Pontificia Universidad Católica de Valparaíso

Facultad de Ingeniería

Escuela de Ingeniería Informática

EVOLUCIÓN DE FRAMEWORKS DE CAJA BLANCA HACIA CAJA NEGRA BASÁNDOSE EN LENGUAJE DE PATRONES.

Autor:

Rubén Antonio Soto Pizarro

Informe final del Proyecto para optar al Título profesional de

Ingeniero Civil en Informática

Profesor guía:

Jorge Bozo Parraguez

Julio de 2006

A mi familia, por la paciencia y por el apoyo incondicional, y en especial a mi madre que nunca dejó de creer en mí.

Agradecimientos

Agradezco a Dios por las oportunidades que me da, porque en todo momento está a mi lado apoyándome, guiándome y siendo la luz en mi camino y por sentir su presencia en cada desafío que la vida me presenta.

A mi familia por su apoyo, por la paciencia, por creer y estar junto a mi en todo momento. A mi madre por su amor y por las enseñanzas entregadas las cuales son la base de mi vida.

A la Universidad por entregarme el conocimiento necesario para el desempeño ético y profesional dentro del mundo laboral y a los profesores que contribuyeron de manera directa en el desarrollo de este proyecto de título.

Resumen

Para mantener constantemente la satisfacción de los requerimientos de los dominios y las necesidades de reutilización de los componentes de software mediante frameworks es necesario que evolucionen desde caja blanca hacia caja gris. Considerando que los lenguajes de patrones son un evidente aporte a la reutilización de componentes, ya que cada patrón describe un problema que ocurre una y otra vez, para describir luego el núcleo de la solución a ese problema, es que se propone una guía detallada para evolucionar los frameworks de caja blanca hacia frameworks de caja gris, apoyando el proceso en los lenguajes de patrones.

Con el fin de elaborar la guía de evolución, se establece el marco teórico sobre el cual se debe desarrollar. Con el marco teórico definido detalladamente se especifica la guía que ayudará a los desarrolladores a evolucionar frameworks de cualquier dominio, la cual será validada mediante la aplicación de una serie de métricas de la Ingeniería de Software, demostrando el real aporte realizado a la reutilización mediante la aplicación de la guía propuesta. Una vez validada la guía se desarrolla una herramienta que asiste su aplicación en una de las etapas más relevantes. Finalmente se muestra como ejemplo parte de la utilización de la guía asistida mediante la herramienta desarrollada.

Finalmente nos podremos dar cuenta que el real aporte a la reutilización de la guía, aunque demostrado por la aplicación de métricas, no se podrá apreciar hasta instanciar el frameworks evolucionado para desarrollar una aplicación de algún dominio específico.

Palabras clave: Frameworks, lenguajes de patrones, caja blanca, caja gris, instanciación.

Abstract

In order to constantly maintain the satisfaction of the requirements of the dominions and the necessities of reusability of the software's components by means of frameworks it is necessary that they evolve from white box towards gray box. Considering that the patterns languages are an evident contribution to the reusability of components, since each pattern describes a problem that happens time and time again, to soon describe the nucleus of the solution to that problem, is that sets out a detailed guide to evolve frameworks of white box towards frameworks of gray box, supporting the process in patterns language.

With the purpose of elaborating the evolution guide, the theoretical frame settles down on which it is due to develop. With the defined theoretical frame in detail, the developed guide will help the developers to evolve frameworks of any dominion, which will be validated by means of the application of a series of Software Engineering metric, demonstrating the real contribution made to the reusability by means of the application of the propose guide. Once validated the guide a tool is developed that attends its application in one of the most excellent stages. Finally one is as example leaves from the use of the guide attended by means of the developed tool.

Finally we will be able to be given account that the real contribution to the reusability of the guide, although demonstrated by the application of metric, will not be able to be appreciated until we use frameworks evolved to develop an application of some specific dominion.

Keys: Frameworks, patterns languages, white box, gray box, instanciation.

Capítulo 1. Introducción.

1.1 Introducción.

La reusabilidad es un tema que surge prácticamente junto con la ingeniería de software, ya que es una estrategia que es utilizada por el ser humano para resolver la mayoría de los problemas a los que se ve enfrentado. Nuestro aprendizaje se basa en la experiencia, ya sea tanto nuestra, como de terceros, por ello, cuando enfrentamos algún problema, buscamos la forma de solucionarlo en experiencias pasadas tratando de adaptar las soluciones anteriores al problema actual, de lo contrario, debemos resolverlo utilizando nuestras competencias en el ámbito del problema, experiencia que quedará almacenada para problemas análogos en el futuro.

La reusabilidad de software juega un papel muy importante en temas tales como: productividad, capacidad de mantenimiento, portabilidad y calidad [1]. Se puede decir que la reutilización es un nuevo modelo de desarrollo de software [2] que requiere un entorno soportado por tecnología que aún no esta madura, y un modelo de desarrollo que explicite cuándo, cómo y dónde reutilizar, o cómo desarrollar el elemento para la reutilización.

En los años 80, la cada vez más popular programación orientada a objetos llegó a reforzar los conceptos de reusabilidad, ya que proporciona avances realmente significativos en este sentido, sin embargo, sólo sería una pequeña muestra consistente en partes de código usados en sistemas mayores, esto es conocido como reusabilidad de grano fino [7], para mejorar los niveles de reutilización se han propuesto técnicas basadas en elementos reutilizables de grano más grueso, como lo son los frameworks [8] y los patrones.

Según lo anterior, patrones, lenguaje de patrones y frameworks son formas de reutilización, como se específica en el trabajo “Un Proceso para la Construcción e Instanciación de Frameworks basados en un Lenguaje de Patrones para un Dominio Específico” [3], en la cual, además construye una herramienta capaz de instanciar frameworks basados en lenguaje de patrones. En sus trabajos, ha logrado relacionar de forma bastante fuerte a patrones, lenguaje de patrones y frameworks, consiguiendo una técnica poderosa para avanzar en la reutilización

de software. Estos trabajos dieron pie a la investigación que se describe en el presente documento.

En este capítulo se ofrece un resumen del marco teórico necesario para construir e instanciar frameworks de caja blanca y posteriormente poder evolucionarlos hacia frameworks de caja gris utilizando la guía propuesta en los capítulos posteriores. Además se definen los objetivos generales y específicos y la metodología utilizada para abordar cada uno de ellos.Los capítulos siguientes se estructuran de la siguiente manera: En el capítulo 2 se presenta la guía propuesta para evolucionar frameworks desde caja blanca hacia caja gris. Posteriormente, en el capítulo 3 se presentan las métricas de software y su aplicación para comprobar la validez de la propuesta. En el capítulo 4 se presenta el desarrollo de un protótipo de una herramienta software, la cual apoyará una de las etapas definidas en la guía propuesta. En el capítulo 5 se muestra un ejemplo de aplicación del protótipo desarrollado. Y en el capítulo 6 se presentan las principales conclusiones.

1.2 Patrones y Lenguajes de Patrones.

1.2.1 Antecedentes.

Los patrones son un método de resolución de problemas reciente en la Ingeniería de Software que ha emergido en mayor medida de la comunidad de orientación a objetos, aunque pueden ser aplicados en cualquier ámbito de la informática y las ciencias en general. Según Christopher Alexander [5], “cada patrón es una regla de 3 partes, que expresa una relación entre un contexto, un problema y una solución.

Cada patrón describe un problema que ocurre una y otra vez, para describir luego el núcleo de la solución a ese problema, de tal manera que pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma” [5]. Un patrón es, entonces, una solución útil, usable y usada, es decir reutilizable de forma eficiente.

De manera análoga a los patrones de Alexander, los patrones de software describen soluciones para los problemas que ocurren con frecuencia en el desarrollo de software, como una manera de capturar el conocimiento práctico alcanzado durante años por los desarrolladores [3]. Podemos pensar en los patrones de software como pares

problema/solución, los cuales pueden ser agrupados en familias de problemas y soluciones similares. Por lo tanto, un lenguaje de patrones es una colección estructurada de patrones que se apoyan unos con otros para transformar requisitos y restricciones en una arquitectura [6].

1.2.2 Acerca de los Patrones de Software.

Un patrón es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún dominio. Si bien la teoría original de los patrones aplica a la construcción de casas y planeamiento urbanístico, también puede ser aplicada al software. Los patrones de software facilitan la reutilización del diseño y de la arquitectura, capturando las estructuras estáticas y dinámicas de colaboración de soluciones exitosas a problemas que surgen al construir aplicaciones.

En 1987, Beck y Cunningham utilizaron las ideas de Alexander en el desarrollo de un pequeño lenguaje de patrones para guiar programadores inexpertos en Smalltalk [12]. Pero es con el trabajo de Gamma, Helm, Johnson, Vlissides, "Design Patterns", en 1995, subtitulado "Elementos de software orientado a objetos reutilizables", cuando el tema empieza a madurar. Estos autores, conocidos como "GoF" ("Gang of Four", la banda de los cuatro), son los que se dan el trabajo de hacer un catálogo de los patrones de diseño que hasta ese momento habían aparecido, y en una lista de 23 patrones, tratan de enumerar el conocimiento acumulado sobre el tema. Es con estos trabajos, que se puede decir, que los patrones de software describen soluciones para los problemas que ocurren con frecuencia en el desarrollo de software, como una manera de capturar el conocimiento práctico alcanzado durante años por los desarrolladores [3]. Podemos pensar en los patrones de software como pares problema/solución, los cuales pueden ser agrupados en familias de problemas y soluciones similares. Así mismo, las agrupaciones de patrones según su nivel de abstracción los podemos reunir en colecciones, catálogos, sistemas y lenguajes. Por lo tanto, un lenguaje de patrones es una colección estructurada de patrones que se apoyan unos con otros para transformar requisitos y restricciones en una arquitectura [6].

Según "GoF", un patrón tendrá cuatro elementos esenciales [13]: Nombre, problema, solución y consecuencias.

- **El nombre**: Este elemento, que alguien podría pensar que es intrascendente, o trivial, ha probado ser fundamental. A cada patrón, se le ha asignado una denominación que permite que los entendidos en el tema, puedan conversar usando un diccionario común. Esto permite un mayor grado de abstracción. Uno de los problemas que tuvo GoF, fue encontrar un nombre apropiado para cada patrón que catalogaron.

- **El problema**: Acá se explica el problema original, y su contexto. Puede describir desde detalles específicos, como algoritmos, o clases y estructuras que se han encontrado inflexibles a la hora de implementarse. Pero primero, todo patrón nace de un problema a solucionar.

- **La solución**: Cada patrón es una solución a un problema, el elemento planteado arriba. Hasta puede que un mismo problema real, tenga dos soluciones parecidas, correspondientes a dos patrones. Pero la elección recaerá en el patrón que mejor se adapte al contexto particular del problema. La solución que un patrón describe, no necesariamente es detallada al nivel de implementación, sino que provee una descripción abstracta, una enumeración de elementos y sus relaciones, para solucionar el problema planteado.

- **Las consecuencias**: Son los resultados de aplicar el patrón, compromisos, que se tienen que aceptar al adoptar el mismo. En general, en software, la moneda de pago es el espacio y el tiempo. A veces, un patrón nos soluciona un tema de espacio, a costa de una mayor complejidad en otra punta de nuestro desarrollo.

Es importante destacar a esta altura que un patrón siempre tiene un problema, y una solución. Pero también, que los lenguajes de patrones que han aparecido, destacan que los elementos clasificados, siempre son patrones que ya se han aplicado exitosamente con anterioridad. No se ha dado el caso de un patrón creado de la nada, o simplemente para engrosar el lenguaje. Cada patrón que aparece en la literatura, se verá respaldado por aplicaciones anteriores. El trabajo del catalogador, ha sido descubrir la esencia del patrón, para poder describirlo en un nivel mayor de abstracción, y aprovechar así su poder en otros ámbitos y contextos parecidos.

Los patrones pueden describirse de distintas maneras, en este trabajo me basaré en el formato de Gamma, el cual contiene los siguientes elementos: nombre y clasificación, intención, otros nombres, motivación, aplicación, estructura, participantes, colaboraciones, consecuencias, implementación, código de ejemplo, usos conocidos y patrones relacionados.

- **Nombre del patrón y clasificación:** Cada patrón tiene un nombre, y una categoría a la que pertenece. El GoF los clasifica en creacionales, estructurales y de conducta. Sobre esto se tratará más adelante.

- **Intención:** Todo esta dentro de un contexto. Cada patrón tiene una intención, una razón, una justificación.

- **Otros nombres:** O el A.K.A. ("Alsa Known As") del patrón. Ahora menos usado, pero en su tiempo, los patrones habían surgido en distintos proyectos y tecnologías. Los primeros autores les dieron un nombre que no siempre coincidía. Estos otros nombres pueden ser enumerados en la descripción del patrón.

- **Motivación:** Más que la intención, esta parte describe un escenario: un problema en particular que aparece, y que ayuda a entender la descripción algo más abstracta del patrón genérico.

- **Aplicación:** En qué situaciones puede ser aplicado un patrón. Pueden darse ejemplos de malos diseños, que pueden beneficiarse de la aplicación de este patrón en particular.

- **Estructura:** La parte más conocida, luego del nombre. Se adopta un diagrama basado en UML ("Unified Modeling Language").

- **Participantes:** La lista de las clases y objetos que participan del patrón de diseño, y las responsabilidades que tienen.

- **Colaboraciones:** Descripción de cómo los participantes colaborar para llevar a cabo sus responsabilidades en el patrón.

- **Consecuencias:** Explica cómo el patrón cumple con sus objetivos, y que compromisos se asumen.

- **Implementación:** Parte más variable, porque para cada patrón puede haber varias posibles implementaciones, incluso, diferentes implementaciones según la tecnología adoptada.

- **Código de ejemplo:** Siempre se da el código de ejemplo de cada patrón, por ejemplo, en Smalltalk o en C++. También han aparecido libros con patrones aplicados a Java, y .Net.

- **Usos conocidos:** Un patrón no es tal, si no ha sido ya empleado en algún caso real. Se debe incluir por lo menos dos ejemplos conocidos de diferentes ámbitos.

- **Patrones relacionados:** Ya se ha mencionado que un patrón es una solución a un problema. Puede que haya problemas parecidos, que tengan más de una solución. De ahí que muchos patrones estén relacionados entre sí. Se trata de explicar acá cuáles son las similitudes y las diferencias, entre patrones relacionados.

Considerando el tipo de patrón, según "GoF", los patrones se clasifican en: creacionales, estructurales y de conducta.

Patrones Creacionales: Abstraen el proceso de instanciación. Ayudan a independizar a un sistema, de cómo sus objetos son creados. En general, tratan de ocultar las clases y métodos concretos de creación, de tal forma que al variar su implementación, no se vea afectado el resto del sistema.

Es común encontrar "competencia" entre estos patrones: hay más de un patrón a adoptar ante una situación.

Abstract Factory: Nos da una interfaz para crear objetos de alguna familia, sin especificar la clase en concreto.

Builder: Separa la construcción de un objeto complejo, de su representación. De esa manera, el mismo proceso de construcción puede crear diferentes resultados.

Factory Method: Se define una interfaz para crear objetos, como en el Abstract Factory, pero se delega a las subclases implementar la creación en concreto.

Prototype: Mediante una instancia prototipo, conseguimos otras instancias de ese objeto.

Singleton: Consigue dar un solo objeto de la clase, en cualquier momento de la aplicación.

Una clase de la cual solamente una sola instancia puede existir

Patrones Estructurales: Se ocupan de cómo clases y objetos se agrupan, para formar estructuras más grandes.

Adapter: Permite convertir una interfaz de una clase, en otra, que es la esperada por algún cliente.

Bridge: Desacopla una abstracción de su implementación en concreto. Luego, es posible cambiar la implementación, o la abstracción, sin cambiar la otra.

Composite: Compone objetos en una estructura de árbol, donde los objetos compuestos se tratan de forma similar a los objetos simples.

Decorator: Agrega responsabilidad a un objeto dinámicamente, dando una alternativa a la extensión de una clase, en lugar de usar subclases.

Facade: Provee una interfaz unificada a un conjunto de funciones de un subsistema. Es una interfaz de alto nivel, para facilitar el uso del subsistema.

Flyweight: Permite compartir objetos, sin repetirlos en el sistema, eficientemente.

Proxy: Provee un subrogado a otro objeto, para controlar el acceso al mismo.

Patrones de Conducta: Más que describir objetos o clases, describe la comunicación entre ellos. Frecuentemente, describen las colaboraciones entre distintos elementos, para conseguir un objetivo.

Chain of Responsability: Desacopla el emisor de un mensaje, de su receptor, permitiendo que haya varios objetos que tengan la oportunidad de manejar el requerimiento. Esto se consigue pasando el requerimiento por la cadena de objetos hasta llegar al encargado de atenderlo.

Command: Encapsula el requerimiento a un objeto, permitiendo incluso el "undo" de la operación.

Interpreter: Construye una representación de la gramática de un lenguaje, junto con su intérprete.

Iterator: Da un modo de acceder a los elementos de un objeto colección, sin exponer su estructura interna.

Mediator: Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones.

Memento: Sin necesitar entrar en la estructura interna de un objeto, permite capturar su estado, para, por ejemplo, poder restaurarlo más adelante.

Observer: Define una relación uno a muchos, entre un objetos y otros que están interesados en sus cambios, nuevamente, sin caer en el acople entre los mismos.

State: Permite a un objeto cambiar su conducta cuando cambia su estado interno, simulando que cambia de clase.

Strategy: Define una familia de algoritmos, y los hace intercambiables.

Template Method: Define el esqueleto de una operación, cuyas operaciones más básicas, quedan delegadas en subclases.

Visitor: Permite recorrer una estructura (un árbol, por ejemplo), aplicando una operación a cada elemento.

1.3 Frameworks orientados a objetos.

1.3.1 Antecedentes.

Un framework orientado a objeto es esencialmente un conjunto de clases abstractas y concretas que colaboran de una manera exacta para proporcionar un marco común en el cual una gran gama de aplicaciones puedan ser construidas. Los frameworks orientados a objeto han surgido en el contexto de la reutilización de software. Ellos permiten la reutilización de grandes estructuras en un dominio particular, que se pueden modificar para lograr aplicaciones específicas en algún dominio. Familias de aplicaciones similares pero no idénticas se pueden derivar de un solo framework [4]. Si consideramos la forma de reutilización que el framework utiliza, existen tres tipos: caja blanca, caja gris y caja negra.

Los frameworks de caja blanca pueden evolucionar para tornarse cada vez más caja negra [11]. Esto puede ser hecho de forma gradual, implementándose varias alternativas que después son aprovechadas en la instanciación del framework. Por lo anterior, en este trabajo se propondrá una guía detallada para la evolución de frameworks de caja blanca hacia frameworks de caja gris basándose en lenguaje patrones.

1.3.2 Acerca de los Frameworks orientados a objetos.

Existen diversas definiciones para frameworks orientados a objetos (frameworks, de acá en adelante) en la literatura, pero en todas ellas encontramos un común denominador, el cual es facilitar la reutilización [14]. Con este motivo, los frameworks se han utilizado en desarrollo de aplicaciones de dominio específico por muchos años. Los primeros frameworks surgieron en el dominio de la interfaz gráfica de usuarios, como es el caso de frameworks MacApp, de Macintosh; Andrew Toolkit, de la Universidad de Carnegie Mellon; ET++, de la Universidad de Zurik; e InterViews, de Stanford. Pero, luego comenzaron a surgir frameworks para otros

dominios, como para algoritmos de ruteo VLSI (*very large scale integration*), sistemas hipermedia, sistemas operacionales y control de manufactura.

Actualmente existen frameworks largamente utilizados, como OLE (Object Linking and Embedding), DSOM (Distributed System Object Model). El surgimiento del lenguaje Java permitió el desarrollo de nuevos frameworks como AWT (Abstract Windows Toolkit), que forma parte de JFC (Java Foundation Classes) y Java Beans [3].

Como ya se puede apreciar, los frameworks se han convertido en la piedra angular de la ingeniería de software moderna, esto se debe a que son aceptados como elementos de reutilización de grano grueso, ya que permiten la generación de aplicaciones que se relacionan directamente con un dominio particular. El framework captura las decisiones de diseño comunes a un tipo de aplicaciones, estableciendo un modelo común a todas ellas, asignando responsabilidades y estableciendo colaboraciones entre las clases que forman el modelo. Además, este modelo común contiene puntos variables (hot spots o puntos calientes), capaces de albergar los distintos comportamientos de las aplicaciones representadas por el framework. La reutilización, por lo tanto, se produce en el proceso de instanciación del framework, rellenando los puntos variables, en los que el desarrollador incluye la funcionalidad específica de su sistema. Los frameworks pueden ser clasificados considerando su finalidad, es decir, su aplicación en un dominio dado, y también pueden clasificarse de acuerdo a la forma de reutilización del frameworks para generar aplicaciones nuevas. Considerando la finalidad del framework se pueden clasificar en dos grupos: frameworks de aplicación y frameworks de dominio.

Un framework de aplicación encapsula una capa de funcionalidad horizontal que puede ser aplicada en la construcción de una gran variedad de programas. Además de los frameworks que implementan las interfaces gráficas de usuario, que representan su paradigma, se pueden clasificar en esta categoría los dedicados al establecimiento de comunicaciones o procesamiento de documentos XML entre otros. Por otra parte, un framework de dominio implementa una capa de funcionalidad vertical, correspondiéndose con un dominio de aplicación o una línea de producto. Estos deberán ser los más numerosos, y su evolución deberá ser también la más rápida, pues deben adaptarse a las áreas de negocio para las que están diseñados. Por esto, este trabajo se basará en la evolución de frameworks de dominio.

Considerando la forma de reutilización, existen tres tipos: frameworks de caja blanca, de caja gris y de caja negra. Los frameworks de caja blanca se extienden mediante herencia, concretamente el desarrollador debe implementar subclases y métodos a partir de las clases y métodos abstractos definidos como puntos variables o puntos calientes (hot spots). Se tiene acceso al código del framework y se permite reutilizar la funcionalidad de sus clases mediante herencia y redefinición de métodos. Estos frameworks son más fáciles de desarrollar pero la exposición del código a los desarrolladores puede causar problemas y además exigen estar relacionados con la implementación [10].

Los frameworks de caja negra se extienden mediante composición, en lugar de utilizar la herencia. Los puntos variables se definen por medio de interfaces que deben implementar los componentes que extiendan el framework, el desarrollador debe combinar las diferentes clases concretas existentes para obtener aplicaciones específicas. Estos frameworks son más abstractos y menos flexibles, lo que hace más difícil la mantención de las aplicaciones derivadas, pero evita la dependencia de la implementación del framework y los clientes.

Los frameworks de caja gris son una mezcla entre frameworks de caja blanca y frameworks de caja negra, por lo tanto su extensión se realiza por medio de herencia, por unión dinámica y también por las interfaces que definen los puntos variables. Estos frameworks, si están bien diseñados, superan las barreras de los otros dos tipos, ya que son más flexibles y fáciles de extender, sin mostrar innecesariamente información no relevante a los desarrolladores. Aunque el nombre de caja gris es menos conocido que los otros, es el que describe la forma de extensión más utilizada.

Análisis de frameworks de caja blanca.

Los principales obstáculos para implantar un modelo de reutilización basado en frameworks vienen dados porque estos son difíciles de construir y no es sencillo aprender a utilizarlos, y estas dificultades se agudizan en el caso de los frameworks de dominio. Las dificultades en la creación de frameworks hacen que, si no se prevee utilizarlos mucho, sea difícil amortizar el coste de su producción. Por ello se estima que las situaciones en que es económicamente rentable afrontar la construcción de un framework son aquellas en que:

Se dispone de varias aplicaciones del mismo dominio (o se deben producir en breve plazo).

Se espera que se requieran nuevas aplicaciones del dominio con cierta frecuencia.

También se admite como cierto que el framework no puede ser considerado como un producto final, sino que debe ser esencialmente evolutivo. Por esa razón se propone una estrategia que se inicia con la construcción de un primer framework de caja blanca que, con la información proporcionada por las sucesivas instanciaciones, va refinándose y transformándose en un framework de caja gris para una posterior transformación a frameworks de caja negra.

Con el fin de entender todos los aspectos relacionados a la construcción e instanciación de un frameworks de caja blanca para su posterior evolución hacia frameworks de caja gris, es importante conocer los conceptos de "método-gancho" y "método-plantilla" (en inglés, *hook-method* y *template-method*, respectivamente), los cuales describen como funciona la reutilización de frameworks de caja blanca. Un método-gancho es definido en una clase abstracta del framework y su implementación es provista por las sub-clases concretas. Las clases abstractas de un framework de caja blanca poseen métodos-plantilla con llamadas a uno o más métodos-gancho. Los métodos-plantilla no necesitan ser traslapados por las clases concretas de aplicación, ya que su lógica no cambia. Los métodos-plantillas tienen como propósito permitir la flexibilidad en frameworks y en software orientados a objetos en general.

