Industrial Product-Line Architectures**. Proceedings of European Reuse Workshop. Madrid, October 1998

[Bos98b] Bosch J. Specifying Frameworks and Design Patterns as Architectural Fragments. En Proceedings TOOLS Asia 98. Australia, 1998.

[Cah99] Cahill V. **Tigger: A Framework Supporting Distributed and Persistent Objects**. En Implementing Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[Cam97] Campo M. **Compreensão Visual de Frameworks através da Instrospeção de Exemplos**. Tesis de Doctorado – CPGCC, Universidade Federal do Rio Grande do Sul – 1997. (en portugués)

[CHSV97] Codenie W., De Hondt K., Steyaert P., Vercammen A. **From Custom Applications to Domain-Specific Frameworks.** Communications of the ACM. v.40, n.20. Octubre 1997.

[Con98] Contreras, J. **A Framework for the Automatic Generation of Software Tutoring.** PhD. Thesis, Université René-Descartes, Paris. 1998

[CPR99] Carro R., Pulido E., Rodríguez P. **Dynamic generation of adaptive Internet-based courses.** Journal of Network and Computer Applications 22, 249-257. 1999

[Deu89] Deuch, P. **Framework and Reuse in the Smalltalk-80 System.** En Software Reusability: Applications and Experience. Biggerstaf, Perlis (eds.) ACM Press, New York, 1989

[DHS98] Demeyer S., De Hondt K., Steyaert P. **Consistent Framework Documentation with Computed Links and Frameworks Contracts.** ACM Computing Surveys, Symposium on OO Application Frameworks. 1998.

[DW98] D'Souza D., Will A. **Objects, Components and Frameworks with UML. The Catalysis Approach.** Addison-Wesley. Reading, Massachusetts, 1998.

[ECK99] Eckstein J. **Empowering Framework Users.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[Fay99] Fayad M. **Future Trends.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[FHLS99] Froehlich G., Hoover H., Liu L., Sorenson P. **Reusing Hooks.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[FN71] Fikes, R., Nilsson N. **STRIPS: A new approach to the application of theorem proving to problem solving.** Artificial Intelligence, 2(3/4), 1971

[FS97] Fayad M., Schmidt, D. **Object-Oriented Application Frameworks.** Communications of the ACM. v.40, n.20. Octubre 1997.

[FSJ99] Fayad M., Schmidt, D., Johnson R. **Application Frameworks.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[GCRM98] García F., Contreras, J., Rodríguez, P., Moriyón, R. **Help generation for task based applications with HATS.** En Proceedings EHCI'98 (Engineering for Human Computer Interaction). Creta (Greece), September 1998.

[GHJV94] Gamma E., Helm R., Johnson R., Vlissides J. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addisson-Wesley, Reading, Mass., 1994.

[GM95] D.Gangopadhyay, S.Mitra. **Understanding Frameworks by Exploration of Exemplars.** En Proceedings of 7th. International Workshop on C.A.S.E. (CASE-95). IEEE Computer Society, Julio 1995.

[Gol83] Goldberg, A. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, 1983.

[Gon99] Gonzalez, P. **Proyecto OOFRA.** http://bogart.sip.ucm.es/pedro/

[HHG90] R.Helm, I.M.Holland, D.Gangopadhyay. **Contracts: specifying behavioral compositions in object-oriented systems.** En Proceedings of Conference

on Object - Oriented Programming, Languages and Applications (OOPSLA'90), ACM/SIGPLAN. New York, 1990.

[IBM99] IBM Corporation. **VisualAge for Java.** 1991-1999. www.software.ibm.com/software/ad/vajava

[JBR99] Jacobson I., Booch G., Rumbaugh J. **The Unified Software Development Process.** Addison-Wesley. Reading, Massachusetts. Enero 1999.

[JCJO92] Jacobson I., Christerson M., Jonsson P., Overgaard G. **Object Oriented Software Engineering: A Use Case Driven Approach.** ACM Press, 1992.

[JF88] Johnson R., Foot B. **Designing Reusable Classes.** Journal of Object Oriented Programming. New York, v.1, n.12, Diciembre 1988.

[JJW95] Johnson, P., Johnson, H. and Wilson, S. **Rapid Prototyping of User Interfaces driven by Task Models.** En Scenario-Based Design, Carrol J.M. (ed.). John Wiley and Sons, 1995.