La presencia de frameworks influencia el proceso de desarrollo de software. Por ejemplo, desarrollar un sistema contable teniendo un framework para construcción de aplicaciones contables es diferente a desarrollar el mismo sistema partiendo desde cero. El desarrollo de software basado en frameworks posee tres etapas [14]: una etapa de desarrollo, en la cual el framework es construido y testeado; una etapa de uso, en la cual el framework es instanciado innumerables veces para aplicaciones específicas; y una etapa de evolución y mantención, en cual el framework evoluciona para incorporar nuevas funcionalidades del dominio. Las etapas de desarrollo y uso son explicadas en los capítulos siguientes y la etapa de evolución es en la que se centra este trabajo. La mantención del framework no se incluye en este trabajo, ya que la mantención no es tan fácil como puede parecer en un principio, debido a que las aplicaciones derivadas de un framework deben sufrir la misma mantención para continuar la compatibilidad con el framework, de lo contrario, el framework deberá ser divido en dos, lo que puede acarrear problemas.

1.4 Comparaciones entre formas de reutilización.

1.4.1 Patrones v/s Frameworks.

Un patrón es una guía para solucionar un problema, en cambio, un framework es una colección de clases que, reunidas, resuelven un problema concreto. El patrón dice cómo resolver un problema, mientras que el framework aporta una solución. Los patrones son más abstractos que los frameworks, porque un framework incluye código ejecutable, en cuanto un patrón representa conocimiento y experiencia sobre software, por lo que se puede decir que los patrones tienen naturaleza lógica y los frameworks naturaleza física.

En general, los patrones son compuestos por dos o tres clases, en cuanto los frameworks envuelven un número bastante mayor de clases, pudiendo englobar diversos patrones [13].

1.4.2 Lenguaje de Patrones v/s Frameworks.

Existe una interrelación estricta entre lenguaje de patrones y frameworks. Primeramente, ambos son concebidos para un dominio específico y resuelven la mayoría de los problemas comunes a aplicaciones de tal dominio. En segundo lugar, lenguajes de patrones generan frameworks, pues ellos contienen las principales abstracciones encontradas en el dominio de aplicación, las cuales dan origen a los componentes de alto nivel de framework.

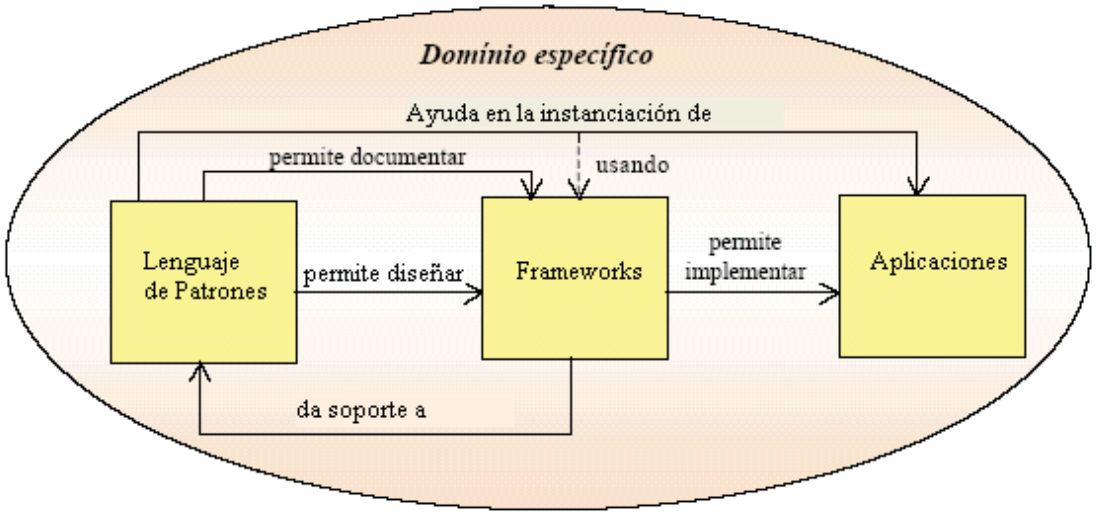
Los lenguajes de patrones pueden ser utilizados para documentar un framework, mostrar como utilizarlo, sin mostrar como funciona o como fue diseñado y además los lenguajes de patrones se pueden organizar en una secuencia de decisiones que generen el diseño de un framework [15]. En otras palabras, los lenguajes de patrones ayudan a los desarrolladores a alcanzar un nivel de entendimiento necesario para extender un framework.

Un framework permite implementar una aplicación diseñada según lenguaje de patrones, por lo tanto, contiene una implementación de todos los patrones que componen una aplicación específica. Al mismo tiempo, un lenguaje de patrones ofrece las reglas para el uso de los elementos del framework y para su extensión.

1.4.3 Lenguajes de Patrones v/s Frameworks v/s Aplicaciones.

La figura 1.1 muestra la relación entre lenguajes de patrones, frameworks y aplicaciones, además resume las ideas descritas anteriormente. Los tres se encuentran en un dominio específico. Los lenguajes de patrones permiten documentar y diseñar un framework. También ayudan en la instanciación de aplicaciones, que pueden ser implementadas usando un framework. Los frameworks permiten implementar aplicaciones y da soporte a lenguajes de patrones, implementando los patrones en él contenidos.

Figura 1.1. Relación entre Lenguaje de Patrones, Frameworks y Aplicaciones.



1.5 Construcción e instanciación de frameworks de caja blanca a partir de un lenguaje de patrones.

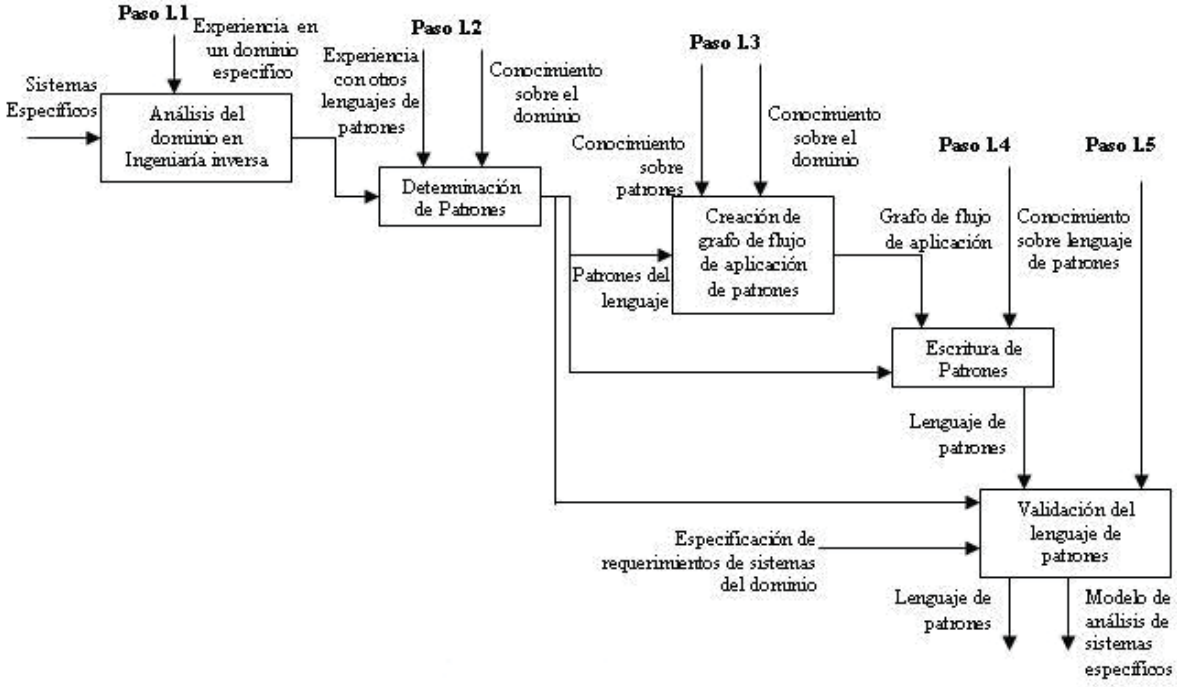
1.5.1 Consideraciones iniciales.

En esta sección se presenta el proceso general para construir e instanciar frameworks de caja blanca a partir de un lenguaje de patrones propuesto por los doctores Braga y Masiero [3]. El proceso general cuenta de cuatro pasos: construcción del lenguaje de patrones, construcción del framework de caja blanca en base al lenguaje de patrones construido, construcción de una herramienta para instanciar frameworks de caja blanca y, por último, instanciación del framework de forma manual y utilizando la herramienta que automatiza el proceso.

1.5.2 Proceso de construcción del Lenguaje de Patrones.

El primer paso del proceso general presentado consiste en el sub-proceso de construcción del lenguaje de patrones para un dominio específico. Cabe destacar que el lenguaje de patrones será construido en base a patrones de análisis, cada cual contendrá uno o más diagramas de clases que ilustren la(s) solución(es) para un problema en cuestión. La figura 1.2 muestra los pasos necesarios para la construcción de un lenguaje de patrones, los que son presentados en las siguientes secciones.

Figura 1.2. Proceso de construcción de un lenguaje de patrones.



Análisis del dominio en ingeniería inversa.

El objetivo de la ingeniería inversa es obtener información técnica a partir de un producto accesible al público, con el fin de determinar de qué está hecho, qué lo hace funcionar y cómo fue fabricado. Este método es denominado ingeniería inversa porque avanza en dirección opuesta a las tareas habituales de ingeniería, que consisten en utilizar datos técnicos para elaborar un producto determinado.

Por lo tanto, para construir un lenguaje de patrones que incluye aplicaciones en un dominio dado, es necesario observar las soluciones que comúnmente emplea para resolver problemas

recurrentes de ese dominio. Así mismo, el punto de partida para la creación de un lenguaje de patrones es obtener un modelo que represente el dominio puro, es decir, que capture las funcionalidades presentes en la gran mayoría de la aplicaciones del dominio.

Una de las maneras de conseguir reunir información sobre las funcionalidades básicas del dominio es por medio de la ingeniería inversa de algunos sistemas ya existentes. Para ello, es necesario que la ingeniería inversa produzca modelos intermedios representando cada uno de los sistemas, para después avanzar hacia un proceso de generalización de los modelos que culminará con un modelo de análisis del dominio [16].

Independientemente de la forma utilizada para conseguir la información sobre el dominio, el resultado de este paso es un modelo de análisis de dominio, constituido por un modelo estático y un modelo dinámico. El modelo estático puede ser expresado usando una notación orientada a objetos, como por ejemplo un modelo de clases de UML, donde cada clase posee como atributos y métodos solamente aquellos que fueran comunes al dominio. Y el modelo dinámico puede ser expresado, por ejemplo, por el diagrama de secuencia de UML, que muestra como funciona la comunicación entre los objetos para implementar las operaciones del sistema.

Determinación de Patrones.

El modelo de análisis de dominio, resultante del paso anterior, es utilizado para definir los patrones del lenguaje de patrones (paso 1.2). Esta actividad depende bastante del conocimiento y experiencia sobre patrones de desarrollo de lenguaje de patrones. Entre tanto, algunas recomendaciones deben ser seguidas para que los patrones sean definidos de forma uniforme y con mayor posibilidad de reutilización:

1. Los patrones de análisis existentes en la literatura deben ser analizados. Es probable que algunos de ellos estén presentes en el modelo de análisis de dominio tratado. En caso de ser encontrados esos patrones, el problema resuelto por ellos debe ser especializado para el caso de nuestro dominio y los patrones deben recibir un nuevo nombre que refleje el problema específico de dominio.
2. Otros lenguajes de patrones de análisis, para dominios similares, deben ser estudiados, observando cuales son sus patrones constituyentes, como están documentados y como se relacionan entre si. Esto contribuye al conocimiento sobre patrones y aumenta las posibilidades del desarrollador de lenguaje de patrones de definir de forma correcta y más fácil de reutilizar por otros usuarios de lenguajes de patrones. En este momento, se puede

optar por un formato para los patrones, o por lo menos identificar los dos o tres formatos más apropiados.

3. La determinación de los patrones debe iniciar por el análisis de las clases básicas del modelo de dominio obtenido en el paso anterior. Clases básicas son aquellas que están involucradas en las funciones básicas o principales representadas en el modelo de dominio, es decir, aquellas que están presentes en todos los sistemas de ese dominio.
4. Las clases básicas deben ser estudiadas en busca de grupos de dos o más clases que sean responsables de las funciones importantes del sistema. Esto puede ser hecho en el propio modelo de clases del dominio, por ejemplo, destacándolas con un color diferente, o creando modelos menores referentes a cada función. Estos grupos de clases básicas representaran los principales patrones del lenguaje, ya que cada patrón debe referirse a una función específica realizada en el dominio. Esto facilita la reutilización, ya que conduce a patrones como problemas/soluciones escritos y objetivos.
5. Las clases complementarias, son diferentes de las clases básicas, ya que agregan mejoras en relación a un patrón, o agregan una función diferente en relación a las funciones básicas del modelo de dominio. Las clases complementarias generalmente representan funcionalidades que son opcionales para el funcionamiento del sistema del dominio estudiado. Así, las clases complementarias tienen mayores posibilidades de ser patrones opcionales, o de estar junto a patrones ya existentes en el lenguaje, formando variables de patrones principales al ir convirtiendo elementos opcionales en patrones principales. Nuevamente, se deja a cargo del analista la decisión de como establecer cuales clases forman parte de un patrón. La existencia de patrones opcionales permite que ellos sean ignorados durante el uso del lenguaje de patrones, en caso de que las funcionalidades por ellos cubierta no fuera necesaria en la aplicación específica.
6. Cada patrón debe recibir un nombre que debe abstraer el contenido del patrón, facilitando su identificación y utilización por otros analistas. Una forma de facilitar la nominación de los patrones es por medio de la observación de sus funciones. El nombre escogido debe ser significativo o suficiente para que su simple citación traiga consigo el valor semántico implicado en el problema que el patrón resuelve. Pueden ser utilizadas metáforas o expresiones que sean conocidas en el dominio cubierto por el lenguaje de patrones.
7. Después de identificar y nombrar los patrones, se puede elaborar una tabla conteniendo los nombres de los patrones, el problema resuelto y una síntesis de la solución propuesta. Esta tabla ayuda al desarrollador de lenguajes de patrones a mantenerse fiel a sus objetivos para cada uno de los patrones que serán utilizados tanto para la creación del grafo del lenguaje de patrones, como durante la etapa de escritura de los patrones.

Creación de un grafo de flujo de aplicación de los patrones.

El tercer paso para la construcción del lenguaje de patrones es un análisis de la interacción entre los diversos patrones que lo componen. Se debe establecer un orden en el cual los patrones deben ser aplicados, de forma de facilitar el modelamiento de los sistemas usando el lenguaje de patrones. Por otra parte, se deben mostrar cuales son los patrones principales y opcionales del lenguaje, informaciones que son obtenidas por medio de la descripción general del patrón. Es importante precisar que el grafo debe indicar la interacción entre los patrones,

mas no tienen la intención de mostrar como es el orden de funcionamiento del sistema resultante.

Una de las estrategias para crear el grafo es empezar por los patrones que representan las funcionalidades más básicas del dominio, y agregar, de forma gradual, los patrones que correspondan a las funcionalidades más específicas, terminando con los patrones que son opcionales, es decir, los patrones que representan problemas no siempre encontrados en aplicaciones del dominio. Entre tanto, se debe evaluar con cuidado el mejor lugar para posicionar los patrones opcionales, ya que muchas veces dependen de la aplicación de un patrón obligatorio y deben ser aplicados luego en esa secuencia. Además de elaborar un diagrama o grafo para ilustrar las interacciones entre los patrones, es deseable repetir esa información sobre las interacciones por medio de una sección en cada patrón denominada "Próximos Patrones". La intención principal es guiar al usuario del lenguaje de patrones respecto de los próximos patrones que debe considerar después de aplicar un determinado patrón.

El flujo de aplicación de los patrones, además de ser influenciado por los patrones opcionales, también puede ser influenciado por otros elementos de los patrones, como por ejemplo, las variantes, las clases y los elementos opcionales. Puede ocurrir que la aplicación de un determinado patrón implica una adición de elementos en otros patrones, otros casos en que la aplicación de un patrón exige la aplicación de otro patrón o, incluso casos en que un patrón puede ser aplicado solamente si otro patrón fue aplicado también. Dependiendo de la notación utilizada para representar el grafo, tal vez esas situaciones no consigan ser mostradas directamente en el grafo, siendo entonces documentadas en la sección “Próximos Patrones” de cada patrón.

Escritura de patrones.

En este paso el analista debe establecer una forma de describir los patrones. Existen varios formatos sugeridos por distintos autores, pero independiente del formato adoptado, existen algunos componentes básicos del patrón que deben estar presentes, estos son: nombre, contexto, problema, fuerzas y solución.

- *Nombre:* debe estar relacionado con la intención del patrón.

- *Contexto:* debe describir como y cuando el patrón puede se aplicado.

- *Problema:* debe describir, de manera directa, el problema que el patrón se propone resolver, frente al contexto anteriormente especificado.

- *Fuerzas:* debe mostrar cuales son los factores que, frente al contexto especificado y al problema establecido, influencian la solución propuesta y como ella es implementada. También puede presentar limitaciones y restricciones de la solución, desventajas en utilizar el patrón o porqué la solución no utiliza otra estrategia.

- *Solución:* puede estar escrito de varias formas, pero debe presentar de manera clara la solución propuesta por el patrón. Pueden ser creadas ilustraciones de la solución, como sub-secciones para describir los participantes de la solución así como la colaboración entre ellos.

Escogido el formato, se deben escribir los patrones basándose en la descripción general de los patrones y las clases que los componen obtenidas en el paso 1.2. Cabe destacar que un patrón es mucho más que una estructura de clases y una descripción sobre ellas, ellos nos presentan básicamente todas las informaciones del contexto en el cual pueden ser aplicados, el problema que resuelven, así como las fuerzas que actúan para formar soluciones [17].

Validación del lenguaje de patrones.

La validación (paso 1.5) finaliza el proceso de construcción de un lenguaje de patrones. Esto puede ser hecho por medio de su aplicación a diferentes sistemas del dominio estudiado, siguiendo el proceso de utilización propuesto en el punto siguiente. Básicamente, esta actividad está constituida por el estudio del documento de requisitos de la aplicación a ser modelada, estudio de la aplicación del lenguaje de patrones como base en el documento de requisitos y evaluación del modelo de clases, confrontando los requisitos deseados con los modelos de clases de la aplicación. Cuantas más aplicaciones fueran modeladas con el

lenguaje de patrones, mejor será su validación, además de contribuir con la mejora del propio lenguaje, ya que nuevas características pueden ser incorporadas.

1.5.3 Proceso de utilización de un lenguaje de patrones.

En general, los lenguajes de patrones de análisis son auto-explicativos, ya que poseen todos los elementos necesarios para ser utilizados por desarrolladores inexpertos. Sin embargo, a continuación se muestran algunos pasos que deben ser seguidos al utilizar un lenguaje de patrones para asegurar que sea sacado el máximo provecho de las ventajas que ofrecen.

1. Estudie el lenguaje de patrones detalladamente, para conocer cual es el dominio por el tratado, cuales son los patrones disponibles y cuando aplicarlos.
2. Produzca el documento de requisitos del sistema específico, siguiendo las recomendaciones de la ingeniería de software. Tal documento debe describir todas las funcionalidades deseadas para el sistema a ser desarrollado.
3. Lea con atención el documento producido en el paso 2, para tener un conocimiento amplio de los requisitos del sistema, necesarios para decidir si el lenguaje de patrones de análisis y el respectivo framework, en caso de que exista, pueden ser utilizados en la construcción del sistema.
4. Use el lenguaje de patrones de análisis para modelar el sistema específico, empleando las informaciones presentes en los diversos elementos de cada patrón para decidir si ellos son aplicables o no.
5. Paca cada patrón aplicable esboce el diagrama referente a la porción del sistema que utilizará tal patrón, usando colores o símbolos diferentes para destacar posibles atributos, métodos u operaciones agregadas a los patrones. Estos diagramas de clases crecen de manera gradual, a medida que nuevos patrones son aplicados.
6. Para cada patrón aplicable haga una marcación en el documento de requerimientos para indicar cuales de los requerimientos fueron atendidos por la aplicación de tal patrón. Para cada patrón aplicable utilice textos explicativos para explicar los papeles desempeñados por cada clase en tal patrón.
7. Para cada patrón aplicable anote el nombre del patrón utilizado y su variante o sub-patrón (si es que tiene).
8. Si el patrón posee un elemento "próximo patrón" (o equivalente), entonces este indica los posibles patrones a investigar después de ser aplicado (o no) el patrón actual. Esto define diversos caminos a seguir, que deben ser anotados por el desarrollador e investigados uno a uno.
9. Revise el documento de requerimientos en busca de requerimientos no atendidos o atendidos parcialmente por el lenguaje de patrones. Complemente el diagrama de clases con tales requerimientos, por medio de la adición de nuevos atributos, clases, relaciones, métodos u operaciones, usando colores diferentes y haciendo anotaciones en el documento de requerimientos que identifiquen la no cobertura de los requerimientos por el lenguaje de patrones.

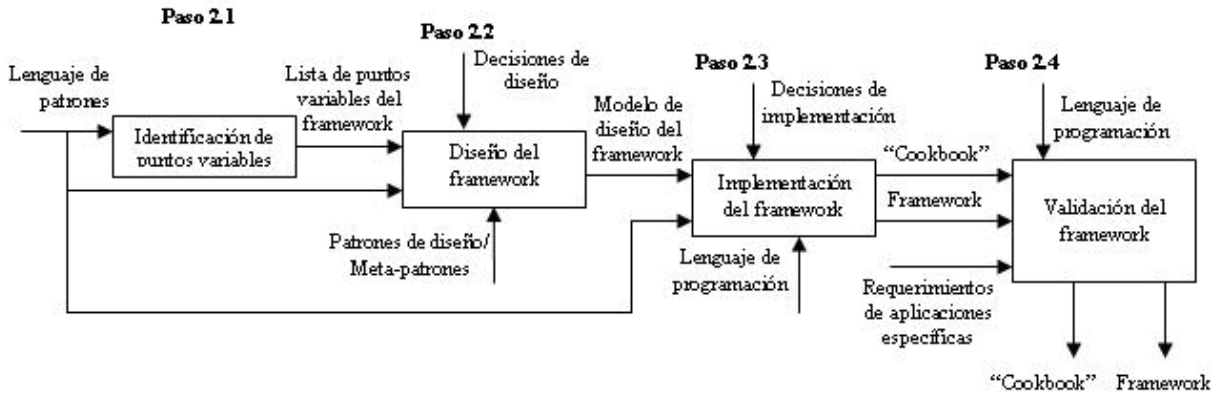
10. Elabore una tabla conteniendo el "historial de patrones y variantes utilizados", con cuatro columnas: la primera mostrando el número y el nombre del patrón utilizado, la segunda muestra la variante o sub-patrón utilizado (use la palabra Default si utiliza el patrón tal cual es presentado en la solución), la tercera nos presenta el nombre de la clase participante del patrón y la cuarta indica el nombre de la clase de la aplicación que desempeña el papel del participante de la tercera columna.

Los pasos definidos ayudan a la aplicación sistemática de los lenguajes de patrones. Esto facilita la futura implementación del sistema utilizando un framework que se apoye en el lenguaje de patrones, ya que son provistas informaciones sobre los patrones utilizados. Por otra parte, el hecho de que las funcionalidades no cubiertas por el framework tendrán un destacado especial en el diagrama, facilitando la identificación de las partes de la aplicación que precisan ser implementadas aparte. Estas informaciones pueden, también, ser utilizadas para mejorar el lenguaje de patrones, incluyendo nuevos patrones u ofreciendo soluciones alternativas para contextos diferentes.

1.5.4. Proceso de construcción de frameworks en base a lenguaje de patrones.

La figura 1.3 ilustra el proceso, el cual está constituido por 4 pasos, los que se explicarán a continuación. Cabe destacar que la secuencia de pasos es flexible, pudiendo iterar cuantas veces sea necesario para obtención de un framework final.

Figura 1.3. Proceso para la construcción de frameworks basados en lenguaje de patrones.

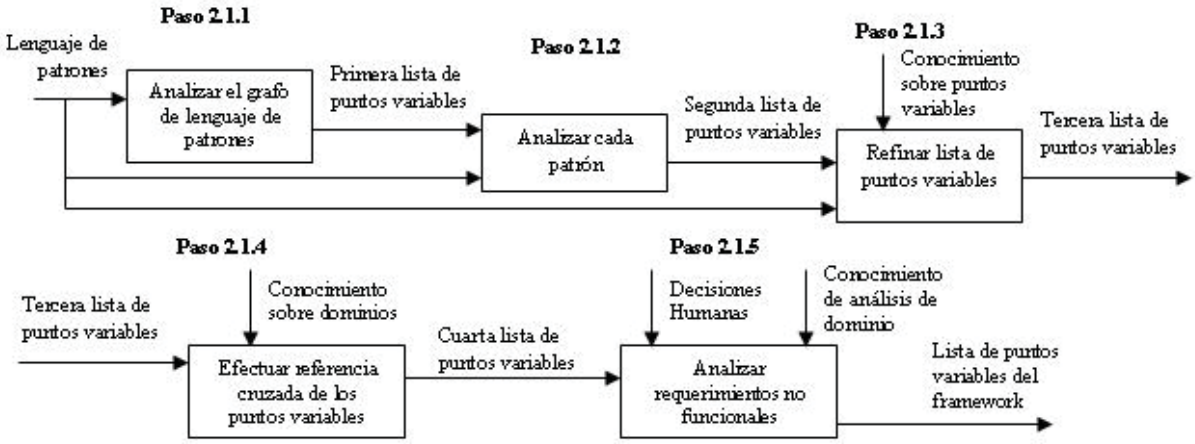


Identificación de Puntos Variables.

La primera actividad para el desarrollo de un framework es la identificación de sus puntos variables, es decir, de los puntos en los que se necesita flexibilidad del software, ya que ellos

variarán las distintas aplicaciones dentro del dominio. Como se aprecia en la figura, es necesaria la información que presentan los lenguajes de patrones, con la cual se llevan a cabo cinco pasos ilustrados en la figura 1.4.

Figura 1.4. Identificación de puntos variables del framework.



El primer sub-paso es analizar el grafo del lenguaje de patrones (si existe), o como alternativa, las secciones "Próximos patrones". El objetivo de este análisis es encontrar los puntos variables relacionados con los patrones opcionales del lenguaje, es decir, patrones que no son usados por todas las aplicaciones pertenecientes al dominio, pero que pueden ser opcionalmente utilizados. El grafo del lenguaje de patrones indica las interacciones entre los patrones y el orden en cual ellos pueden ser aplicados. La sección "Próximos patrones", comúnmente utilizada en los patrones de un lenguaje de patrones, indica los caminos a seguir después de aplicar (o no) el patrón actual. Por lo tanto, esos son lugares propicios para encontrar puntos variables del framework que será construido.

En el segundo sub-paso se debe analizar individualmente cada patrón, analizando cada una de sus secciones constituyentes, ya que ellas pueden indicar diversos puntos variables del framework. En las secciones "Variantes" o "Sub-patrones", podrían encontrarse soluciones alternativas para el problema resuelto por los patrones. Por lo tanto, puede haber indicaciones de aspectos variables que deberían estar disponibles para los usuarios del framework.

El tercer sub-paso refina la especificación de cada uno de los puntos variables, con la intención de proveer la información suficiente, para que posteriormente puedan ser diseñados e implementados. Además, se incluirán otros puntos variables no incluidos en el lenguaje de

patrones, pero que pueden ser incluidos para aumentar la flexibilidad del framework. La información de estos nuevos puntos variables debe ser utilizada para mejorar el lenguaje de patrones en el próximo ciclo iterativo.

En el cuanto sub-paso se debe efectuar una referencia cruzada entre los puntos variables identificados, lo que ayuda a detectar inconsistencias en la lista como un todo, generando algunos puntos variables más. El conocimiento del dominio es imprescindible en este punto y, así como en el tercer paso los nuevos puntos variables descubiertos son utilizados para mejorar el lenguaje de patrones.

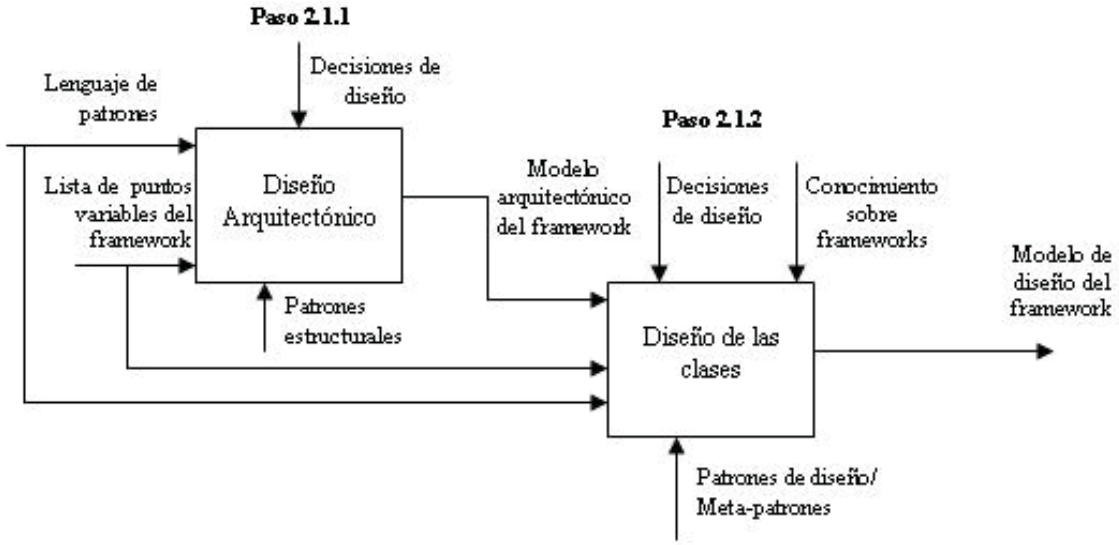
En el quinto sub-paso, son considerados los requerimientos no funcionales de la aplicación, ya que ellos puedan originar nuevos puntos variables, incluyendo portabilidad, usabilidad, seguridad y confiabilidad. También deben ser evaluados los aspectos del diseño e implementación que puedan traer más flexibilidad al framework, aumentando su potencial de reutilización.

El resultado del paso 2.1 es una lista de puntos variables del framework cuyos elementos son compuestos de un código de identificación de punto variable, su descripción, su tipo, su sección fuente correspondiente en el lenguaje de patrones y el patrón en el cual fue identificado. El tipo de punto variable permite saber lo que debe ser hecho para obtener una aplicación a partir del framework.

Diseño del framework.

Identificados los puntos variables se inicia, entonces, el diseño del framework (paso 2.2 de la figura 3). El framework debe poseer una estructura flexible de modo de acomodar todos los puntos variables encontrados en el punto anterior. Se necesitan dos pasos para diseñar el framework, como lo indica la figura 1.5: diseño arquitectónico y diseño de jerarquía de clases.

Figura 1.5. Diseño del framework.



El primer sub-paso es responsable del diseño arquitectónico del framework, durante el cual el desarrollador toma decisiones respecto de la arquitectura del framework que mejor atienda los requerimientos del framework. Los mecanismos de persistencia de los objetos deben ser escogidos y diseñados, así como la interfaz gráfica de usuario, los tópicos de seguridad y control de acceso, y, finalmente, aspectos de distribución. Este sub-paso no solamente envuelve al desarrollador del framework sino que también a la gerencia de la organización, ya que la elección de la arquitectura de software implica mayores gastos y tiempo de desarrollo, al mismo tiempo en que proporciona un framework más o menos reutilizable a largo plazo.

El segundo paso consiste en la creación de la jerarquía de clases del framework, a partir de la estructura propuesta por cada patrón del lenguaje de patrones. Se asume que los patrones poseen una sección con su estructura en forma de un diagrama de clases o algún modelo de análisis similar. También son entradas para este proceso la lista de los puntos variables del framework y los patrones de diseño o meta-patrones.

Implementación del framework.

La implementación del framework (paso 2.3 de la figura 1.3), está constituida por dos pasos ilustrados en la figura 1.6: implementación de clases en un lenguaje de programación específico y documentación del mapeado entre lenguaje de patrones y framework.

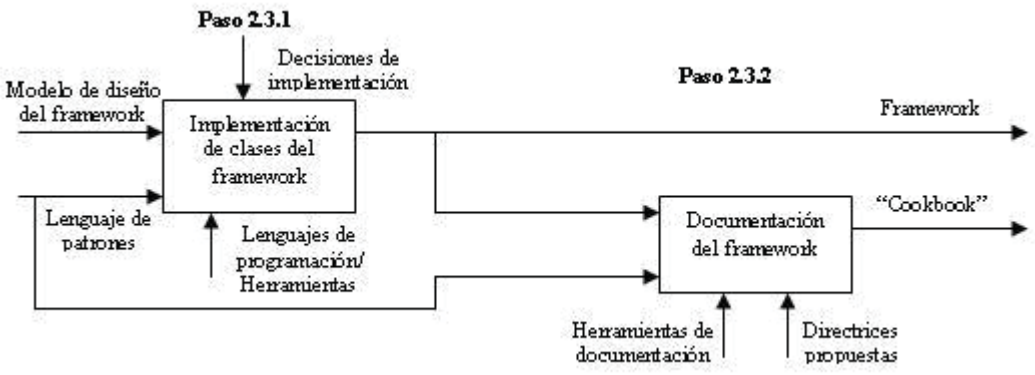
El primer sub-paso es la implementación de las clases obtenidas desde el modelo de diseño del punto anterior, usando un lenguaje de programación específico. El lenguaje de patrones

provee el apoyo necesario al programador, cuando surgen dudas en cuanto a la funcionalidad de las clases participantes, ya que es una fuente excelente de información sobre el dominio. En este paso, además, se toman decisiones relevantes sobre la implementación, por ejemplo, elección de estructuras de datos, elección de dispositivos utilizados para el ingreso de datos y para la interfaz gráfica, etc. El resultado de este paso es el código del framework. Es importante resaltar, aquí, que este sub-paso es bastante complejo, dependiendo de la arquitectura de software definida en el paso anterior. Para facilitar el proceso se pueden utilizar componentes de software o bibliotecas reutilizadas de otros proyectos, además de herramientas de ingeniería de software existentes.

El lenguaje de patrones puede ser utilizado para guiar la implementación del framework de forma incremental. Cada patrón puede ser considerado como una unidad funcional, de manera que la implementación pueda comenzar por el primer patrón y proseguir con los demás patrones hasta que todo el framework este implementado.

El segundo sub-paso trata sobre la documentación adecuada del framework. Las directrices provistas en el proceso detallado conducen a una documentación que provee el mapeado del lenguaje de patrones para las clases del framework. El propósito general es de facilitar la instanciación del framework para aplicaciones específicas, de forma que, teniendo en las manos el historial de patrones y variantes aplicados en el modelado, la documentación del framework provea medios para identificar cuales clases deben ser creadas, así como cuales clases del framework deben ser especializadas para originarlas, y cuales métodos deben ser sobrepuestos.

Figura 1.6. Implementación del framework.

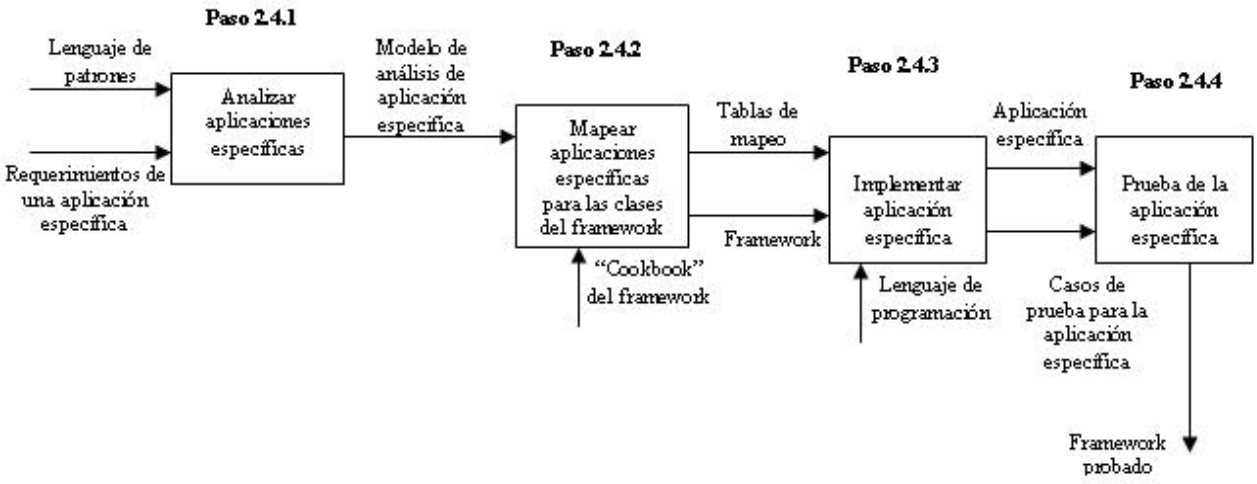


Resumiendo, los usuarios del framework pueden fácilmente saber lo que debe ser hecho en cada parte del framework para obtener aplicaciones específicas, de acuerdo a de que forma utilicen los patrones del lenguajes de patrones para modelar sus aplicaciones.

Validación del framework.

La actividad final del proceso de construcción del framework es su validación (paso 2.4 de la figura 1.3), compuesta de cuatro sub-pasos ilustrados en la figura 1.7. El objetivo de la validación es probar el framework para distintas aplicaciones específicas, antes de que sea liberado para su uso general. Las entradas a este proceso son el framework, su manual de instanciación y los documentos de requerimientos para varias aplicaciones específicas del dominio. El lenguaje de patrones puede ayudar al desarrollador del frameworks a definir cuales aplicaciones van a ser probadas.

Figura 1.7. Validación del framework.



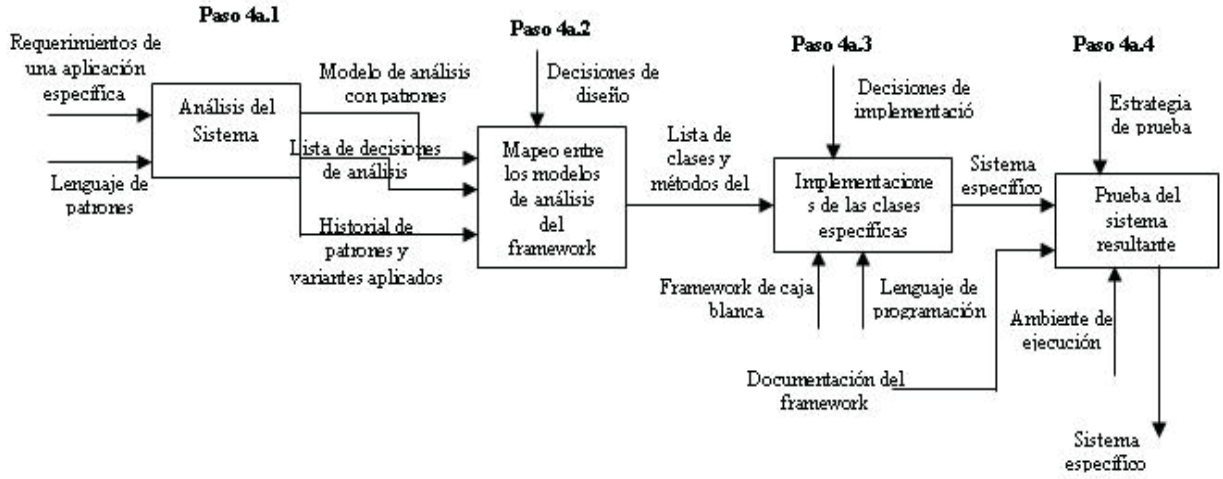
Se debe resaltar que el proceso de validación utiliza el mismo proceso de instanciación de aplicaciones, descrito en la sección 1.5.5, ya que para probar el framework, éste debe ser instanciado para aplicaciones concretas del dominio

1.5.5 Proceso de instanciación de frameworks.

La reutilización de un framework de caja blanca es realizada por medio de herencia. Los requerimientos de la aplicación a ser instanciada deben ser analizados para descubrir cuales

son las clases y a partir de que clases del framework ellas deben ser especializadas. Se sugiere un proceso para instanciación ilustrado en la figura 1.8. Este proceso se compone de cuatro pasos: análisis del sistema, mapeo entre el modelo de análisis y el framework, implementación de las clases específicas y prueba del sistema resultante.

Figura 1.8. Proceso de instanciación de framework de caja blanca.



Análisis del sistema.

El primer paso para instanciar un framework de caja blanca es el análisis del sistema, con base en lenguaje de patrones (paso 4a.1 de la figura 1.8). Este paso es explicado detalladamente en la sección que muestra el proceso de utilización de los lenguajes de patrones para el modelado de un sistema específico del mismo dominio del lenguaje. Las entradas para este paso son el documento de requerimientos de un sistema específico y el lenguaje de patrones. El resultado es un modelo de análisis del sistema, el historial de los patrones y variantes utilizados y una lista de las decisiones tomadas cuando el lenguaje de patrones no atiende a todos los requerimientos del sistema.

Correlación entre el modelo de análisis y el framework.

Al poseer el modelo de análisis del sistema y el historial de los patrones y variantes utilizados, se aplica entonces el segundo paso del proceso general (paso 4a.2 de la figura 1.8) para producir el mapeo entre el modelo de análisis y las clases del framework. Se asume que el framework ha sido documentado de acuerdo a las directrices sugeridas en el paso de implementación del framework, es decir, la relacione entre patrones del lenguaje de patrones

y las clases del framework debe estar documentada apropiadamente. El resultado de este paso es una lista de las clases de la aplicación y los métodos correspondientes a ser implementados.

El procedimiento a seguir para realizar el mapeo entre el modelo de análisis y el framework es específico para cada par "lenguaje de patrones y framework". El algoritmo genérico, contiene una abstracción de las actividades más importantes en la realización del mapeo:

1. Considere el "Historial de los patrones y variantes aplicados" creado durante el análisis del sistema que contiene, para cada patrón aplicado, o variante o sub-patrón utilizado y los papeles desempeñados por cada clase. Sea TH el nombre de esa tabla. Para cada línea de TH, sea P el patrón aplicado, V la variante o sub-patrón utilizado, R la clase participante del patrón y A la clase de aplicación, tal que A desempeña el papel R en el patrón P y variante V.
2. Construya una tabla conteniendo las clases a ser creadas en el nuevo sistema. Sea TC esa tabla. La tabla debe tener cuatro columnas: clase de aplicación, clase a crear, súper-clase del framework y nuevos atributos.
3. Para cada línea de TH, encuentre, en la documentación del framework, las clases que deben ser creadas en el sistema nuevo para la tupla (A, P, V, R). Como resultados son obtenidos los nombres de las nuevas clases a ser creadas referentes a A, denominadas A1,A2, ..., An. Así como las súper-clases del framework de las cuales ellas deben heredan, denominadas S1, S2, ...,Sn. Cree entonces en la tabla TC, "n" líneas cada cual conteniendo: A, en la primera columna; Ai, en la segunda columna; Si en la tercera columna; y nuevos atributos agregados a la clase A, en la cuarta columna (tales atributos fueron destacados en el modelo de análisis).
4. Finalmente, considere a tabla TC resultante. Para cada línea de TC, examine la documentación del framework para identificar cuales son los métodos que precisan ser sobrepuestos. Defina el contenido de esos métodos. Incluya también los métodos para implementar los nuevos atributos y operaciones.

Implementación del sistema específico.

En el paso 4a.2 fueron definidas todas las clases que serán implementadas en la nueva aplicación, conjuntamente con sus métodos. En esta sección se han provisto sugerencias para implementaciones dichas correctamente de esas clases (paso 4a.3 de la figura 1.8). Debe ser utilizada, en la implementación, el mismo lenguaje de programación en el cual fue implementado el framework. El resultado es el código fuente de las nuevas clases de aplicación. Las siguientes recomendaciones pueden ser usadas para guiar la implementación:

1. Inicialmente, cree todas las clases identificadas durante el mapeo, así como los métodos correspondientes.
2. Cree los métodos SET y GET para todos los atributos adicionales de las clases (cuarta columna de la tabla TC).

3. En el caso de sistemas de información convencionales, es común la existencia de una GUI (graphics user interface) representando la ventana principal de aplicación, conjuntamente con el menú a ser ejecutado por el usuario final. Ese menú debe contener atajos para todas las operaciones del sistema
4. Implemente otras clases y operaciones no previstas en el framework. Es necesario un profundo entendimiento de framework de caja blanca, para que sean realizadas las conexiones de las nuevas funcionalidades con las provistas por el framework.
5. Finalmente, compile la nueva aplicación.

Prueba del sistema resultante.

El sistema obtenido en el paso 4a.3 debe ser testeado, tanto para garantizar que atiende a los requerimientos establecidos, como para verificar si funciona en el ambiente del usuario final. Se sugiere que sea escogida una estrategia de pruebas, y que los casos de prueba sean elaborados para chequear el cumplimiento de los requerimientos funcionales y no funcionales. El resultado deseado es una aplicación debidamente probada, que se entregue al usuario final.

1.6 Descripción de objetivos.

En base a los conceptos planteados anteriormente se definen los siguientes objetivos generales y específicos para el proyecto.

1.6.1 Objetivos generales.

Proponer una guía general, basada en lenguaje de patrones, para la evolución de los framework de caja blanca hacia frameworks de caja gris, la cual deberá estar lo suficientemente detallada.

Describir el aporte a la reutilización realizado por la guía propuesta mediante métricas utilizadas en ingeniería de software.

Desarrollar una prototipo de una herramienta que asista uno de los pasos del proceso de evolución de frameworks basada en lenguajes de patrones para cualquier par frameworks/lenguaje de patrones.

1.6.2 Objetivos específicos.

- Analizar el proceso de construcción e instanciación de frameworks de caja blanca basados en lenguajes de patrones para un dominio específico.
- Estudiar la construcción e instanciación de frameworks de caja gris y su diferencia con la construcción e instanciación de frameworks de caja blanca.
- Creación detallada de la guía para evolucionar frameworks de caja blanca hacia frameworks de caja gris.
- Formulación de métricas para la validación de la reutilización.
- Validar mediante las métricas formuladas la guía propuesta.
- Analizar el proceso de construcción de herramientas que asistan la evolución de frameworks.
- Creación del modelo que sustente una herramienta que asista uno de los pasos del proceso de evolución de frameworks de caja blanca hacia frameworks de caja gris basándose en lenguaje de patrones.
- Desarrollo de un prototipo de la herramienta que asista la evolución de frameworks.

1.7 Metodología.

El objetivo de cualquier investigación es adquirir conocimientos, por ello, es fundamental la elección del método adecuado que nos permita conocer la realidad, además es fundamental formular el problema a solucionar de forma suficientemente concreta como para que pueda ser resuelto efectivamente por la vía científica (solucionable científicamente), y suficientemente clara como para saber qué datos buscar para resolverlo.

El problema a solucionar mediante esta investigación es validar una guía que ha sido claramente definida para ayudar a los desarrolladores a evolucionar frameworks de caja blanca hacia frameworks de caja gris apoyándose en lenguaje de patrones, y asistir el proceso de evolución mediante una herramienta.

De esta forma, considerando los objetivos definidos en éste proyecto las actividades que se realizarán para abordar su desarrollo serán las siguientes:

1. Se identificarán y analizarán las características de la construcción e instanciación de frameworks de caja blanca basada en lenguaje de patrones.

Con esto se pretende entender los procesos que se llevan a cabo para desarrollar aplicaciones específicas a partir de un framework desarrollado en base a un lenguaje de patrones.

2. Se identificarán y analizarán las diferencias de la construcción e instanciación de frameworks de caja blanca con la construcción e instanciación de frameworks de caja gris basándose en lenguaje de patrones.

Con el fin de generar una guía para la evolución de frameworks de caja blanca hacia frameworks de caja gris, apoyando el proceso en lenguaje de patrones. La propuesta deberá tener el suficiente nivel de detalle como para que pueda servir de ayuda a otros desarrolladores.

3. Se aplicarán métricas de ingeniería de software a frameworks evolucionados utilizando la guía propuesta para validar su mayor aporte a la reutilización.

Con el fin de validar la propuesta realizada se aplicarán métricas a frameworks específicos. De ésta forma se identificará si existen problemas en la propuesta y de ser así se tomarán las acciones correctivas necesarias, en base a esto se presentará la propuesta con sus respectivas mejoras.

4. Se identificarán y analizarán las características de la construcción de un prototipo de una herramienta que asista el proceso de evolución de frameworks de caja blanca hacia frameworks de caja gris.