[JLCH99] Jolin A., Lavin D., Charpenter S. **Visual Builders: Framework Design Issues.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[Joh92] Johnson R. **Documenting frameworks using patterns.** En Proceedings of Conference on Object-Oriented Programming, Languages and Applications (OOPSLA'92). ACM/SIGPLAN, New York, 1992.

[Joh93] Johnson R. **How to Design Frameworks.** En Tutorial Notes of Conference on Object-Oriented Programming, Languages and Applications (OOPSLA'93). Whashington DC, 1993

[Joh97] Johnson R. **Frameworks = Componentes + Patterns.** Communications of the ACM. v.40, n.20. Octubre 1997.

[Jol99] Jolin A. **Usability and Framework Desing.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[JWMP93] Johnson P., Wilson S., Markopoulos P., Pycock J.. **ADEPT, Advanced Design Environment for Prototyping with Task Models.** Proceedings of INTERCHI'93, ACM Press. 1993

[KP88] Krasner G., Pope S. **A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80.** Journal of Object-Oriented Programming, 1, 3 .1988

[Lea96] Leake, D. (Editor) **Case-Based Reasoning: Experiences, Lessons and Future Directions.** AIII Press / MIT Press. ISBN 0-262-62110-X. 1996

[LHM+90] Lindskov-Knudsen J., Hedin G., Magnusson B., Trigg R., Sörensen-Sandvad E. **Documenting Object Oriented Systems.** Proceedings of the Nordic Workshop on Programming Environments Research. Editores: Solberg, Venstrand. Norway, 1990.

[LK94] Lajoie R., Keller R. **Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert.** En Proceedings of the 62nd Congress of the ACFAS, Canadá, mayo 1994.

[LK98] Lauder A., Kent S. **Precise Visual Specification of Desing Patterns.** En Proceedings of European Conference on Object Oriented Programming (ECOOP 98). Lecture Notes in C. Science, Springer-Verlag, Berlín, 1998.

[Mat96] Mattsson, M. **Object-Oriented Frameworks – A survey of methodological issues.** Tesis presentada en la University College of Karlskrona/Ronneby, Department of Computer Science and Business Administration, Sweden. LU-CS-TR 96-167. 1996.

[MCK97] Meusel M, Czarnecki K., Kopf W. **A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext.** En Proceedings of European Conference on Object Oriented Programming (ECOOP'97). Lecture Notes in Computer Science, Springer Verlag, Berlín, 1997.

[ME96] Meijler T., Engel R. **Making Design Patterns explicit in FACE, a Framework Adaptive Composition Environment.** En Proceedings of First European Conference on Pattern Languages of Programming. Julio 1996.

[Mey92] Meyer B. **Applying Design by Dontract.** IEEE Computer, Octubre 1992

[MS99] Mili H., Sahraoui H. **Describing and Using Frameworks.** En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[OC00] Ortigosa A., Campo M. **Using Incremental Planning to Foster Application Framework Reuse.** A ser publicado en Journal of Software Engineering and Knowledge Engineering (JSEKE).

[OC98] Ortigosa A., Campo M. **An Adaptive Approach to Object-Oriented Application Framework Reuse.** Proceedings of European Reuse Workshop. Madrid, October 1998

[OC99] Ortigosa A., Campo M. **SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation.** Technology of Object-Oriented Languages and Systems 25, June 1999, IEEE Press. ISBN 0-7695-0275-x

[OCM99] Ortigosa A., Campo M., Moriyón R. **Enhancing Framework Usability through Smart Documentation.** Proceedings of the 3rd Argentine Symposium on Object Orientation. Buenos Aires, septiembre de 1999.

[OH98] Orfali R., Harkey D. **Client/Server Programming with Java and Corba.** John Wiley and Sons. New York, 1998

[Opd92] Opdyke W. **Refactoring Object Oriented Frameworks.** Ph.D. Thesis, Urbana-Champaign, University of Illinois, 1992.

[Ort95] Ortigosa A. Proposta de un Ambiente Adaptável de Apoio ao Processo de Desenvolvimento de Software. Dissertación de Maestría. UFRGS, Porto Alegre. 1995 (en portugués).

[Par79] Parnas, D. **Designing software for ease extension and contraction.** IEEE Transactions on Software Engineering, v.5, n.2, p. 128-137, Febrero 1979.

[Par94] ParcPlace Systems, Inc. **Visual Works 2.0 Manual.** 1994.

[Pat97] Paternò, F. **Understanding Task Model and User Interface Architecture Relationships.** CNUCE Internal Report, December 1997.