Con esto se pretende entender los procesos que se llevan a cabo para evolucionar frameworks desarrollados en base a un lenguaje de patrones, pero ahora de forma asistida, resaltando el apoyo que brinda la guía validada.

5. Se construirá un prototipo de una herramienta que asista la evolución de frameworks.

Con el fin de ayudar a la aplicación de la guía y con ella a la reutilización de frameworks se implementará un prototipo de una herramienta que asista el primer punto de la guía planteada, el cual consiste en encontrar los puntos comunes entre las aplicaciones del dominio y mapear

esos puntos con los patrones del framework de caja blanca para determinar que métodos y clases pueden ser modificados para así poder evolucionar el framework a caja gris.

1.8 Resumen del capítulo.

En este capítulo se dio a conocer de modo general la estructura de este proyecto de título y junto con ella se presentó el marco teórico necesario para desarrollar una guía para la evolución de frameworks de caja blanca hacia caja gris basándose en lenguaje de patrones.

Se presentaron los patrones de software y los lenguajes de patrones demostrando el gran aporte que realizan a la reutilización de componentes, también se presentaron los frameworks de caja blanca y se enseñaron sus procesos de construcción y posterior instanciación para crear sistemas para dominios específicos, lo que constituye el marco teórico necesario para dar forma a la guía propuesta, la cual será formulada en el capítulo siguiente, de acuerdo los objetivos generales y específicos planteados y la metodología de trabajo a realizar.

Capítulo 2. Guía propuesta para la evolución de Frameworks de caja blanca hacia caja gris basándose en lenguaje de patrones.

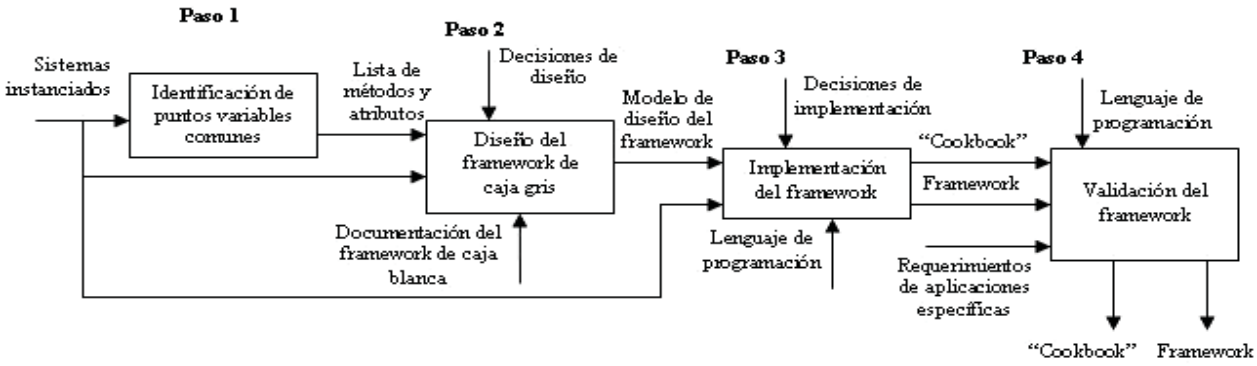
2.1. Introducción.

Es importante destacar que no se ha encontrado información referente a la construcción e instanciación de frameworks de caja gris desde cero, debido a que es un proceso mucho más difícil, por lo tanto, se debe realizar una primera versión del framework, el cual será de caja blanca, que son mucho más sencillos de desarrollar, y posteriormente teniendo experiencia en la instanciación de él para crear aplicaciones específicas, será posible lograr el conocimiento necesario para evolucionarlo y así obtener un framework de caja gris.

2.2. Guía para la evolución de frameworks de caja blanca hacia frameworks de caja gris.

Se ha definido una guía que tiene como objetivo ayudar a otros desarrolladores a evolucionar sus frameworks de caja blanca para dominios específicos en framework de caja gris, lo cual es un paso intermedio sumamente importante para lograr frameworks de caja negra, estos últimos nos permitirán instanciar aplicaciones específicas de una manera mucho más fácil, ya que no será necesario tener conocimiento de la implementación del framework, sino que solamente de las interfaces que definen los métodos que completarán las funciones en la nueva aplicación. La guía ha sido diseñada en 4 etapas o actividades, como lo indica la figura 2.1, las cuales son presentadas a continuación: identificación de puntos variables, diseño del framework de caja gris, implementación del framework de caja gris y validación del framework obtenido.

Figura 2.1. Proceso de evolución de frameworks de caja blanca hacia frameworks de caja gris basándose en lenguajes de patrones.



2.2.1. Identificación de puntos y variables comunes.

Como ya se mencionó anteriormente, la primera actividad para el desarrollo de un framework de caja blanca es la identificación de sus puntos y variables. Ahora bien, lo que permitió su desarrollo permitirá su evolución, entregándonos puntos comunes entre las diferentes aplicaciones que han sido instanciadas, los cuales entregarán las clases implementadas en la aplicación y haciendo un mapeo de esta información con la documentación del framework se determinarán desde que súper-clases del framework heredaron su comportamiento.

Para identificar los puntos comunes entre las distintas aplicaciones instanciadas a partir de un framework para un dominio específico existen diversas formas, una de ellas es el análisis de ingeniería inversa, el cual entregará las funciones comunes entre las distintas aplicaciones analizadas, esta opción puede complicarse debido a la gran concentración de esfuerzos que esta metodología conlleva al tratar de comprender el funcionamiento de las aplicaciones y al ser contraria al común funcionamiento de los procesos de ingeniería. Por otro lado, es posible comparar los requerimientos de las aplicaciones instanciadas, para ello es necesario definir claramente el dominio en el cual se enmarcan las aplicaciones instanciadas y así poder determinar las funcionalidades comunes dentro del dominio. Teniendo las funciones comunes es necesario compararlas con las funciones que son implementadas por el framework y que son explicadas claramente en la documentación de éste, así se determinarán qué clases proveen los métodos que realizarán las funciones enumeradas, cómo se relacionan unas con otras y su orden de aplicación para lograr el resultado deseado.

Sin importar la forma que se utilice para determinar los puntos y variables comunes, el resultado de esta etapa será un listado de los métodos y atributos que son heredados y las súper-clases que proveen esos métodos y atributos, además de la implementación de las clases

y métodos encontrados, identificando sus partes mutables e inmutables, así como los costes de los posibles cambios.

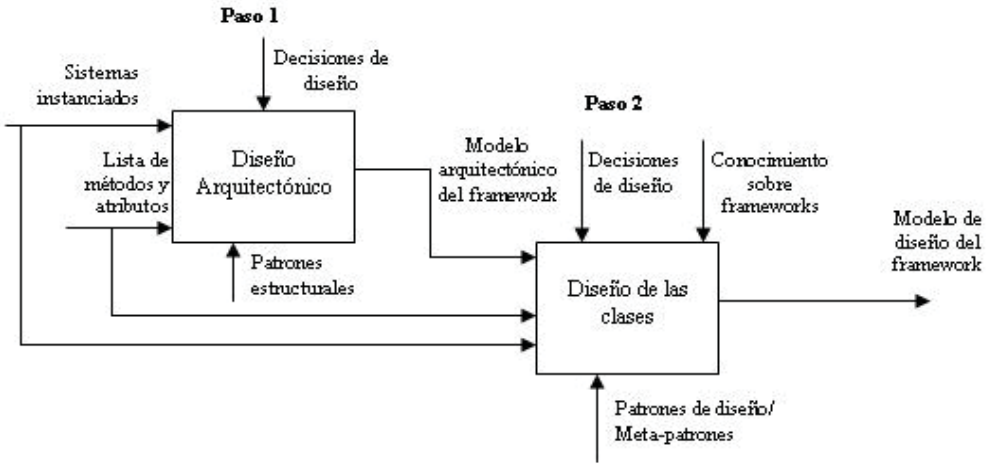
Analizando el comportamiento de cada método, de las clases que los proveen y las relaciones entre ellas, es posible determinar que métodos deben explicar su comportamiento mediante interfaces, debido a que es innecesario saber su funcionamiento y sólo nos interesa obtener cierta respuesta de acuerdo a las entradas que le son proporcionadas, además podemos determinar que clases presentan más de un comportamiento de acuerdo con respecto a una misma entrada, esto nos indicará que la clase puede ser divida en dos o más clases abstractas que representen cada comportamiento y entre ellas deberán relacionarse mediante composición.

Este proceso manual de recolección de información puede ser apoyado por una herramienta que identifique los puntos comunes y genere el listado, la cual será analizada, diseñada y construida en este trabajo. Toda la información recolectada permitirá determinar las pautas para diseñar el framework de caja gris, ya que con el listado obtenido se debe rehacer el modelo de clases del framework como se explica en el siguiente punto.

2.2.2. Diseño del framework de caja gris.

Al haber encontrado las clases, métodos y atributos más comunes entre las aplicaciones instanciadas y tener la documentación correspondiente a cada una de ellas, se realiza el diseño de framework de caja gris, esta etapa se divide en dos sub-etapas: diseño arquitectónico y diseño de clases.

Figura 2.2. Etapas del diseño del framework de caja gris.



El diseño arquitectónico debe ser desarrollado de la misma manera como se realizó en el framework de caja blanca. Es preciso definir que una Arquitectura de Software es una representación de alto nivel de la estructura de un sistema o aplicación, que describe las partes que la integran, las interacciones entre ellas, los patrones que supervisan su composición, y las restricciones a la hora de aplicar esos patrones [2]. Es importante tener en consideración que la arquitectura deberá ser la adecuada para un modelo que incluya herencia y composición, por lo tanto, deberá permitir expresar los cambios en las conexiones e interacciones entre los componentes tanto en tiempo de instanciación como de ejecución. La arquitectura resultante deberá cubrir a lo menos los siguientes objetivos:

1. Comprender y manejar la estructura de las aplicaciones complejas.
2. Reutilizar la estructura (o partes de ella) para resolver problemas similares.
3. Planificar la evolución de la aplicación, identificando sus partes mutables e inmutables, así como los costes de los posibles cambios.
4. Analizar la corrección de la aplicación, y su grado de cumplimiento respecto a los requisitos iniciales (prestaciones o fiabilidad).
5. Permitir el estudio de alguna propiedad específica del dominio.

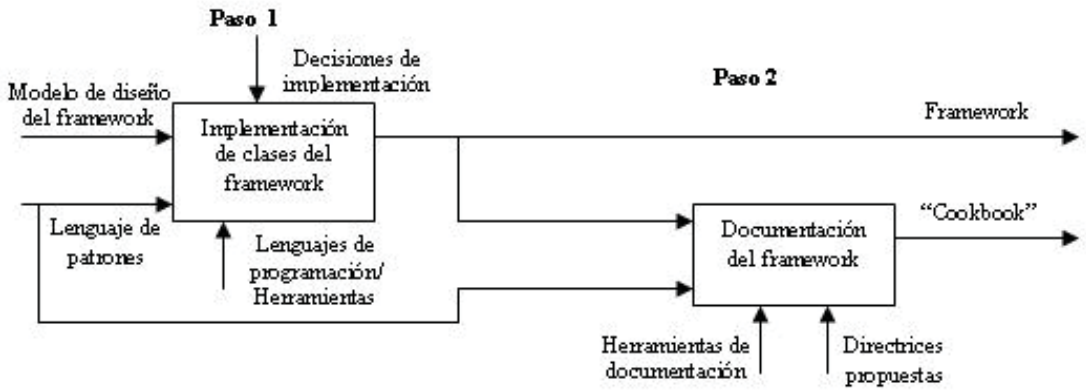
El segundo paso consiste en la creación del diagrama de clases del framework, es en este punto donde se produce la mayor diferencia con respecto a los frameworks de caja blanca, ya que el diagrama no sólo se basará en herencia, sino que deberá incorporar composición. A continuación se proponen los pasos a seguir para transformar el diseño de clases:

1. Generar un ranking con la lista de puntos variables comunes encontrados en la etapa 1, con el fin de determinar los métodos y atributos públicos (comportamiento) que son mayormente utilizados por las instanciaciones y las súper-clases que los proveen.
2. Crear las interfaces para los métodos y atributos mejor rankeados. Se entiende por interfaz como la declaración del comportamiento externo de las clases. En Java hay un lenguaje construido para esto.
3. Para cada súper-clase encontrada que encapsule múltiples comportamientos, cree clases abstractas que encapsulen cada comportamiento.
4. Toda relación de herencia que incluyera el comportamiento encapsulado de la clase original, substitúyala por una composición que reconstruya el comportamiento deseado. Esto reducirá la duplicación del código, así como la necesidad de crear las subclases nuevas para cada nuevo uso.
5. Todo comportamiento no encapsulado en una clase abstracta debe ser heredado desde la súper-clase que lo provee.

2.2.3. Implementación del framework de caja gris.

Al igual que en la implementación del framework de caja blanca, esta etapa se compone de dos sub-etapas: implementación de clases en algún lenguaje de programación específico y documentación de la evolución que se ha llevado a cabo.

Figura 2.3. Etapas de la implementación de frameworks de caja gris.



El primer paso es la implementación de las clases obtenidas desde el modelo de diseño del punto anterior, usando un lenguaje de programación específico. Como se vio en la implementación del framework de caja blanca el lenguaje de patrones provee el apoyo necesario al programador, cuando surgen dudas en cuanto a la funcionalidad de las clases participantes, ya que es una fuente excelente de información sobre el dominio, pero ahora se debe considerar que ciertas funcionalidades ya no será heredadas de súper-clases del modelo, sino que se formarán nuevas clases compuestas las que entregarán las funcionalidades

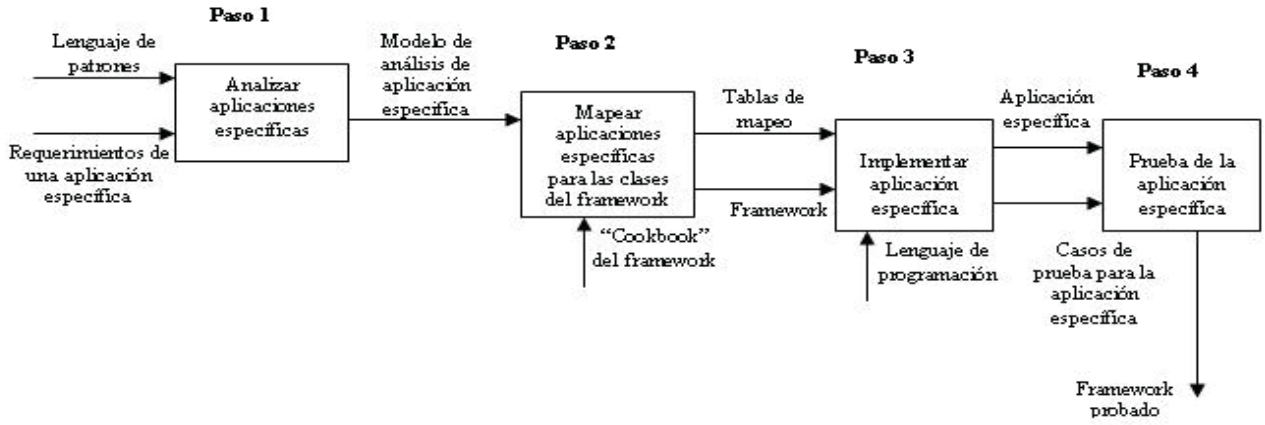
deseadas. Es importante una buena elección de estructuras de datos, dispositivos utilizados para el ingreso de datos y para la interfaz gráfica, debido a la mayor complejidad en la arquitectura se recomienda utilizar programación basada en componentes o bibliotecas de reutilización como complemento a las técnicas comunes de la ingeniería de software. El resultado de este paso es el código del framework.

El segundo paso es la documentación de la evolución que ha tenido el framework, esto es necesario para apoyar de manera activa la instanciación de aplicaciones específicas, de forma que, teniendo en las manos las variaciones que ha sufrido el diagrama de clases se provean los medios para identificar cuales clases deben ser creadas, así como cuales clases del framework deben ser especializadas para originarlas, y cuales deben ser compuestas y los métodos que deben ser sobrepuestos. También es un proceso importante para la mantención de las aplicaciones instanciados, ya que este proceso sólo es posible teniendo la documentación necesaria actualizada.

2.2.4. Validación del framework de caja gris.

La actividad final del proceso de evolución de framework de caja gris es su validación (paso 4 de la figura 2.1), compuesta de cuatro sub-etapas ilustradas en la figura 2.4. Al igual que en los frameworks de caja blanca, el objetivo de la validación es probar el framework para distintas aplicaciones específicas, antes de que sea liberado para su uso general.

Figura 2.4. Etapas de la validación del framework de caja gris.

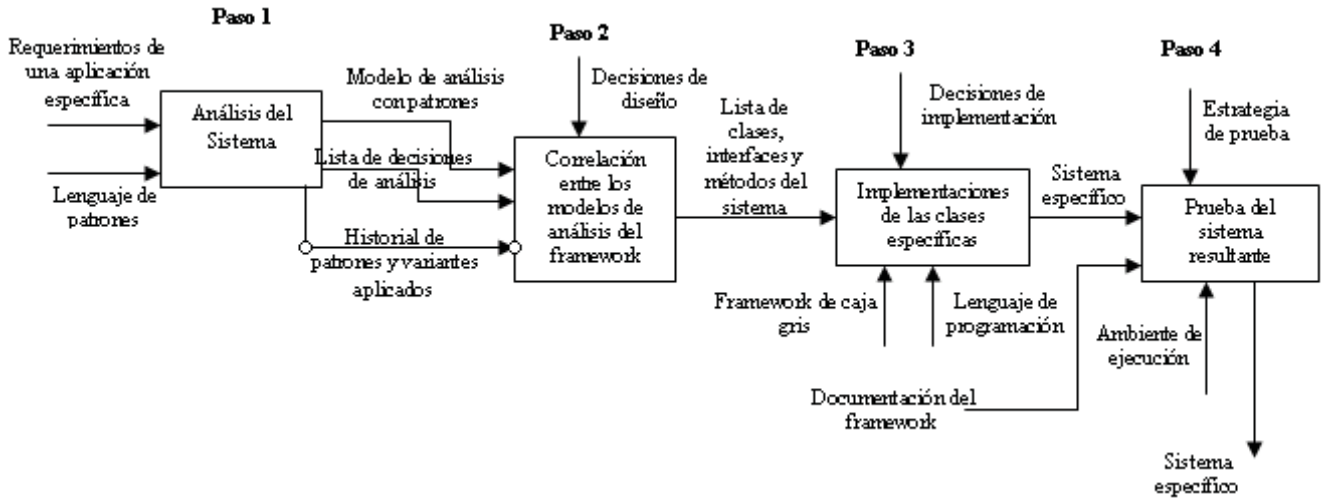


Se debe resaltar que el proceso de validación utiliza el mismo proceso de instanciación de aplicaciones, ya que para probar el framework, éste debe ser instanciado para aplicaciones concretas del dominio.

2.3. Proceso de instanciación de frameworks.

Como ya se mencionó, la reutilización de un framework de caja blanca es realizada por medio de herencia, en cambio, los framework de caja gris lo hacen tanto por herencia como por composición. Los requerimientos de la aplicación a ser instanciada deben ser analizados para descubrir cuales son los comportamientos que se reutilizarán y a partir de que clases del framework son obtenidos. Se sugiere un proceso para instanciación ilustrado en la figura 2.5. Este proceso se compone de cuatro pasos: análisis del sistema, correlación entre el modelo de análisis y el framework, implementación de las clases e interfaces específicas y prueba del sistema resultante.

Figura 2.5. Proceso de instanciación de framework de caja gris.



2.3.1. Análisis del sistema.

El primer paso para instanciar un framework de caja gris es el análisis del sistema, con base en lenguaje de patrones (paso 1 de la figura 2.5). En este paso se debe seguir el proceso detallado en la sección 1.5.3, utilización de los lenguajes de patrones para el modelado de un sistema específico del mismo dominio del lenguaje. El resultado es un modelo de análisis del

sistema, el historial de los patrones y variantes utilizados y una lista de las decisiones tomadas cuando el lenguaje de patrones no atiende a todos los requerimientos del sistema.

2.3.2. Correlación entre el modelo de análisis y el framework.

Al poseer el modelo de análisis del sistema y el historial de los patrones y variantes utilizados, se aplica entonces el segundo paso del proceso general (paso 2 de la figura 2.5) para producir la correlación entre el modelo de análisis y las clases del framework. Se asume que el framework ha sido documentado de acuerdo a las directrices sugeridas en el paso de implementación del framework (ver sección 2.2.3), es decir, la relación entre patrones del lenguaje de patrones y las clases del framework debe estar documentada apropiadamente. El resultado de este paso es una lista de las clases de la aplicación (concretas y abstractas), interfaces de los métodos y los métodos correspondientes a ser implementados.

Como se mencionó el procedimiento a seguir para realizar la correlación es específico para cada par "lenguaje de patrones y framework", es por ello, que se presenta un algoritmo genérico, él cual contiene una abstracción de las actividades más importantes en la realización de la correlación:

1. Cree una tabla (TH) con el historial para cada patrón aplicado, variante o sub-patrón utilizado y los papeles desempeñados por cada clase. Para cada línea de TH, sea P el patrón aplicado, V la variante o sub-patrón utilizado, R la clase o interfaz participante del patrón y A la clase de aplicación, tal que A desempeña el papel R en el patrón P y variante V.
2. Construya una tabla (TC) conteniendo las clases o interfaces a ser creadas en el nuevo sistema. TC debe tener, al menos, cuatro columnas: clase de aplicación, clase o interfaz a crear, súper-clase o clase componente del framework y nuevos atributos.
3. Para cada fila de TH encuentre, en la documentación del framework, las clases o interfaces que deben ser creadas en el sistema nuevo para la tupla (A, P, V, R). Como resultados son obtenidos los nombres de las nuevas clases o interfaces a ser creadas referentes a A, denominadas A_1, A_2,..., A_n. Así como las súper-clases o clases compuestas del framework de las cuales ellas deben extender su comportamiento, denominadas C_1, C_2, ...,C_n. Cree entonces en la tabla TC, "n" filas, cada cual conteniendo: A, en la primera columna; A_i, en la segunda columna; C_i en la tercera columna; y nuevos atributos agregados a la clase A, en la cuarta columna.
4. Finalmente, considere a tabla TC resultante. Para cada fila de TC, examine la documentación del framework para identificar cuales son los comportamientos que deben ser extendidos. Defina el contenido de esos métodos e incluya también los métodos para implementar los nuevos atributos y operaciones.

2.3.3. Implementación del sistema específico.

En el paso 2 fueron definidas todas las clases e interfaces que serán implementadas en la nueva aplicación, conjuntamente con sus métodos. En esta sección se han provisto sugerencias para implementar correctamente dichas clases o interfaces (paso 3 de la figura 2.5). Debe ser utilizada, en la implementación, el mismo lenguaje de programación en el cual fue implementado el framework. El resultado es el código fuente de las nuevas clases de la aplicación. Las siguientes recomendaciones pueden ser usadas para guiar la implementación:

1. Inicialmente, cree todas las clases e interfaces identificadas durante la correlación, así como los métodos correspondientes.
2. Cree los métodos SET y GET para todos los atributos adicionales de las clases (cuarta columna de la tabla TC).
3. En el caso de sistemas de información convencionales, es común la existencia de una GUI (graphics user interface) representando la ventana principal de aplicación, conjuntamente con el menú a ser ejecutado por el usuario final. Ese menú debe contener atajos para todas las operaciones del sistema.
4. Implemente otras clases y operaciones no previstas en el framework. Es necesario un profundo entendimiento de las interfaces del framework, así como, de las métodos que serán heredados, para que sean realizadas las conexiones de las nuevas funcionalidades con las provistas por el framework.
5. Finalmente, compile la nueva aplicación.

2.3.4. Prueba del sistema resultante.

El sistema obtenido en el paso 3 debe ser testeado, tanto para garantizar que atiende a los requerimientos establecidos, como para verificar si funciona en el ambiente del usuario final. Se sugiere que sea escogida una estrategia de pruebas, y que los casos de prueba sean elaborados para chequear el cumplimiento de los requerimientos funcionales y no funcionales. El resultado deseado es una aplicación debidamente probada, que se entregue al usuario final.

2.4. Ejemplo de utilización de la guía propuesta.

2.4.1. Antecedentes.

Se presenta en esta sección un ejemplo de aplicación del lenguaje de patrones GRN en el modelado de un Sistema de Reparación de Agujeros (SRA), descrito en [9] y reacondicionado a continuación.

2.4.2. Descripción de los requerimientos principales de SRA.

"Los ciudadanos pueden acceder a un sitio Web y relatar la localización y gravedad de los agujeros. A medida que los agujeros son relatados ellos son registrados en un "sistema de reparación del ministerio de obras públicas" y se les otorga un número de identificación, almacenado por dirección de la calle, tamaño (en una escala de 1 la 10), localización (en medio de la calle, en la acera, etc.), sector (determinado por la dirección de la calle) y prioridad de reparación (determinada por el tamaño del agujero). Datos de la orden de servicio son asociados con cada agujero e incluyen la localización y tamaño del agujero, número de identificación del equipo de reparación, número de personas en el equipo, equipamiento atribuido, horas aplicadas en la reparación, estado del agujero (trabajo en marcha, reparado, reparación temporal, no reparado), cantidad de material usado y coste de la reparación (calculado a partir de horas aplicadas, cantidad de personas, material y equipamiento usados). Finalmente, un archivo de daños es creado para contener información sobre daños relatados debido al agujero e incluyen nombre del ciudadano, dirección, número del teléfono, tipo de daño y cuantía en pesos del perjuicio causado por el daño. SRA es un sistema on-line, por lo tanto, todas las consultas deben ser hechas interactivamente".

2.4.3. Aplicación de GRN para modelado de SRA.

Con base en los requerimientos de SRA, GRN fue utilizada con el objetivo de modelar el sistema, produciendo un modelo de clases con la indicación de los papeles desempeñados por cada clase en los patrones utilizados. Además, se produce un histórico de los patrones y variantes utilizados, así como una lista de decisiones de análisis tomadas cuando el lenguaje de patrones no atiende a los requerimientos del sistema.

Conforme recomienda el proceso presentado en la sección 1.5.2, el lenguaje de patrones GRN debe ser estudiado previamente por el desarrollador y los requerimientos del sistema deben ser obtenidos usando alguna de las técnicas de la Ingeniería de Software. En el caso de SRA, se consideran los requisitos contenidos en el texto presentado en esta sección. Dos sub-sistemas pueden ser identificados en los requerimientos: la reparación del agujero y el control de daños causados a los ciudadanos. Por lo tanto, GRN debe ser aplicado en dos ciclos, cada uno de ellos para uno de los sub-sistemas. El modelo de análisis final que fue producido aplicando GRN a SRA es mostrado en la figura 2.6.

Los comentarios indican los patrones de GRN usados para modelar SRA, siendo que una misma clase puede desempeñar papeles diferentes en más que un patrón. El formato utilizado en el interior de los comentarios es "P#n: papel", donde "n" es el número del patrón y "papel" es el papel desempeñado por la clase en tal patrón. Se Debe destacar que los métodos y las operaciones canónicas no son incluidos.

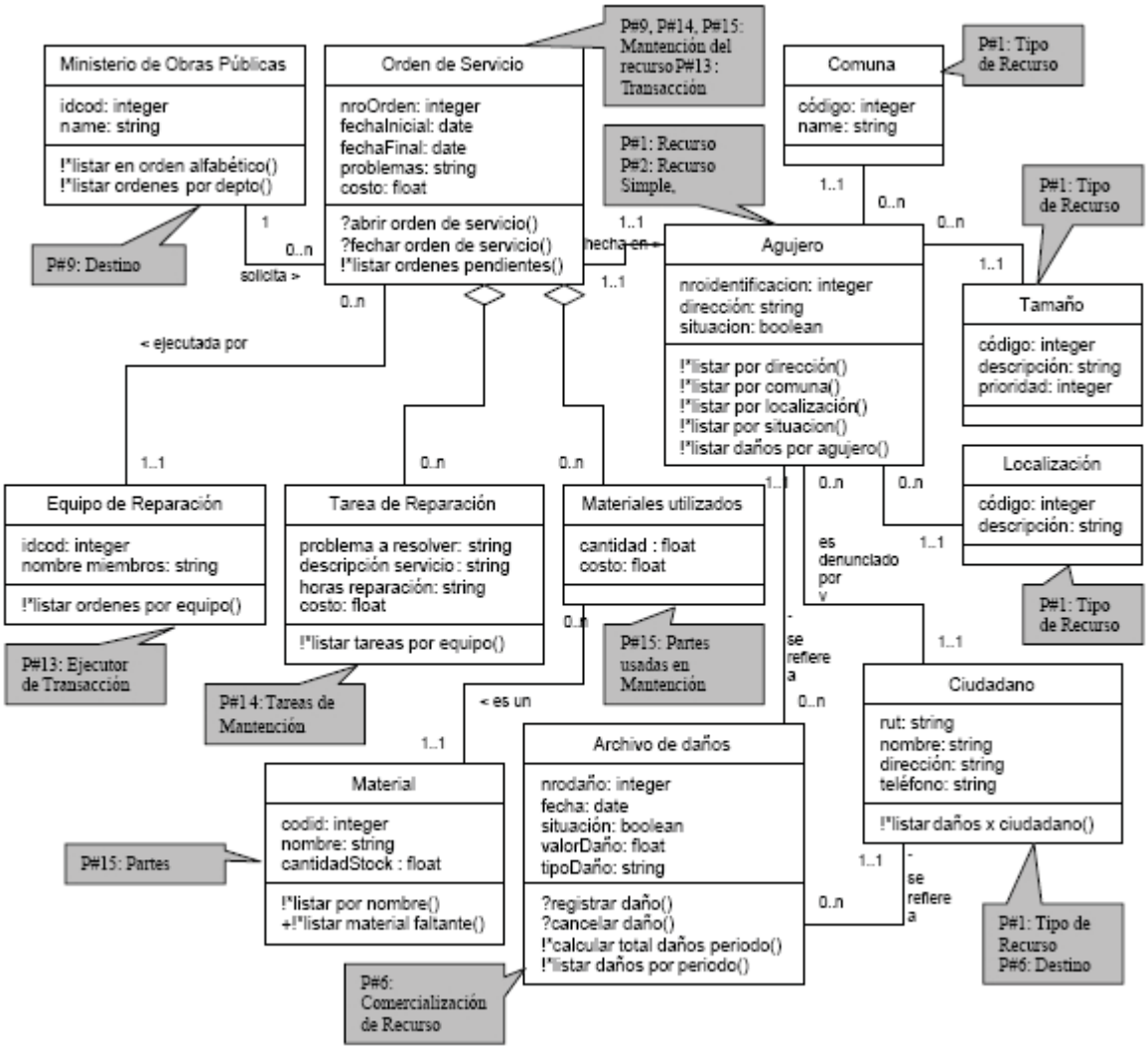
El modelo de la figura 14 fue obtenido de forma gradual, a medida que los patrones fueron aplicados. Comenzando por el sistema de reparación de agujeros, el primer patrón de GRN — IDENTIFICAR EL RECURSO — es aplicado, siendo que el Agujero es identificado como el Recurso. Las posibles clasificaciones del agujero, por ejemplo, con relación al sector, al tamaño y a la localización, llevan a la aplicación de la variante "Múltiples Tipos de Recursos" de ese patrón. Se aplica entonces el patrón 2 – CUANTIFICAR EL RECURSO, en el cual el agujero puede ser caracterizado como simple, pues posee características que llevan a controlarlo individualmente (otras opciones, tales como recursos mensurables, instanciados o tratados en lotes, son inadecuadas en este ejemplo).

Después de identificado y cuantificado el recurso, se inicia el modelado de la gestión del recurso. En ese caso en particular, el tipo de gestión deseado para el agujero es la reparación, que se ensambla en el patrón 9 – MANTENER EL RECURSO. Analizando detalladamente ese patrón, se percibe que uno de los participantes opcionales — Origen — puede ser removido, ya que no es importante para el sistema que sean registradas unidades específicas del Ministerio de Obras Públicas. Así, el papel de Destino es atribuido al Ministerio de Obras Públicas, que, en verdad desempeña también el papel de Origen, porque es él quien recibe una solicitud de reparación de agujero y, después, realiza tal reparación.

Finalmente se aplican los patrones para complementar detalles de la transacción de reparación del agujero, tales como: el patrón 13 – IDENTIFICAR EL EJECUTOR DE LA TRANSACCIÓN, el cual puede lidiar con el equipo de reparación responsable por el agujero; el patrón 14 – IDENTIFICAR LAS TAREAS DEL MANTENIMIENTO, que permite la discriminación de cada una de las tareas necesarias para la reparación del agujero; y el patrón 15 – IDENTIFICAR LAS PIEZAS DEL MANTENIMIENTO, que permite registrar cada uno de los materiales consumidos en la reparación.

Durante el análisis del sistema usando GRN, se hace necesario, conforme se recomienda en el apéndice A, hacer la correlación entre los atributos, métodos y operaciones del patrón versus los atributos, métodos y operaciones de las clases concretas del sistema siendo modelado. En el caso de SRA, nuevos atributos fueron añadidos, como por ejemplo el atributo numeroDePersonasEquipo (clase Orden de Servicio).

Figura 2.6. Modelo de análisis de SRA con patrones.



Habiendo modelado el sub-sistema de reparaciones, se parte ahora para el sub-sistema de control de los perjuicios causados por los agujeros. Se analizan los aspectos semánticos de GRN, se puede concluir que ese sub-sistema se refiere a un tipo de transacción no cubierta por GRN: el control del perjuicio no se ensambla en ninguna de las tres principales actividades abordadas en GRN: no se trata de un arriendo, en el cual un recurso cambia temporalmente de propietario, no se trata tampoco de una comercialización, en que la posesión del recurso es transferida hacia otro propietario, y tampoco se trata de un mantenimiento, en la cual un recurso con problemas pasa por servicios y cambios de piezas para volver a funcionar.

Sin embargo, si analizáramos sólo la sintaxis de los patrones de GRN y que comparemos los atributos envueltos en el control de los perjuicios causados por los agujeros, vemos que la clase Comercialización del Recurso, del patrón 6 – COMERCIALIZAR EL RECURSO,

posee semejanzas que la hacen atractiva para ser reutilizada en el modelado de ese sub-sistema. Hechas las debidas adaptaciones, el patrón 6 es entonces utilizado para concluir la etapa de análisis de SRA. Ese tipo de uso de un patrón es denominado "abstracción del dominio", o, en inglés, "flexing". La pre-condición para hacer la abstracción de dominio es que el patrón posea la estructura computacional requerida, aunque no cargue la semántica del dominio anticipada por su diseñador. En este trabajo, llamamos a eso "proceso de sustitución semántica/sintáctica", y usamos, en la Tabla 5, la referencia general al patrón (USAR EL RECURSO).

El resultado del modelado de SRA usando GRN es el modelo de clases mostrado en la figura 2.6 y el histórico de patrones usados en el modelado (Tabla 1). Otro posible resultado del modelado podría haber sido una lista de las decisiones de análisis tomadas durante el proceso, en la cual estarían discriminados los requisitos no atendidos por GRN y como tales requisitos fueron modelados (por medio de clases adicionales, relaciones, etc.). Esa lista de decisiones sería importante durante la implementación de la aplicación específica y en su futuro mantenimiento. Ella no fue necesaria en el caso de SRA, ya que GRN apoyó el modelado de todos sus requerimientos.

Tabla 1. Histórico de patrones utilizados en la instanciación de SRA.

Patrón	Variante	Participante	Clase de Aplicación
1. Identificar el Recurso	Múltiples tipos	Recurso	Agujero
		Tipo de Recurso	Sector
		Tipo de Recurso	Tamaño
		Tipo de Recurso	Localización
		Tipo de Recurso	Persona
2. Cuantificar el Recurso	Recurso Simple	Recurso	Agujero
9. Mantener el Recurso	Sin origen	Recurso	Agujero
		Mantenimiento del Recurso	Orden de Servicio
		Destino	Ministerio de Obras Públicas
13. Identificar el Ejecutor de la Transacción	Sin comisión	Ejecutor de la Transacción	Equipo de Reparación
		Transacción	Orden de Servicio
14 - Identificar las Tareas del Mantenimiento	Ejecutor de la Transacción en vez de ejecutor de la tarea	Mantenimiento del Recurso	Orden de Servicio
		Tarea del Mantenimiento	Tarea de Reparación
15 - Identificar las Piezas del Mantenimiento	Default	Mantenimiento del Recurso	Orden de Servicio
		Pieza usada en el Mantenimiento	Material usado en reparación
		Pieza	Material
6 - Comercializar/Usar el Recurso	Sin origen	Recurso	Agujero
		Comercio del Recurso	Archivo de Perjuicios
		Destino	Persona

2.5. Resumen del capítulo.

En este capítulo se presentó formalmente la guía propuesta para evolucionar frameworks de caja blanca hacia frameworks de caja gris basándose en lenguaje de patrones. La guía fue divida en cinco etapas consecutivas siguiendo el mismo modelo utilizado para construir e instanciar los frameworks de caja blanca. Se mostró que el primer paso es fundamental para obtener una buena evolución del framework, ya que es en ese paso donde determinamos los puntos que deberán evolucionar. Teniendo como base lo realizado en la etapa uno, y siguiendo paso a paso el resto de las etapas, se obtendrá un framework de caja gris, el cual será más fácil de instanciar y por lo tanto más reutilizable.

En este capítulo tambíen se mostró como instanciar un framework de caja gris evolucionado utilizando la guía, y se presentó un ejemplo práctico de utilización de la guía, con un sistema de reparación de agujeros.

Capítulo 3. Validación del aporte a la reutilización de la guía propuesta.

3.1. Introducción.

La medición a través de métricas es algo muy común en el mundo de la ingeniería, desgraciadamente el mundo de la ingeniería de software está lejos de esta realidad ya que encontramos dificultades sobre qué medir y cómo interpretar los resultados de las mediciones realizadas.

Las razones para evaluar los resultados al utilizar la guía que fue propuesta son:

1. Para indicar la calidad del producto.
2. Para evaluar los beneficios en términos de productividad y de calidad, derivados del uso de nuevos métodos y herramientas de la ingeniería de software.
3. Para ayudar a justificar el uso o construcción de nuevas herramientas

Las métricas en la ingeniería de software son las que se relacionan con el desarrollo del software, las que miden aspectos como funcionalidad, complejidad y eficiencia. Dentro de éstas serán utilizadas: métricas técnicas y métricas de productividad [18].

Métricas técnicas: Se centran en las características de software por ejemplo: la complejidad lógica, el grado de modularidad. Mide la estructura del sistema, el cómo esta hecho.

Métricas de productividad: Se centran en el rendimiento del proceso de la ingeniería del software. Es decir que tan productivo va a ser el software que se diseñará.

3.2. Formulación de métricas.

Existen una gran cantidad de métricas que son utilizadas para capturar atributos del software de una forma cuantitativa, sin embargo, son muy pocas las que han sobrevivido a la fase de formulación. Esto se debe a múltiples problemas, entre ellos [19]:

Las métricas no se definen siempre en un contexto en el que el objetivo de interés que se pretende alcanzar.

Incluso si el objetivo es explícito, las hipótesis experimentales a menudo no se hacen explícitas.

Las definiciones de métricas no siempre tienen en cuenta el entorno o contexto en el que serán aplicadas.

No siempre es posible realizar una validación teórica adecuada de la métrica porque el atributo que queremos medir no siempre está bien definido.

Un gran número de métricas nunca se ha validado empíricamente.

Para tener una buena formulación de las métricas es importante:

Definir claramente el objetivo que queremos alcanzar mediante la definición de métricas.

Preguntarse cómo, a través de las características específicas del producto a medir, podemos alcanzar nuestro objetivo.

Intentar responder a cada una de esas preguntas mediante la definición de métricas.

Muchas métricas de la ingeniería de software se orientan a la medición de la utilización real, pero sólo unas pocas de ellas describen la reutilización directamente. Utilizaré cinco de las seis métricas clásicas de Chidamber y Kemerer que pueden ser usadas para medir reutilización: *Métodos Ponderados por Clase* (MPC), *Profundidad del árbol de herencias* (PAH), *Acoplamiento entre Objetos de las Clases* (AOC), *Falta de Cohesión de los Métodos* (FCM) y *Número de hijos por clase* (NHC) [5]. Se omitió la métrica *Respuesta a una Clase (RC)* por su dificultad de medición y sólo determina el nivel de complejidad y no el nivel de reutilización.

La métrica de los MPC resulta de la agregación de las complejidades de los métodos de una clase dada. Dicha métrica podría ser utilizada como medida de predicción de la reutilización de la clase desde el punto de vista de que “clases con un gran número de métodos tendrían una mayor dificultad para ser aplicadas en más de una aplicación específica, lo que limitaría sus posibilidades de reutilización”.

Considerando una clase C_i, con métodos M_1,… M_n que son definidos en las clases. Sea c_1,…, c_n la complejidad de los métodos. Entonces:

$$\mathrm{MPC} = \sum_{i=1}^{n} c_i$$

Base teórica: Puesto que los métodos son característicos de las clases de objetos y la complejidad es determinada por la cardinalidad de su sistema de características. El número de métodos es, por lo tanto, una medida de la definición de la clase así como pueden ser atributos de una clase, puesto que los atributos corresponden a características.

Puntos de vista:

1. El número y la complejidad de los métodos pueden predecir el tiempo para desarrollar/mantener la clase.
2. Cuanto mayor sea el nº de métodos mayor impacto potencial tendrá en los hijos, sus herederos potenciales.
3. Las clases con gran nº de métodos serán de aplicación específica, y por lo tanto más difíciles de reutilizar.

La métrica PAH es la cuenta de la profundidad en el árbol de herencias de una clase dentro de un marco de software. Esta métrica está relacionada con la reusabilidad desde el punto de vista de que "clases que son más profundas en el árbol de herencia son más complejas (debido a que tienen probablemente, heredadas más características), y así tienen menos predisposición a ser reutilizadas". Aquí el concepto clave es la herencia.

Base teórica: PAH es una medida de cuántas clases antecesoras pueden potencialmente afectar el comportamiento de la clase medida.

Puntos de vista:

1. Mayor profundidad de la clase implica que más métodos pueden heredar su comportamiento, por lo tanto, más difícil de explicar su comportamiento.
2. Es un aspecto de la complejidad del diseño.
3. A mayor profundidad de una clase mayor posibilidad de reutilización de métodos heredados.

La métrica AOC es el número de clases con las que la clase considerada se encuentra acoplada. Altos AOC impiden la reutilización ya que van en contra del diseño modular.

Base teórica: AOC se relaciona con la noción que un objeto está acoplado a otro si uno de ellos actúa sobre el otro, es decir, métodos de uno usan métodos o instancias de variables del otro. Según lo indicado anteriormente, puesto que los objetos de la misma clase tienen las mismas características, dos clases se acoplan cuando los métodos declarados de una clase usan métodos o instancias de variables definidos por otra clase.

Puntos de vista:

1. Cuanto mayor es, peor es la modularidad y la reutilización.
2. Cuanto mayor es, peor es el encapsulamiento y más cuesta mantenerlo.
3. Está relacionado con la complejidad de las pruebas (más recursos).

La métrica FCM es una medida de la carencia de solapamiento en el uso de los atributos de una clase. Si los métodos utilizan subconjuntos separados de atributos de la clase, se podría hacer la hipótesis de que los métodos no están correctamente agrupados, y que probablemente la clase se podría dividir en muchas más. Clases con un alto FCM dificultan la reutilización ya que aumentan la dificultad de comprensión.

Métodos de C: $M_1 ... M_n$

A_i= {variables usadas por M_i}

$P = \{(A_i, A_j) / A_i \cap A_j = \emptyset\}$

$Q = \{(A_i, A_j) / A_i \cap A_j \neq \emptyset\}$

FCM(C)= | P | - | Q | si | P | > | Q |

FCM(C)= 0, en otro caso

Base teórica: Esto utiliza la noción del grado de semejanza de métodos. El grado de semejanza para dos métodos M_1 y M_2 en la clase C_1 esta dado por:

$\sigma ( ) = \{ I_1 \} \cap \{ I_2 \}$ donde $\{ I_1 \}$ y $\{ I_2 \}$ son los conjuntos de instancias de variables usadas por M_1 y el M_2.

Puntos de vista:

1. la cohesión de métodos dentro de una clase es deseable, puesto que promueve la encapsulación.
2. La carencia de la cohesión implica que las clases se deben partir probablemente en dos o más subclases. Baja cohesión aumenta la complejidad, de tal modo aumentando la probabilidad de errores durante el proceso del desarrollo.

Finalmente, la métrica NHC es la medida del número de subclases inmediatas subordinadas a una clase en la jerarquía de clases.

Base teórica: NHC se relaciona con la noción del alcance de características. Es una medida de cuántas subclases van a heredar los métodos de la clase padre.

Puntos de vista:

1. Mientras mayor sea el número de hijos, más compleja será la reutilización, ya que es más complejo explicar el comportamiento de sus métodos heredados.
2. A mayor número de hijos, mayor es la probabilidad de abstracción incorrecta de la clase padre. Si una clase tiene una gran cantidad de hijos, quizá es un caso de un uso erróneo de herencia.
3. El número de hijos da una idea de la influencia potencial que una clase tiene en el diseño. Si la clase tiene una gran cantidad de hijos, puede requerir la prueba de métodos en la clase.

3.3. Criterios de evaluación de las métricas.

Varios investigadores recomiendan las características que las métricas de software deben poseer para aumentar su utilidad, por ejemplo, se sugiere que las métricas sean sensibles a las diferencias externamente observables en el ambiente de desarrollo, y deben también corresponder a las nociones intuitivas sobre las diferencias características entre los artefactos del software que son medidos. La mayoría de las características recomendadas son cualitativas en naturaleza y por lo tanto, la mayoría de las propuestas para métricas han tendido a ser informales en sus evaluaciones.

Al querer tener una métrica mas rigurosa es deseable tener fijos los criterios con los cuales evaluar. De las nueve propiedades que Weyuker [20] propone, sólo seis serán explicadas acá para definir un criterio de evaluación, debido a que tres de ellas no aplican al objetivo que apuntan las métricas definidas.

Característica 1: Noncoarseness: Dado una clase P y una métrica μ otra clase Q puede siempre ser encontrada tal que: $\mu(Q) \neq \mu(P)$. Esto implica que no todas las clases tienen el mismo valor para una métrica, si no ha perdido su valor como medida.

Característica 2: Nonuniqueness: (noción de la equivalencia) Pueden existir clases distintas P y Q tales que:

$\mu(P) = \mu(Q)$. Esto implica que dos clases pueden tener el mismo valor métrico, es decir, las dos clases son igualmente complejas.

Característica 3: Los detalles del diseño son importantes: Dados dos diseños de clases, P y Q, que proporcionan la misma funcionalidad, no implica que $\mu(P) = \mu(Q)$. Las especificaciones de las clases deben influenciar el valor métrico. La intuición detrás de la característica 3 es que aunque dos diseños de clases realizan la misma función, lo importante en el diseño es determinar el métrico para cada clase.

Característica 4: Monotonicity: Para todas las clases P y Q, $\mu(p) \leq \mu(P+Q)$ y $\mu(Q) \leq \mu(P+Q)$ donde P+Q implica la combinación de P y de Q. Esto implica que el métrico para la combinación de dos clases nunca puede ser menos que el métrico para cualquiera de las clases componentes.

Característica 5: Operación de no equivalencia de interacción: Existen P, Q y R tales que:

$\mu(P) = \mu(q)$ no implica que $\mu(P+R) = \mu(Q+R)$. Esto sugiere que la interacción entre P y R puede ser diferente que la interacción entre Q y R resultando en diversos valores de complejidad para P+R y Q+R.

Característica 6: La Interacción Aumenta la Complejidad: Existen P y Q tales que: $\mu(P)+\mu(Q) < \mu(P+Q)$. Este principio detrás de esta característica es que cuando se combinan dos clases, la interacción entre las clases puede aumentar el valor métrico de la complejidad.

Supuesto 1:

X_i = el número de métodos en una clase dada i.

Y_i = el número de métodos llamados desde un método dado i.

Z_i = el número de variables instanciadas por un método i.

C_i = el número de parejas entre una clase dada de objetos i y el resto de las clases.

X_i, Y_i, Z_i, C_i son variables al azar discretas cada uno caracterizada por una cierta función de distribución general. Además, todo X_i es independiente e idénticamente distribuido (i.i.d). Lo mismo es verdad para todo Y_i, Z_i y C_i. Esto sugiere que el número de métodos, de variables y de acoples siguen una distribución estadística no evidente a un observador del sistema. Además, el observador no puede predecir variables, métodos, etc. de una clase basada en el conocimiento de las variables, métodos y acoples de otra clase en el sistema.

Supuesto 2: En general, dos clases pueden tener un número finito de métodos "idénticos" en el sentido que la combinación de las dos clases resultaría en la versión de una clase de los métodos idénticos que llegan a ser redundantes. Por ejemplo, una clase "foo_1" tiene un método "dibujar" que es responsable de dibujar un icono en una pantalla; otra clase "foo_2" también tiene un método del "dibujar". Ahora un diseñador decide tener una sola clase "foo" y combina las dos clases. En vez de tener dos métodos "dibujar" distintos el diseñador puede decidir tener un solo método "dibujar" (no obstante modificado para reflejar la nueva abstracción).