[Pat98] Paternò, F., Breedvelt-Schoutern, I.M. and Koning, N.M. **Deriving Presentations from Task Models.** En Proceedings of ECHI'98, Crete (Greece). 1998

[PP95] Pangoli S, Paternò F. **Automatic Generation of Task-oriented Help.** Proceedings of UIST'95. ACM Press. 1995

[PPSS95] Pree W., Pomberger G., Schappert A., Sommerlad P. **Active Guindance Of Framework Development**. En Software: Concepts and Tools, v.16, n.16, p. 94-103, Berlin, 1995

[Pre94] Pree, W. **Meta-Patterns: Abstracting the Essentials of Object-Oriented Frameowrks**. En Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'94). Springer-Verlag, Berlín, 1994.

[Pre95] Pree W. **Design Patterns for Object-Oriented Software Development**. Addison-Wesley Publishing Company. – ACM Press Books. ISBN 0-201-42294-8. 1995

[Pre95b] Pree W. **Framework Development and Reuse Support**. En Visual Object-Oriented Programming, Concepts and Environments. Burnett M., Goldberg A., Lewis T. (eds.). Prentice Hall, Manning, 1995.

[Pre96] Pree W. **Framework Patterns**. SIGS Books. New York, 1996

[Pree99] Pree W. **Hot-Spot-Driven Development**. En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[Pue97] Puerta, A. **A Model-Based Interface Development Environment**. IEEE Software Vol 14, Number 4. Julio / Agosto 1997.

[Rat97] Rational Inc. **UML Semantics**, version 1.1 Setiembre 1997. http://www.rational.com/uml

[RBP+91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. **Object-Oriented Modeling and Design**. Prentice Hall, 1991.

[RC93] Ross M., Carroll J. **Developing Minimalist Education for Object-Oriented Programming and Design**. Tutorial Notes, the 8th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'93). Washington, USA, 1993.

[RJB99] Rumbaugh J., Jacobson I., Booch G. **The Unified Modeling Language Reference Manual**. Addison-Wesley, 1999. ISBN 0-201-30998-X

[Sch97] Schmid H. **Systematic Framework Design by Generalization**. Communications of the ACM. v.40, n.20. Octubre 1997.

[Sch99] Schmid H. **Framework Design by Systematic Generalization**. En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[SG96] Shaw M., Garlan D. **Software Architecture – Perspectives on an Emerging Discipline**. Prentice-Hall, 1996. ISBN 0-13-182957-2

[Sou99] Soundarajan N. **Understanding Frameworks**. En Building Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (Eds.) John Wiley and Sons, N.Y, 1999.

[SSC95] Szekely, P., Sukaviriya, P., Castells, P., et al. **Declarative Interface Models for User Interface Construction Tools**. Proceedings of 6th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'95). Wyoming (USA), August 1995.

[SSP95] Schappert A., Sommerland P., Pree W. **Automated framework development**. En Symposium on Software Reusability (SSR'95), ACM Software Engineering Notes. Aug. 1995

[SP99] Szyperski C., Pfister C. **Compound User Interface Frameworks**. En

Implementing Application Frameworks. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999.

[VB94] Veloso M., Blythe J. **Linkability: Examining Causal Link Commitments in Partial-Order Planning**. En Proceedings of the Second International Conference on AI Planning Systems. Junio 1994.

[VS95] Veloso M., Stone P. **FLECS: Planning with a Flexible Commitment Strategy.** En Journal of Artificial Intelligence Research, 3. 1995

[WBJ90] Wirfs-Brock R., Johnson R. **Surveying Current Research in Object Design.** Communications of the ACM, New York, v.33, n.9, Septiembre 1990.

[Wel94] Weld D. **An Introduction to Least Commitment Planning**. AI Magazine, Summer/Fall 1994.

[Wel99] Weld D. **Recent Advances in AI Planning**. AI Magazine, 1999. To appear.

[ZK97] Zeiliger, R. and Kosbie, D. **Automating Tasks for Groups of Users: A System-Wide "Epiphyte" Approach** in *INTERACT'9.7* Howard S., Hammond J., Lyndgaard G. (eds.), Chapman & Hall Press, IFIP, Sydney. 1997

Reunido el tribunal que suscribe en el día
de la fecha, acordó calificar la presente Tesis
doctoral con Sobresaliente cum Laudem
Madrid, 29 de Mayo de 2000