Supuesto 3: El árbol de la herencia está "lleno", es decir, hay una raíz, nodos intermedios y hojas. Este supuesto indica simplemente que un uso no consiste solamente en clases independientes; hay un cierto uso de subclases.

3.4. Medición de aplicación de la guía propuesta.

3.4.1. Consideraciones iniciales.

Para validar la guía se realizará la medición de partes del diagrama de clases del frameworks de caja blanca para Gestión de Recursos de Negocios (GREN), el cual se basa en el lenguaje de patrones para el mismo dominio (GRN), comparando los resultados con los obtenidos al medir las mismas partes del diagrama luego de aplicarles la guía para la evolución de frameworks de caja blanca hacia frameworks de caja gris que aquí se propone.

3.4.2. Diagramas de clases a medir.

Figura 3.1. Parte del diagrama de clases del framework de caja blanca GREN [3].

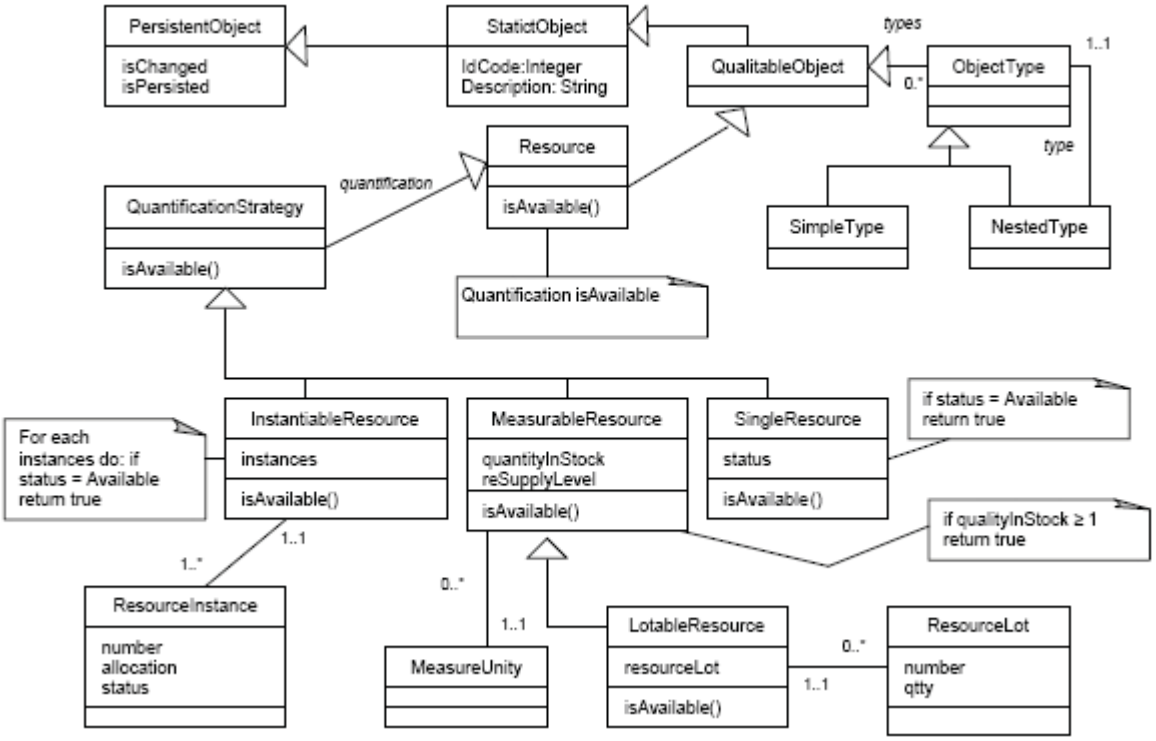
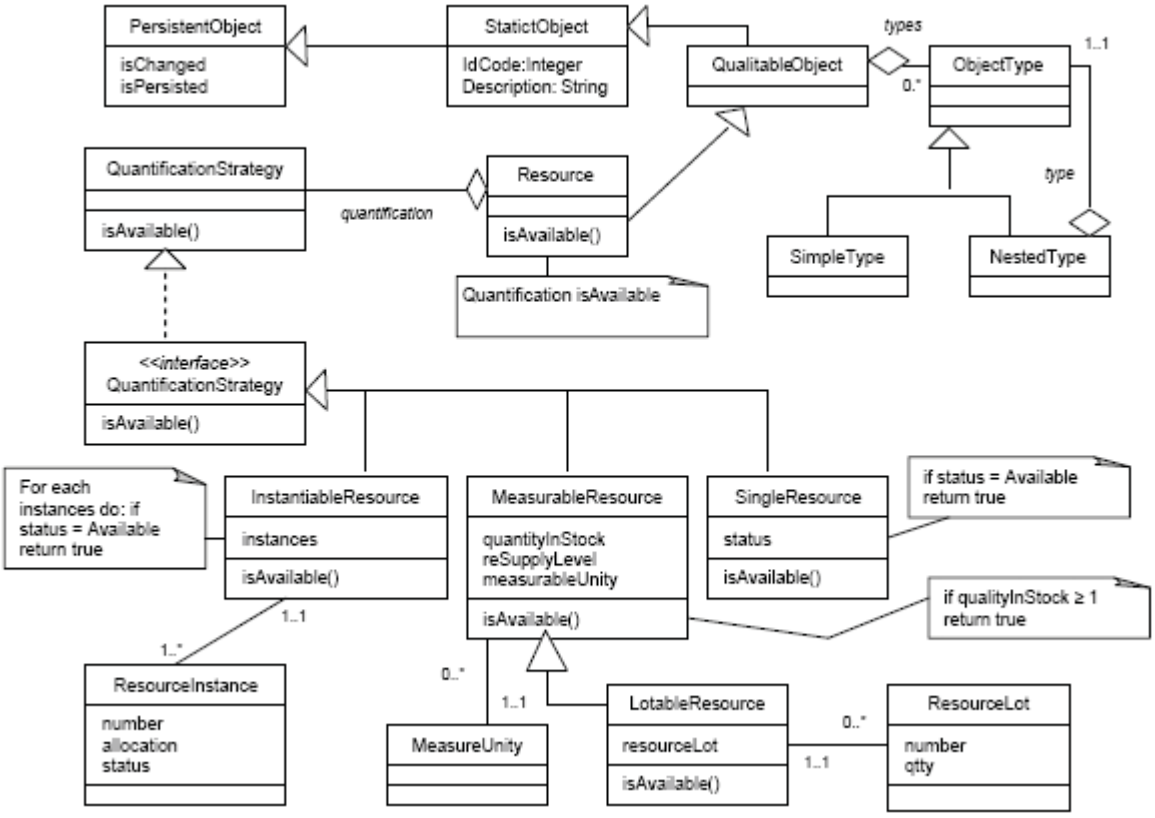


Figura 3.2. Parte del diagrama de clases del framework de caja gris GREN.



3.4.3. Resultados de la medición.

Métodos ponderados por clases:

En la sumatoria ponderada de los métodos por clases de la figura 3.1, correspondiente a parte del diagrama de clases del framework de caja blanca GREN basado en el lenguaje de patrones GRN, se identifican tres métodos *IsAvailable( ), IsChanged( )* y *IsPersisted( ).*

De igual manera, en la sumatoria ponderada de los métodos por clases de la figura 3.2, correspondiente a parte del diagrama de clases del framework de caja gris GREN basado en el lenguaje de patrones GRN, también se aprecian los métodos *IsAvailable( ) , IsChanged( )* y *IsPersisted( ).*

Tabla 2: Comparación en las mediciones de ambos diagramas para la métrica MPC.

FIGURA	MÉTRICA	MÁXIMO	MÍNIMO
5	MPC	3	0

6	MPC	2	0

Profundidad del árbol de herencias:

En la profundidad del árbol de herencia es notable la diferencia que se produce entre ambos diagramas, debido a, que precisamente en este punto es donde centra su atención la evolución de frameworks de caja blanca (basado solamente en herencia) hacia frameworks de caja gris (basado en herencia y composición), como se puede apreciar en las mediciones realizadas.

Tabla 3: Comparación en las mediciones de ambos diagramas para la métrica PAH.

FIGURA	MÉTRICA	MÁXIMO	MÍNIMO
5	PAH	7	0
6	PAH	5	0

Acoplamiento entre objetos de clases:

En la revisión del acoplamiento entre objetos de clases de la figura 3.1, correspondiente a parte del diagrama de clases del framework de caja blanca GREN basado en el lenguaje de patrones GRN, se identifica un sólo métodos *IsAvailable( ),*que se acopla entre distintas clases.

De igual manera, en la revisión del acoplamiento entre objetos de clases de la figura 3.2, correspondiente a parte del diagrama de clases del framework de caja gris GREN basado en el lenguaje de patrones GRN, también se aprecia un el método *IsAvailable( ).*

Tabla 4: Comparación en las mediciones de ambos diagramas para la métrica AOC.

FIGURA	MÉTRICA	MÁXIMO	MÍNIMO
5	AOC	1	0
6	AOC	1	0

Número de hijos por clase.

En la revisión del número de hijos por clases de las figura 3.1 y 3.2, correspondientes a parte del diagrama de clases del framework de caja blanca y de caja gris correspondientemente, se identifica:

Tabla 5: Comparación en las mediciones de ambos diagramas para la métrica NHC.

FIGURA	MÉTRICA	MÁXIMO	MÍNIMO
5	NHC	15	0
6	NHC	7	0

3.4.4. Discusión de los resultados obtenidos.

Como se pudo apreciar y de acuerdo a los criterios de medición explicados anteriormente, la guía para la evolución de frameworks es de mucha ayuda para mejorar la reutilización de componentes, ya que nos presenta:

Menos métodos ponderados por clases, por lo tanto, más fácil de reutilizar y menor costo de desarrollo/mantención.

Menor profundidad de árbol de herencias, menor complejidad para explicar el comportamiento de las clases.

Menor acoplamiento entre clases implica mayor modularidad y reutilización.

Mayor cohesión, por lo tanto, mayor encapsulamiento.

Menor cantidad de hijos por clases implica menor complejidad y menos pruebas de sus métodos.

Es de esperar que la construcción de una herramienta que asista la evolución sea el paso necesario para lograr la confirmación del aporte hecho en materia de reutilización de software y la importancia de este tema en el desarrollo de software de gran escala.

3.5. Resumen del capítulo.

Tan importante como proponer una guía, es importante validar que la guía propuesta cumpla los objetivos planteados y sea un real aporte a la reutilización de componentes y con esto a la ingeniería de software. En este capítulo se presentó la manera en que se validó la correlación entre lo propuesto y lo esperado. La validación se llevo a cabo, aplicándole métricas de ingeniería de software a frameworks evolucionados utilizando la guía propuesta. Las métricas utilizadas fueron cinco de las seis métricas clásicas de Chidamber y Kemerer, las cuales miden la reutilización, modularidad, encapsulamiento y complejidad.

En este capítulo, también se presentó como ejemplo parte de las mediciones realizadas y la discusión de los resultados obtenidos.

Capítulo 4. Construcción de una herramienta que apoye la evolución de Frameworks.

4.1.Introducción.

La dificultad de reutilización de los frameworks, por lo tanto, su evolución para mantenerse de acuerdo a los requisitos de sistemas específicos, fue uno de los factores fundamentales para la motivación de este trabajo. La guía propuesta para la evolución de frameworks con base en un lenguaje de patrones, sistematiza esa tarea y la hace sensible de ser ejecutada por desarrolladores no familiarizados con la estructura interna del framework. Sin embargo, tal actividad envuelve trabajo repetitivo y bastante conocimiento del paradigma de Orientación de Objetos. Por otro lado, la sistematización de esa tarea, que se hizo viable después de la creación de una guía para la evolución, motiva la construcción de una herramienta que asista ese proceso, específicamente es de interés asistir el punto uno de la guía propuesta, él cual trata acerca de la identificación de puntos y variables comunes dentro de las especificaciones de los sistemas instanciados por el framework de caja blanca y su mapeo con las clases que implementa el framework para saber qué métodos brindan tal funcionamiento, qué clases implementan los métodos descritos y cuál es la relación entre las clases, para aportar la información necesaria para evolucionar el diagrama de clases del framework de caja blanca basado solamente en herencia en el diagrama del framework de caja gris basado en herencia y composición.

4.2. Alternativas para la construcción de una herramienta.

En general, la construcción de una herramienta para asistir la evolución de frameworks es una tarea totalmente direccionada a las características específicas del frameworks. Como la implementación de la herramienta se relaciona directamente con el framework, muchas veces el coste de su construcción se justifica si el framework que será evolucionado servirá para instanciar un número elevado de aplicaciones. Sin embargo, las peculiaridades del framework evolucionado según la guía propuesta en este trabajo, lo hace distinto de otros frameworks,

posibilitando que la construcción de una herramienta para asistir su evolución sea hecha por un proceso bien definido y cuyo resultado pueda ser reutilizado por otro frameworks.

El proceso para construcción de una herramienta puede variar dependiendo de la cobertura deseada para la herramienta obtenida, así como lo muestra la tabla 1. Dos alternativas pueden ser utilizadas, que son denominadas alternativas "A" y "B". Cuando se pretende utilizar la herramienta para ayudar en la evolución de un único par "lenguaje de patrones y framework" (por ejemplo, un framework que es utilizado para la generación de múltiples aplicaciones de un dominio específico), entonces se utiliza la alternativa "A", en la cual se hace el proyecto de la herramienta a medida, por lo tanto, totalmente personalizada para el framework y su lenguaje de patrones asociado. Eso hace a la herramienta menos flexible en términos de facilidad de adaptación a otros frameworks, pero por otro lado, la hace más fácil de construir y usar.

Cuando se pretende especializar la herramienta para usarla con varios lenguajes de patrones y frameworks, entonces es deseable que el proceso de construcción considere las debidas generalizaciones que deben ser hechas para permitir su adaptación y uso con diferentes pares "frameworks y lenguajes de patrones", lo que constituye la alternativa "B".

Tabla 6: Alternativas posibles para construcción de una herramienta.

CONSTRUCCIÓN DE UNA HERRAMIENTA			
Alternativa	**Ventajas**	**Desventajas**	**Usar cuando…**
"A", herramienta específica para un par "frameworks y lenguaje de patrones".	Mayor facilidad de construcción y uso de la herramienta, mayor eficiencia.	Sirve solamente para un frameworks específico.	La herramienta servirá para instanciar uno o pocos frameworks.

"B", herramienta genérica, que debe ser adaptada a cada par "frameworks y lenguaje de patrones".	Adaptabilidad a otros frameworks y leguajes de patrones, sabiendo usar para un framework, automáticamente se sabe usar para otros, mayor facilidad de integración de frameworks.	Mayor complejidad de la construcción y el uso, necesidad de adaptación, posible desempeño inferior, necesidad de entrenamiento inicial.	Se pretende utilizar la herramienta para evolucionar más de un framework.

En este trabajo se explorarán las dos alternativas, comenzando por la alternativa "A", que es la construcción de una herramienta específica para un framework y un lenguaje de patrones, seguida de la alternativa "B", en la cual se muestra cómo construir una herramienta genérica, que debe ser adaptada para cada par "framework y lenguaje de patrones". La primera alternativa será explorada con un mayor nivel de detalle, por ser más simple de desarrollar y usar y presentaría una mayor eficiencia para un par frameworks y lenguaje de patrones específico.

4.3. Análisis de la construcción de una herramienta específica.

El proceso de construcción de una herramienta específica para un framework y su lenguaje de patrones sigue los pasos que se detallan a continuación. Para cada patrón del lenguaje de patrones, se debe permitir que el usuario lo aplique (o a cualquiera de sus variables), indicando los papeles desempeñados por cada clase de la aplicación. La información sobre el uso de los patrones en el modelo de la aplicación específica deben quedar almacenados para su uso en el fututo, ya sea para la construcción de aplicaciones con base en otras similares ya registradas, o para la alteración de una aplicación para fines de reingeniería o mantenimiento.

Se propone la arquitectura de la herramienta a construir, en la cual se dispone de una interfaz gráfica con el usuario, la cual interactúa con los tres módulos que componen la herramienta: un módulo de especificación del dominio, un módulo de especificación del lenguaje de patrones y un módulo generador de pautas para la evolución.

El módulo de especificación del dominio tiene el propósito de registrar en una base de datos los requerimientos comunes a las aplicaciones del dominio.

El módulo de especificación del lenguaje de patrones es responsable de permitir la representación, adecuada, del lenguaje de patrones y su mapeo con las clases del framework.

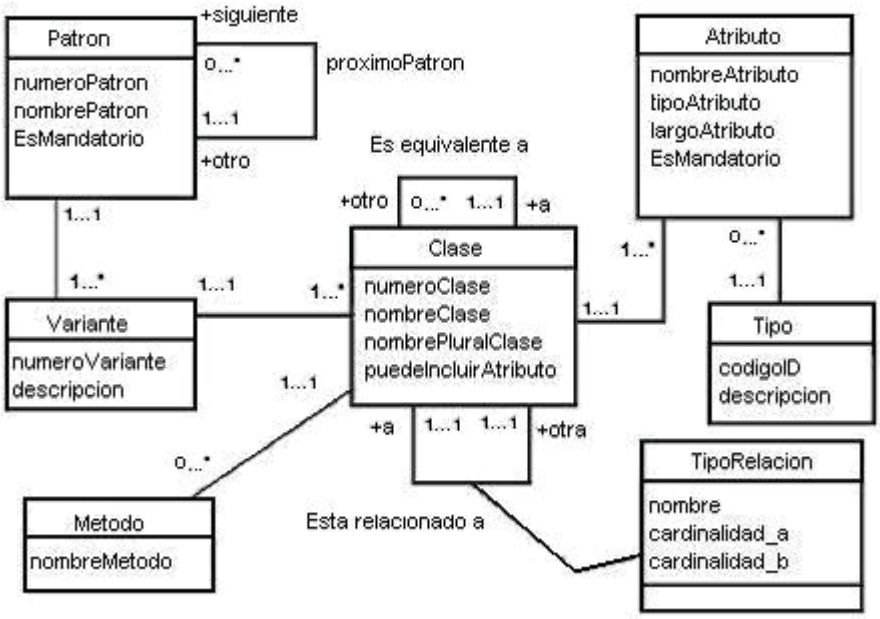
Y por último, el módulo generador de pautas de evolución se basa en la información disponible en las bases de datos de la especificación del dominio y las clases del framework a evolucionar para generar un listado con los métodos que deberán explicar su comportamiento mediante interfaces y las clases que deberán desagregarse en otras clases que se relacionarán mediante composición o agregación. A continuación se describen los pasos que constituyen el proceso de construcción.

4.3.1. Crear un modelo del lenguaje de patrones.

Con base en el lenguaje de patrones específico, se crea un modelo que represente todos sus elementos, principalmente los que sean necesarios para posibilitar la evolución del framework específico. Ellos son: los patrones contenidos en el lenguaje y sus posibles variantes, las clases participantes de cada patrón/variante y los métodos y operaciones de cada patrón/variante. Ese modelo debe ser implementado por la herramienta, por ejemplo, por medio de una base de datos conteniendo esa información.

La figura 4.1 muestra el modelo de clases asociado a la base de datos, en el cual cada clase representa un aspecto importante del lenguaje de patrones como, por ejemplo, patrones, variantes, clases y atributos.

Figura 4.1. Metamodelo para un lenguaje de patrones.



Para dar soporte a ese mecanismo de representación del lenguaje de patrones, se puede crear un sistema específico para registrar la información sobre el lenguaje de patrones, que quedaría disponible al desarrollador de frameworks. Alternativamente, se puede entrar con los datos directamente en la base de datos, utilizando un sistema administrador de bases de datos.

4.3.2. Crear un mapeo para el lenguaje de patrones y el framework.

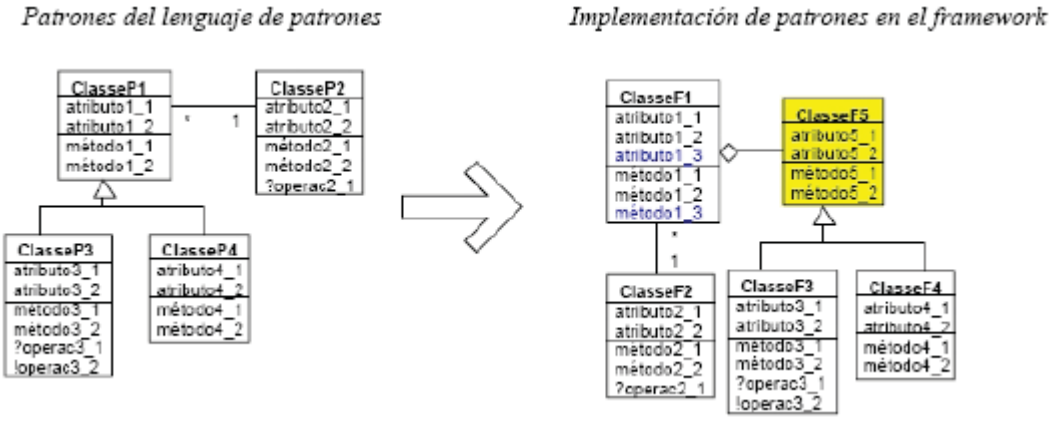
El segundo paso en la construcción de la herramienta es suministrar medios para mapear, de forma asistida, cada clase participante de los patrones y variantes del lenguaje de patrones, para las respectivas clases del framework que la implementan. Eso incluye la especificación de los métodos del framework que deben ser sobrepuestos de acuerdo con el uso de los patrones.

De entre las características comunes encontradas en frameworks desarrollados con base en un lenguaje de patrones se pueden citar:

1. Clases de los patrones poseen una o más clases en el framework correspondiente. Por ejemplo, en la figura 8, la clase ClasseP3, que en el patrón poseía el comportamiento de la clase ClasseP1, en el framework posee también el comportamiento de la nueva ClasseF5. Así, al evolucionar el framework de caja blanca, como la reutilización es hecha por medio de herencia y composición, puede ser necesario crear una o más clases. Ese mapeo puede ser

implementado por medio de una tabla que muestre, para cada variante del patrón, las clases del framework que deben ser especializadas durante la evolución. De ese modo, la herramienta será capaz de asistir la creación de esas clases.

Figura 4.2. Ilustración de correspondencia entre un patrón y su implementación en el framework.



2. Clases abstractas del framework poseen métodos-gancho que deben ser sobrepuestos por sus clases descendientes. Ese mapeo puede ser implementado por medio de una tabla conteniendo tales métodos, con indicación de los criterios usados para sobreponer. Es posible que la herramienta gestione el código de esos métodos de forma asistida, pero eso debe ser tratado caso a caso. Como mínimo, manteniendo ese mapeo en una tabla, se puede suministrar al desarrollador una lista con los métodos que sean sobrepuestos.

3. Clases del framework evolucionado pueden poseer atributos adicionales, que deben ser debidamente tratados. Por ejemplo, las clases especializadas a partir del framework heredan los atributos de las súper-clases y se crean interfaces para mostrar su comportamiento, y opcionalmente, incluyen nuevos atributos, que deben tener, por lo menos, métodos constructores, de atribución de valor y de lectura de valor. Ya que es posible identificar cuáles atributos fueron incluidos, pues el meta-modelo de la figura 4.1. permite que un atributo de una clase de la aplicación no esté conectado a un atributo del patrón, entonces se pude construir una tabla conteniendo la lista de métodos que sea generada por la herramienta para los nuevos atributos.

4. Parte de la funcionalidad del lenguaje de patrones puede no tener implementación correspondiente en el framework. Muchas veces el framework puede implementar

parcialmente algunas de las funcionalidades cubiertas por el lenguaje de patrones. Así, durante el mapeo se pueden crear mecanismos para detectar si la aplicación de un patrón/variante posee implantación correspondiente en el framework. En caso negativo, durante la fase de consistencia de información se puede emitir un mensaje de alerta, impidiendo que tal uso del patrón se efectúe, y que no será posible por el framework apoyar su implementación.

Con base en el framework asociado al lenguaje de patrones, se crea un modelo que mapee los patrones del lenguaje de patrones (junto con sus variantes y participantes) a las clases de framework que los implementan. Dos tipos principales de información son deseables: cuáles son las súper-clases del framework que deben ser especializadas en la nueva aplicación, de acuerdo con los patrones y variantes utilizados, y cuales son los métodos que deben ser sobrepuestos en las clases recién creadas. Ese modelo debe ser implementado de forma que pueda ser recuperado posteriormente por el generador de pautas para la evolución, que dará las directrices para crear un nuevo modelo basado en herencia y composición. Se puede desarrollar un subsistema para permitir el registro del mapeo, pero es probable que el coste no se justifique, ya que el mapeo es incluido una única vez.

4.3.3. Crear una interfaz gráfica con el usuario.

La herramienta construida según el proceso propuesto en este trabajo debe poseer una interfaz gráfica con el usuario (GUI), basada en el lenguaje de patrones asociado al framework. Por lo tanto, se pueden enumerar requisitos para esa GUI, de forma que la herramienta posea todos los elementos básicos necesarios. Los siguientes requisitos forman parte de la funcionalidad que se espera encontrar en una herramienta que asista la evolución de frameworks con base en lenguaje de patrones:

1. La GUI de la herramienta debe permitir al desarrollador de aplicaciones llenar toda la información sobre patrones aplicados, así como posibles variantes y sub-patrones. Los datos sobre la estructura del lenguaje de patrones deben ser cargados a partir de la base de datos creada. Los datos específicos de la aplicación instanciada son almacenados en la base de datos creada.
2. La GUI debe permitir la aplicación de los patrones de forma iterativa, o sea, un patrón puede ser aplicado más de una vez, siempre que respete el orden de aplicación de patrones impuesta por el lenguaje de patrones, de acuerdo con los establecido por la relación entre los patrones, denotado en el meta-modelo de la figura 4.1. por *próximoPatrón*.

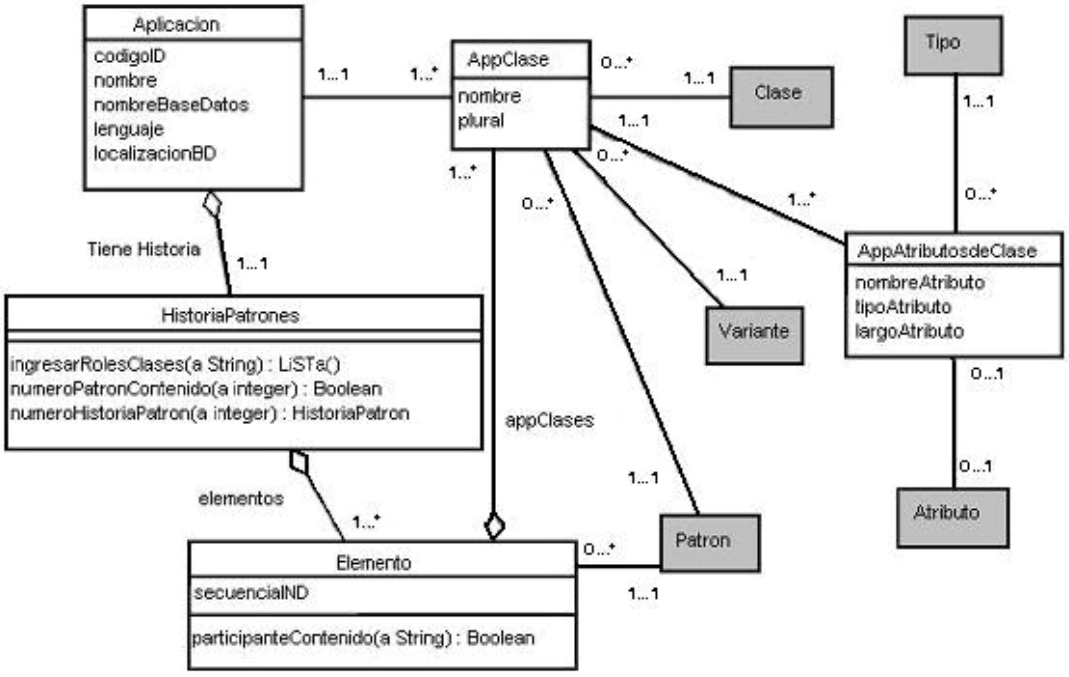
3. La GUI debe permitir que el histórico del uso del lenguaje de patrones sea almacenado en la base de datos, lo que puede ser hecho por medio de un botón u opción en el menú que, cuando se accionado, almacene los datos suministrados por el usuario en la base de datos referente al meta-modelo de la figura 4.1. La información almacenada en la base de datos puede ser utilizada en el futuro para obtener información sobre el uso del lenguaje de patrones, permitiendo la reutilización de esa información en la evolución de frameworks similares. De este modo, es también deseable que se permita crear una lista de las directrices seguidas para evolucionar un framework a partir de las directrices creadas anteriormente, lo que puede ser implementado, por ejemplo, por medio de una opción "Guardar como", en la cual se suministre el nombre de la nueva lista de directrices a ser creada con base en una ya existente.
4. La GUI debe permitir la visualización (por ejemplo, por medio de un informe en pantalla o impreso) del histórico de la utilización del lenguaje de patrones en el modelado de una evolución específica. Eso permitirá al desarrollador verificar los caminos seguidos durante el uso del lenguaje de patrones, en términos de patrones aplicados y papeles desempeñados por las clases de la aplicación específica.
5. La GUI debe permitir que el desarrollador añada atributos a una o más clases participantes de los patrones. En general, las clases de los patrones poseen sólo los atributos comunes a todas las aplicaciones del dominio. Por lo tanto, nuevos atributos pueden ser incluidos de acuerdo con los requisitos del framework específico. Debe ser posible determinar qué clases pueden o no recibir nuevos atributos, pues el framework puede poseer limitaciones en cuanto a eso. Otra funcionalidad deseable es la remoción de atributos referente al meta-modelo de la figura 4.1. que contiene los atributos *puedeIncluirAtributos* (clase Clase) y esMandatorio (clase Atributo).
6. La GUI puede implementar un algoritmo para verificar si el usuario llenó adecuadamente los campos, de acuerdo con la obligatoriedad de las clases y relación entre patrones. Si alguna inconsistencia es encontrada, se debe emitir un aviso para que el desarrollador modifique los patrones ya incluidos o aplique nuevos patrones.
7. Finalmente, la GUI puede ofrecer facilidades de ayuda al usuario, que podrá obtener información sobre los patrones aplicados y sobre el lenguaje de patrones como un todo.

4.3.4. Crear un módulo generador de pautas para la evolución.

Con base en la información sobre el framework específico, la herramienta debe ser capaz de generar las pautas para la evolución de los diagramas de clases, mostrando los métodos más utilizados (por lo menos se debe suministrar una lista de tales métodos, para que los desarrolladores puedan, con base en esa lista, crear el diagrama nuevo manualmente).

Como en esta sección está siendo explorada la alternativa "A" de la tabla 6 para la construcción de la herramienta, entonces existe la posibilidad de optimizar algunas funciones de acuerdo con las características específicas del framework y del lenguaje de patrones, ya que la implementación no queda presa a una arquitectura genérica.

Figura 4.3. Meta-modelo para evoluciones generadas usando un lenguaje de patrones.



La generación de las pautas para la evolución del framework asistirá según el acercamiento propuesto. Para eso, se utiliza la información existente sobre el mapeo entre el lenguaje de patrones y el framework, que permite saber cuáles clases deben ser compuestas, qué interfaces deben ser creadas y cuáles métodos deber ser sobrepuestos. Sin embargo, las directrices mostradas a continuación suministran los principios generales que se deben seguir durante la creación del generador de pautas. Se debe resaltar que tales principios se aplican sólo para la evolución de frameworks de caja blanca basados en lenguaje de patrones, según la guía propuesta en este trabajo.

1. Para identificar las clases del framework que deben ser especializadas durante la evolución, por lo tanto, cuáles serán las clases que serán compuestas y cuáles serán las interfaces que se crearán, la herramienta debe poseer un mecanismo que permita recorrer todas las clases del framework específico. Para cada una de ellas, se debe recoger, en la tabla de mapeo del lenguaje de patrones para el framework, las súper-clases del framework que se separaran en clases compuestas e interfaces. Eso puede ser hecho con base en el histórico de patrones aplicados, que está representado, en el meta-modelo de la figura 9, por la clase *HistoriaPatrones*. Dependiendo de las facilidades ofrecidas por el lenguaje de programación en el cual la herramienta está implementada, esas clases pueden ser listadas.
2. Para cada una de las clases e interfaces nuevas que deban ser creadas, identifique cuáles son los nuevos atributos y que deben, por lo tanto, ser incluidos en la declaración de las clases. Métodos para tratar tales atributos pueden ser descritos en la fase de adaptación de la herramienta.

3. Para cada una de las nuevas clases e instancias creadas, identifique cuáles son los métodos que serán sobrepuestos. Para ello, se debe tomar todas las clases pertenecientes a la jerarquía de las nuevas clases e interfaces y verifique cuáles son sus métodos-gancho. Si el lenguaje de programación permita, esos métodos puedes ser listados. Una forma de implementar esa funcionalidad es por medio de una tabla conteniendo las clases abstractas del framework, sus métodos-gancho, las pre-condiciones para la sobre posición del método y el nombre de un método del generador de pautas.
4. Dependiendo del framework en particular, es posible añadir funciones para asistir la creación de la base de datos para la persistencia de los objetos. Por ejemplo, si el framework utiliza una base de datos relacional en la persistencia de los objetos, entonces un "script" puede ser producido para asistir la creación de las tablas referentes a las nuevas clases e interfaces creadas. El mapeo entre las clases de los patrones y las tablas que serán creadas puede ser implementado de forma de la facilitar la creación de los "script".

4.4. Diseño de una herramienta específica.

4.4.1. Consideraciones Iniciales.

Como se presentó en el análisis de la construcción de una herramienta específica, hecho en el punto 4.3, se han definido tres módulos que deben ser diseñados: un módulo de especificación del dominio, un módulo de especificación del lenguaje de patrones y un módulo generador de pautas de evolución. Para llevar a cabo el diseño de los tres módulos mencionados, se utilizarán técnicas de UML para el diseño de software, tales como diagramas de casos de uso, de secuencia y de clases. A continuación se describe el diseño de cada uno de los módulos mencionados.

Diagramas de casos de uso.

A continuación se presentan los diagramas de caso de uso que describen el modelamiento del prototipo a desarrollar.

Figura 4.4. Caso de uso de alto nivel - Relación con la herramienta.

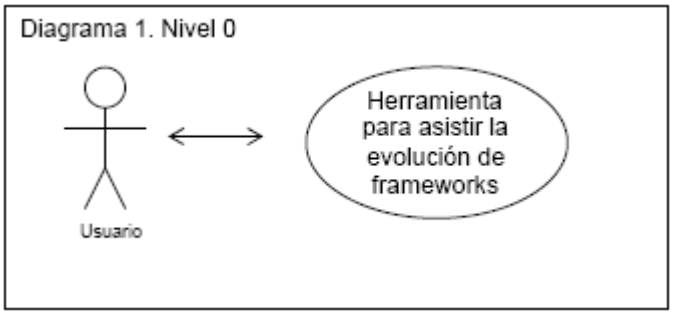


Figura 4.5. Caso de uso de alto nivel - Módulos que conforman la herramienta.

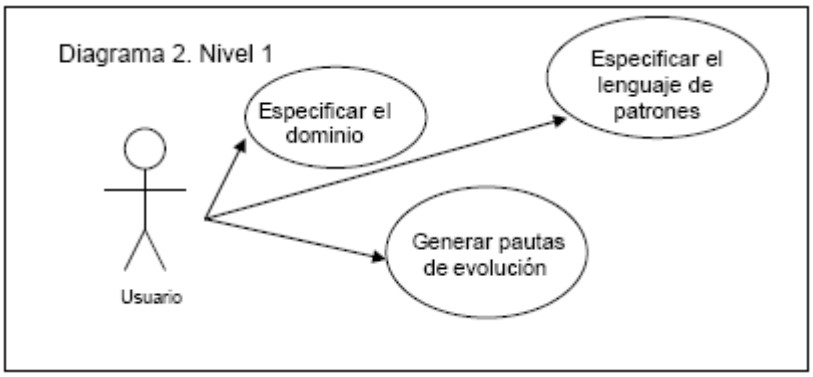
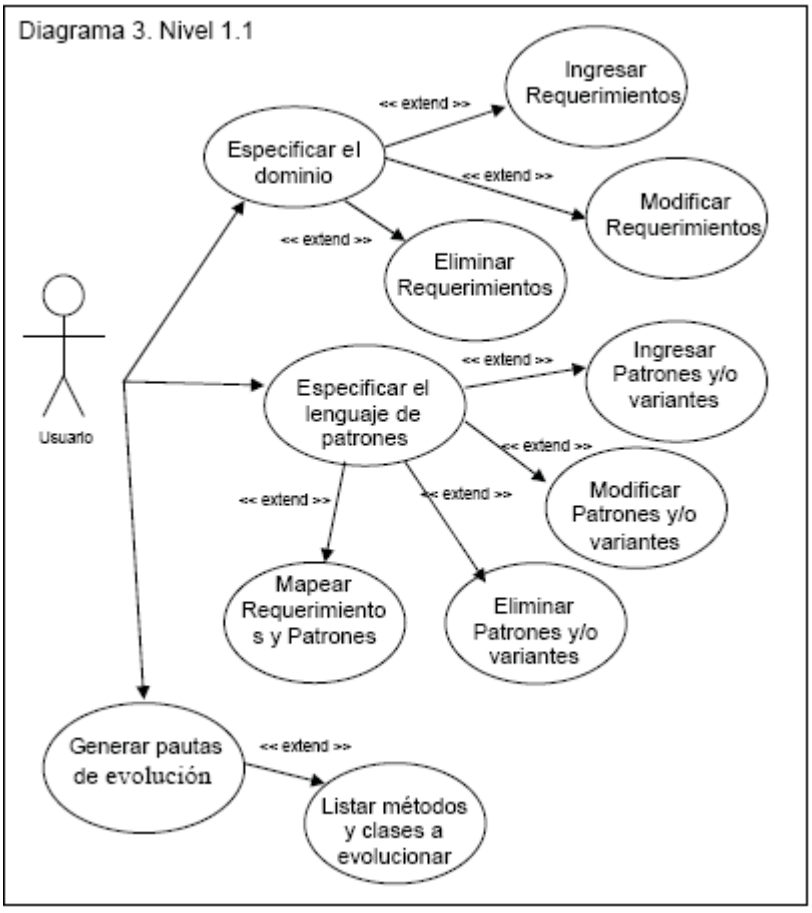


Figura 4.6. Extensión de los módulos que conforman la herramienta.



4.4.3. Descripción de casos de uso.

A continuación se describen los casos de uso presentados en la sección anterior.

Casos de uso:

Nombre:	Especificar el dominio.
Actores:	Usuario.
Función:	Permitir la especificación de requerimientos del dominio en que se enmarca el framework.
Descripción:	El usuario puede registrar los requerimientos del dominio específico, identificando así las funciones que deberán realizar las aplicaciones que pertenezcan al mismo dominio. También es posible modificar o eliminar requerimientos del dominio.

Nombre:	Especificar el lenguaje de patrones.
Actores:	Usuario.
Función:	Permitir la especificación del lenguaje de patrones del dominio especificado.
Descripción:	El usuario puede registrar los patrones y/o variantes utilizadas para solucionar problemas presentes en el dominio. También es posible modificar o eliminar patrones y/o variantes utilizadas. Información que puede ser mapeada con los requerimientos para saber qué patrones cumplen qué funciones y cómo se encuentra solución.

Nombre:	Generar pautas de evolución
Actores:	Usuario.
Función:	Permitir la generación de un listado de clases y métodos que deben ser modificados para evolucionar el framework.
Descripción:	El sistema luego de realizar el mapeo entre dominio y patrones genera un ranking con los métodos más utilizados, las clases que los proveen y las relaciones existentes entre esas clases, mostrando al usuario el listado resultante.

Nombre:	Especificar el dominio, Ingresar requerimientos.
Actores:	Usuario.
Función:	Permitir el ingreso de los requerimientos comunes de las aplicación del dominio específico.
Descripción:	El usuario ingresa directo a la base de datos los requerimientos especificando el nombre y detalle de cada función que deben realizar las aplicaciones pertenecientes al dominio.

Nombre:	Especificar el dominio, Modificar requerimientos.
Actores:	Usuario.
Función:	Permitir la modificación de los requerimientos comunes de las aplicación del dominio específico.

Descripción:	El usuario busca el requerimiento a modificar en la base de datos indicando el nombre de la función realizada por las aplicaciones pertenecientes al dominio a modificar, realiza los cambios necesarios y guarda los cambios realizados.

Nombre:	Epecificar el dominio, Eliminar requerimientos.
Actores:	Usuario.
Función:	Permitir la eliminación de los requerimientos comunes de las aplicación del dominio específico.
Descripción:	El usuario busca directo en la base de datos el(los) requerimiento(s) a eliminar especificando el nombre de la(s) función(es) que deben realizar las aplicaciones pertenecientes al dominio, la(s) elimina guardando el(los) cambio(s) realizado(s).

Nombre:	Especificar el lenguaje de patrones, Ingresar patrones y/o variantes.
Actores:	Usuario.
Función:	Permitir el ingreso de patrones y/o variantes utilizados en el framework.
Descripción:	El usuario ingresa directo a la base de datos los patrones y/o variantes utilizados en el framework especificando el nombre y detalle de cada patrón/variante.

Nombre:	Especificar el lenguaje de patrones, Modificar patrones y/o variantes.
Actores:	Usuario.
Función:	Permitir la modificación de cada patrón/variante utilizado en el framework.
Descripción:	El usuario busca el patrón/variante a modificar en la base de datos indicando el nombre del patrón/variante a modificar, realiza los cambios necesarios y guarda los cambios realizados.

Nombre:	Especificar el lenguaje de patrones, Eliminar patrones y/o variantes.
Actores:	Usuario.
Función:	Permitir la eliminación de cada patrón/variante utilizado en el framework.
Descripción:	El usuario busca directo en la base de datos el patrón/variante a eliminar especificando el nombre, lo elimina guardando los cambios realizados.

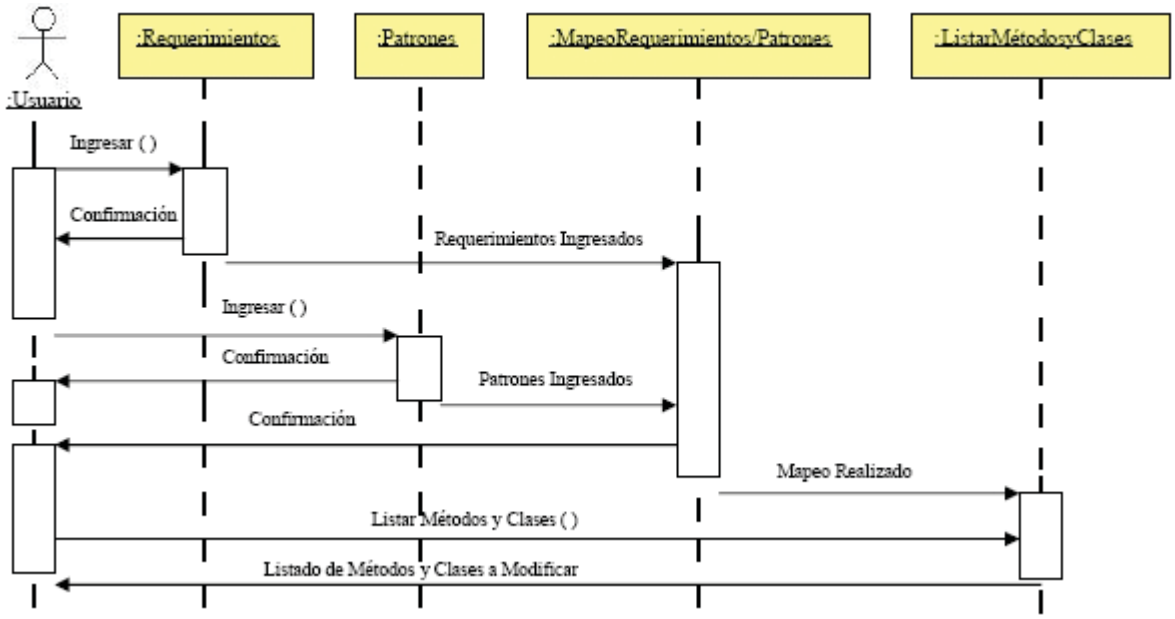
Nombre:	Especificar el lenguaje de patrones, Mapear requerimientos y patrones.
Actores:	Usuario.
Función:	Permitir contrastar la información de dominio ingresada en la base de datos con los patrones, buscando las coincidencias.
Descripción:	El usuario indica al sistema que realice la comparación, el sistema realiza una comparación registro por registro en las tablas que contienen la información del dominio y de los patrones buscando coincidencias, al encontrar una coincidencia la registra en una nueva tabla dentro de la base de datos,

	registrando toda la información que acompaña al método o función encontrada (clase que la provee, relación, cardinalidad, etc.).

Nombre:	Generar pautas de evolución, Listar métodos y clases a evolucionar.
Actores:	Usuario.
Función:	Permitir obtener un listado ordenado de los métodos y clases que podrían explicar mejor su compartimiento o que podrían cambiar su interrelación.
Descripción:	El usuario indica al sistema que realice el listado, el sistema realiza un ordenamiento de la tabla que contiene las coincidencias encontradas, haciendo un ranking de los métodos más coincidentes, las clases que podrían presentar múltiples comportamientos y sus interrelaciones, presentándole el listado al usuario en pantalla o impreso.

4.4.4. Diagramas de secuencia.

Figura 4.7. Diagrama de Secuencia de la Herramienta.



4.5. Análisis de la construcción de una herramienta genérica.

4.5.1. Consideraciones iniciales.

En esta sección se propone un proceso para el desarrollo de una herramienta genérica, que puede ser adaptada conforme los requisitos específicos de cada par "framework y lenguaje de patrones", permitiendo la evolución de ese framework basándose en su lenguaje de patrones. En esta sección no profundizaré, ya que he optado por la construcción de un prototipo de la alternativa explicada en el punto anterior.

La herramienta genérica puede ser considerada bajo dos puntos de vista: como un framework que puede ser adaptado para evolucionar frameworks específicos (en ese caso, herramientas específicas) o como un modelo de objetos adaptativos [21] (del inglés adaptive object model), es decir, como un sistema que representa clases, atributos, relaciones y comportamiento como meta datos, de forma es un modelo basado en instancias en vez de clases. Para permitir el comportamiento del sistema, basta que sean cambiados los meta datos.

Es más correcto considerar a la herramienta genérica como un framework, y documentarla de forma de facilitar su uso para la evolución de frameworks específicos. A pesar de ensamblarse en la definición de modelos de objetos adaptativos, algunas características hacen que sea difícil considerarla como tal, como por ejemplo la necesidad de escribir el código para la generación de los métodos-gancho.

El proceso propuesto para el desarrollo de una herramienta genérica tiene dos pasos: en el primero se construye la herramienta genérica siguiendo las especificaciones mostradas en el punto anterior y en el segundo es adaptada para un par específico "frameworks y lenguaje de patrones". El segundo paso puede ser repetido cuantas veces sea necesario, produciendo herramientas para evolucionar frameworks específicos. El primer paso tiene como resultado una herramienta genérica, que deberá ser adaptada, en el segundo paso, para cada caso particular (lenguaje de patrones y su framework asociado). El proceso de construcción de una herramienta genérica no será explicado en este trabajo, debido a su complejidad, solo se hará un acercamiento al paso necesario para adaptar la herramienta.

4.5.2. Adaptación de una herramienta genérica.

Esta sección se preocupa de la adaptación de la herramienta construida en la sección 4.3 de acuerdo con cada par específico "framework y lenguaje de patrones". Las condiciones mínimas necesarias para hacer tal adaptación son: existencia de un lenguaje de patrones compuesto de patrones de análisis para los cuales existe un diagrama de clases ilustrando la solución y sus variantes; un framework que implementa los patrones del lenguaje de patrones; y la documentación del framework, que mapea los patrones del lenguaje de patrones para las clases del framework.

Diversas actividades, descritas a continuación, son necesarias para adaptar la herramienta al framework específico.

Registrar información sobre el lenguaje de patrones.

Usando la estructura suministrada por la herramienta, se deben insertar todas las meta-informaciones sobre el lenguaje de patrones específica, tales como:

Nombre de todos los patrones;
Nombre de todas las variantes de cada uno de los patrones;
Clases participantes de todos las variantes de cada uno de los patrones, así como sus atributos (obligatorios y opcionales);
Relación entre los patrones (orden en el cual pueden ser aplicados y restricciones de aplicación); y

Operaciones de entrada y salida que pueden ser incluidas en los menús del sistema.

Para dar de alta esa información en la base de datos se puede utilizar el propio sistema administrador de base de datos, o un sistema que soporte esa actividad, si ha sido implementado durante la construcción de la herramienta genérica.

Registrar información sobre el mapeo entre el lenguaje de patrones y el framework específico.

Para cada patrón/variante del lenguaje de patrones, se deben mapear sus clases participantes a las clases del framework que las implementa. La principal fuente de información para la realización de esa actividad es el manual de instanciación del framework, o cookbook, producido durante la fase de documentación del framework.

Habiendo construido, usando el proceso descrito en el trabajo de proyecto 1, ese manual contiene tablas que mapean los patrones, en especial las clases participantes de sus variantes, a las clases del framework que las implementan por medio de tablas de una base de datos, conforme lo propuesto en la sección 6.4.1. Para ello, las tablas del cookbook deben ser almacenadas en las respectivas tablas de la herramienta. Si es necesario, se puede recurrir al código fuente del framework para esclarecer posibles dudas en cuanto al mapeo.

Crear los métodos del generador de pautas para la evolución para cada método-gancho del framework.

Una de las tablas de la herramienta contiene los métodos-gancho pertenecientes a las clases abstractas del framework, y que necesitan ser sobrepuestos. Para utilizar los mecanismos propuestos para la generación asistida de los métodos, se deben implementar métodos especiales en la herramienta que listen esos métodos y los incluyan en la clase apropiada del framework que está siendo evolucionado.

Para crear un método-gancho, se deben mostrar los métodos, por ejemplo por medio de una variable texto (string) que contenga los nombres de los métodos, y después usar las características de reflexión del lenguaje de programación para incluir ese método en la clase apropiada. Existen por lo menos dos formas de hacer eso: se pueden crear métodos específicos en el generador de pautas para listar cada método-gancho y después almacenar en una tabla en la base de datos de la herramienta, la lista de métodos que deben ser generados.

Implementar la generación de *scripts* para creación asistida de bases de datos.

Dependiendo del framework, puede ser necesario crear una base de datos para la persistencia de los objetos, sea ella orientada a objetos o relacional. Es deseable que esta tarea sea asistida por al herramienta, por ser una tarea trabajosa y sensible de errores. En el caso de bases de datos relacionales, se puede montar un script por medio de una variable de tipo texto (string), que contenga los comandos para crear la base de datos junto con la información ingresada por el usuario. Ese script puede ser ejecutado accionando un botón o una opción del menú.

Es necesario observar que el mapeo del lenguaje de patrones para el framework, presente en el cookbook, puede suministrar directrices sobre cuáles tablas e interfaces deben ser creadas y sobre la estructura de tales tablas (columna, índice, llaves, etc.). Así, la creación de los scripts puede ser hecha con base en tablas que reflejen el mapeo del lenguaje de patrones para las tablas utilizadas en la persistencia de los objetos.

4.6. Resumen del capítulo.

En este capítulo se presentó el analisis y diseño de una herramienta específica para un par Lenguaje de patrones/ Frameworks, que asiste el primer paso o etapa de la guía propuesta y validada en los capítulos anteriores. Se utilizaron casos de usos, la descripción de ellos y diagrama de secuencia para modelar el funcionamiento de la herramienta.

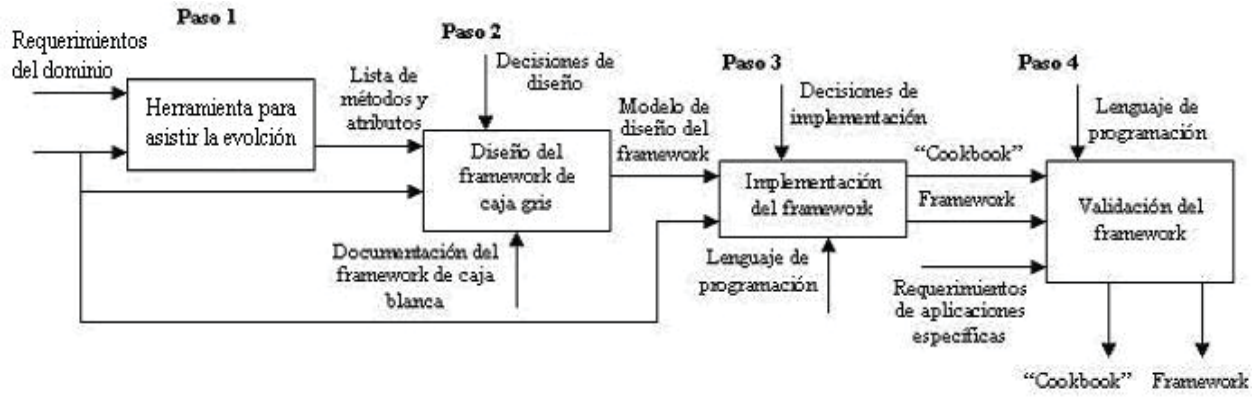
En este capítulo también se presentó el análisis de la contrucción de una herramienta genérica, es decir, que pueda ser utilizada para asistir la guía para evolucionar cualquier par Lenguaje de patrones/Frameworks. Se pudó demostrar que sólo se necesita adaptar la herramienta específica ya construída desarrollando los módulos necesarios para la especificación de lenguaje de patrones y para ingresar las clases del framework relacionado.

Capítulo 5. Validación de la Herramienta propuesta.

5.1. Consideraciones iniciales.

Para validar la herramienta, ésta se insertará dentro del procedimiento indicado por la guía, presentada y validada en este trabajo, para evolucionar el framework de caja blanca GREN inserto en el dominio de gestión de recursos de negocios (venta, arriendo y mantención de algún recurso). La herramienta asistirá el primer paso de la guía (como lo muestra la figura 14), ayudando al usuario a encontrar los métodos y clases comunes entre las distintas aplicaciones dentro del dominio que pueden ser instanciadas por el framework GREN, luego se continuará con el desarrollo de la guía para obtener un framework de caja gris, más fácil de reutilizar e instanciar. Los pasos 3 y 4 de la guía no se ejecutarán debido a la complejidad y tiempo de desarrollo que estos requieren.

Figura 5.1. Guía para la evolución asistida por una herramienta.



5.2. Aplicación de la guía para la evolución del framework de caja blanca GREN.

5.2.1. Ingreso de requerimientos del dominio a la herramienta.

La aplicación debe ocuparse del alquiler de los recursos, que pueden ser activos prestados a un cliente para cierto período o los servicios hechos por un experto durante una cierta hora.

El sistema debe permitir que se reserven previamente los recursos de modo que estén disponibles para el usuario según un horario. Por ejemplo, en un alquiler de autos generalmente se reserva un auto cuando se intenta alquilar por un período específico. Varios clientes podrían formar una lista de gente interesada en un recurso específico. Sin embargo, cuando está disponible, la reservación es innecesaria.

La aplicación se debe ocupar del comercio de los recursos, que pueden implicar vender y/o comprar recursos. El negociar recursos se puede pensar en la transferencia de la característica del recurso, en el cual un recurso poseído por una parte será transferido a otra. En una venta, si el recurso no está disponible en la acción, el cliente puede completar una orden que sea entregada cuando sea posible, si el recurso está disponible se entregará al cliente. En una compra se realiza una orden al vendedor quien entrega el recurso dentro de cierto período.

La aplicación se debe ocupar del mantenimiento o de la reparación de un recurso, que generalmente serán activos del cliente que presentan averías o necesitan mantenimiento periódico. Los cuales deben ser reparados para ser utilizados otra vez o evitar que fallen dentro de un intervalo del tiempo. Por ejemplo, los automóviles, los televisores, artículos eléctricos, etc. son recursos que tienen a menudo problemas durante su ciclo vital.

El sistema debe tratar con una o más de las siguientes transacciones: órdenes, ventas, compras, alquileres, asignaciones, reservaciones, reparaciones, o mantenciones. Estas transacciones se refieren a los recursos del negocio manejados por los sistemas específicos tales como productos en un almacén, videocintas en una tienda de alquiler, libros en una biblioteca, tiempo del médico en una clínica médica, o automóviles en un taller de reparaciones mecánico, etc.

El sistema debe considerar la forma de cuantificación del recurso. Hay ciertas aplicaciones en las cuales es importante rastrear casos específicos de un recurso, porque se tramitan individualmente. Por ejemplo, un libro en una biblioteca puede tener varias copias, cada uno prestadas a un distinto lector. Algunas aplicaciones se ocupan de cierta cantidad del recurso o de porciones del recurso. En estas aplicaciones, no es necesario saber qué caso particular del recurso fue tramitado realmente. Por ejemplo, cierto peso de acero se vende. En otras aplicaciones, el recurso se trata en su totalidad, como por ejemplo un auto que va a mantención o un doctor que examina a un paciente.

Una vez identificado y cuantificado un recurso del cual la aplicación se ocupa, el sistema debe controlar la manera en que los recursos se almacenan, para facilitar su recuperación cuando sea necesario. Esto implica muchas consideraciones tales como el volumen ocupado por un recurso, el ambiente necesario para mantenerlo en buenas condiciones y la frecuencia en la cual es utilizado.

La mayoría de las transacciones del recurso tienen un costo, pagado por uno de los participantes implicados. Por ejemplo, para alquilar una casa, para comprar un refrigerador o para reparar un artículo, los clientes tienen que pagar cierta cantidad. Si el cliente no tiene todo el dinero disponible inmediatamente, algunas aplicaciones ofrecen la opción de pagar en cuotas. Esto da lugar a una administración más compleja por el sistema, porque las cuotas entrantes y atrasadas necesitan un control exacto. Además, el sistema debe ofrecer a los clientes varias opciones de pago, de modo que se sientan libres de utilizar la más conveniente.

La aplicación debe manejar más de un recurso en una transacción. Por ejemplo, cuando un cliente va a un almacén de alquiler de videos, alquilará probablemente más de un video en la misma visita. O cuando se hace una petición de compra, varios productos distintos se solicitan generalmente al mismo tiempo. Así, se debe permitir que una sola transacción tenga muchos artículos.

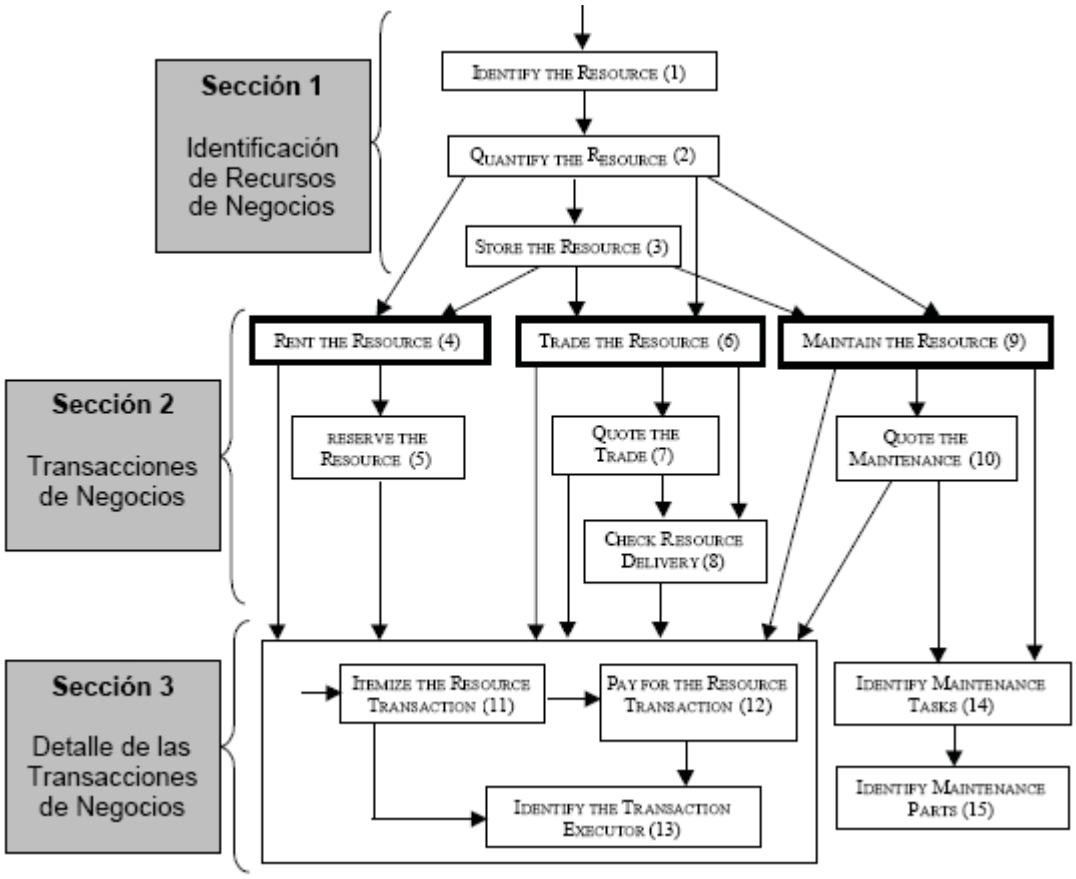
El sistema debe registrar los datos de la persona que ejecutó la transacción del recurso. Por ejemplo, en una tienda se debe registrar al vendedor que realizó la venta para administrar las comisiones.

Durante la mantención de recursos, probablemente algunas piezas del recurso necesiten ser cambiadas. En tales casos el sistema debe permitir especificar qué piezas se utilizan en la mantención.

Cuando un recurso presenta una avería y necesita ser mantenido, algunos servicios de trabajo son generalmente necesarios. Por ejemplo, un automóvil con problemas de freno podía necesitar tener un cambio de las pastillas, un ajuste del cable de freno y una lubricación del pedal. En algunos casos, cada uno de estos servicios de trabajo es hecho por una persona distinta. Por lo tanto, es importante que el sistema permita especificar las diferentes tareas realizadas durante el mantenimiento.

5.2.2. Ingreso de los patrones del lenguaje de patrones GRN a la herramienta.

Figura 5.2. Patrones del lenguaje de patrones GRN y su dependencia [3].



Identificación de Recursos (1): El patrón permite determinar cómo representar el recurso de negocio de las transacciones procesadas por el sistema. Determinando el tipo de recurso y su descripción.

Cuantificación de Recursos (2): El patrón permite determinar cómo cuantificar el recurso. Existen cuatro sub-patrones para cuantificar: RECURSO SIMPLE, RECURSO MENSURABLE, RECURSO INSTANCIABLE y RECURSO EN LOTES.

Almacenamiento de Recursos (3): El patrón permite determinar la forma en que el recurso es almacenado. El patrón ayuda a determinar el departamento de la organización que tienen recursos a almacenar, la localización para almacenar el recurso y la condición de almacenaje.

Arrendar el Recurso (4): El patrón permite determinar cómo la aplicación manejará los recursos que son arrendados a los clientes, además de ayudar a registrar todos los elementos

que son necesarios en una transacción de arriendo, por ejemplo los datos del cliente, la fecha de devolución, etc.

Reservar el Recurso (5): El patrón permite determinar cómo manejar la reservación del recurso previo a su arriendo real. Es indispensable mantener la información actualizada de las reservas y junto con ella la información de stock de recursos.

Comercializar el Recurso (6): El patrón permite determinar cómo administrar los recursos que serán comercializados, entiéndase comercializar como una transacción de compra o venta de los recursos.

Cotizar la Comercialización (7): El patrón permite administrar las cotizaciones que se realizan antes de realizar una transacción de compra o venta.

Comprobar la entrega del Recurso (8): El patrón permite realizar las acciones necesarias para comprobar que el producto comprado sea el correcto o que el producto vendido haya sido entregado correctamente al cliente y no descontar el stock antes de retirarlo de bodega.

Mantener el Recurso (9): El patrón permite administrar la mantención de recursos. Se deben registrar datos de esta transacción ya que son importantes tanto para el cliente como para la organización, por ejemplo la fecha de última mantención y los materiales utilizados en ella.

Cotizar la Mantención (10): El patrón permite manejar las cotizaciones de mantención realizadas por los clientes en forma previa a realizar una transacción de mantención.

Detallar la transacción del Recurso (11): Este patrón permite administrar las transacciones en la que intervienen más de un recurso. Por ejemplo es posible que en un cliente compre más de un recurso en una misma transacción de venta. Es necesario registrar la información de cada recurso que ha sido transado.

Pagar por la transacción del Recurso (12): Este patrón permite manejar las distintas alternativas de pago que son ofrecidas a los clientes ante las distintas transacciones de recursos.

Identificar al Ejecutor de la Transacción (13): Este patrón permite registrar al ejecutor de una transacción, ya que diferentes aplicaciones podrían necesitar registrar datos además del nombre del ejecutor, por ejemplo, porcentaje de comisión, meta de venta, etc.

Identificar las Tareas de Mantención (14): Este patrón permite determinar cómo poder identificar las tareas involucradas en una transacción de mantención o en una cotización de mantención.

Identificar Partes para la Mantención (15): Este patrón permite determinar cómo identificar las partes y piezas utilizadas en una transacción de mantención o en una cotización de mantención.

5.2.3. Mapeo entre los requerimientos y los patrones.

Tabla 7. Mapeo de los Requerimientos y los patrones del dominio.

Patrones / Requerimientos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Arrendar recursos				X											
Reservar previamente los recursos					X										
Vender y/o compra recursos						X									
Mantener recursos									X						
Distintos tipos de transacciones	X														
Cuantificar los recursos		X													
Distintas formas de pago												X			
Transar varios recursos											X				
Registrar ejecutor de la transacción													X		
Especificar piezas de mantención															X
Detallar tareas de mantención														X	

5.2.4. Lista de métodos y clases del framework de caja blanca a modificar.

Tabla 8. Lista de métodos y clases.

Métodos
IsAvailable( )
ListByIdCode( )
RegisterPayment( )
CalculateFreeSpace( )

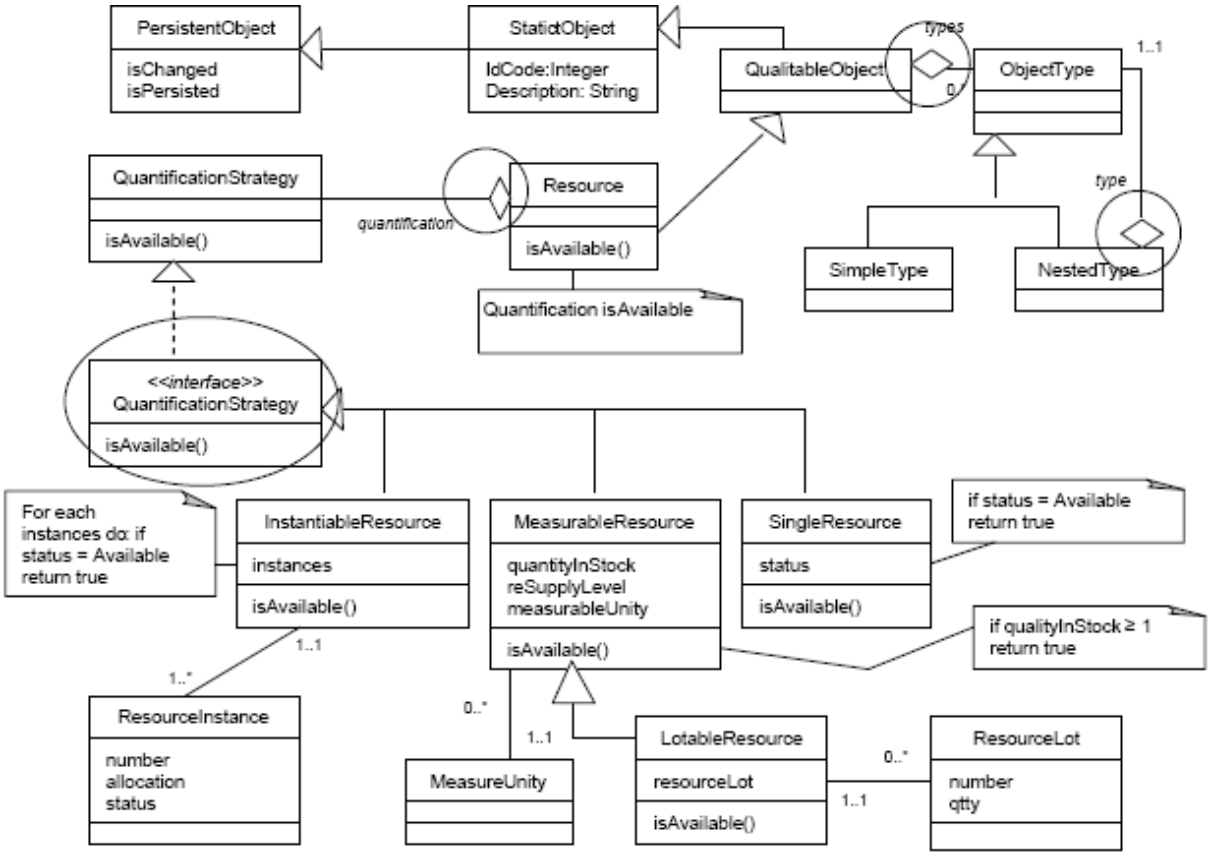
Clases
Resource
QualifiableObject
Store

Para los métodos descritos se debe crear la interfaz que explique su comportamiento y para las clases se deben crear clases abstractas que encapsulen los múltiples comportamientos de cada una, cambiando la relación de herencia por composición.

5.2.5. Diseño del framework de caja gris.

Se realizan los cambios al modelo de clases del framework de caja blanca, agregando interfaces para explicar el comportamiento de los métodos descritos en el listado anterior y se reemplazan las relaciones de herencia de las clases encontradas por relaciones de composición o agregación.

Figura 5.3. Parte del diagrama de clases del framework de caja gris GREN.



5.3. Consideraciones finales.

Como se mencionó anteriormente, no se realizarán los últimos dos puntos de la guía, debido a su complejidad, ya que implicarían implementar un framework en forma completa y luego utilizarlo para instanciar aplicaciones específicas del dominio al cual apunta, trabajo que puede llevar meses cuando es realizado por un equipo de desarrolladores. A pesar de no concluir los pasos de la guía se ha demostrado mediante las métricas de ingeniería de software aplicadas a los diagramas de ambos frameworks (caja blanca y caja gris) que al aplicar la guía se permite evolucionar un framework de caja blanca a caja gris y con esto, se consigue una mayor reusabilidad, modularidad y encapsulamiento, y se disminuye el acoplamiento y la complejidad.

5.4. Resumen del capítulo.

En este capítulo se presentó un ejemplo de utilización de la guía propuesta, asistida por la herramienta construída. En el ejemplo se utilizó el par lenguaje de patrones GRN (Gestión de

Recursos de Negocios) y el framework de caja blanca GREN (basádo en el lenguaje de patrones GRN).

Con la utilización de la guía apoyada por la herramienta construída se puede apreciar el aporte real hecho a la reutilización de componentes al obtener un framework de caja gris, mucho más fácil de instanciar para obtener aplicaciones específicas, reduciendo el tiempo de desarrollo y los costos asociados.

Conclusiones.

De acuerdo a los objetivos planteados para este proyecto, teniendo en cuenta los objetivos generales: “Proponer una guía general, basada en lenguaje de patrones, para la evolución de los framework de caja blanca hacia frameworks de caja gris, la cual deberá estar lo suficientemente detallada, validar la guía propuesta mediante métricas utilizadas en ingeniería de software y desarrollar una herramienta que asista el proceso de evolución de frameworks basada en lenguajes de patrones para un dominio específico. A partir de estos objetivos se definieron una serie de objetivos específicos que en conjunto permitieron alcanzar los objetivos planteados.

Los objetivos generales se fueron cumpliendo gracias al levantamiento de información realizado en donde se recopilaron los datos necesarios para entender los conceptos de frameworks y lenguajes de patrones que proporcionan el marco general del proyecto, luego de internalizar los conceptos y planteada la problemática a resolver se identificaron y analizaron las características de la construcción e instanciación de frameworks de caja blanca basada en lenguaje de patrones, con lo que se logro entender los procesos que se llevan a cabo para desarrollar aplicaciones específicas a partir de un framework desarrollado en base a un lenguaje de patrones. Además, se identificaron y analizaron las diferencias de la construcción e instanciación de frameworks de caja blanca con la construcción e instanciación de frameworks de caja gris basándose en lenguaje de patrones, lo que generó la creación de una guía que apoyándose en los lenguajes de patrones ayuda a evolucionar frameworks de caja blanca hacia frameworks de caja gris, la cual se basa en la modificación del diagrama de clases del frameworks determinando los métodos, clases y funciones más utilizados para crear las interfaces que explican su comportamiento y se crean las clases abstractas que encapsulan los anteriores múltiples comportamientos.

Se aplicaron cinco de las siete métricas clásicas de ingeniería de software de Chidamber y Kemerer, las cuales se utilizaron para medir el niveles de reusabilidad de frameworks evolucionados, utilizando la guía propuesta para validar su mayor aporte a la reutilización, con el fin de validar la propuesta realizada, de ésta forma se puede concluir que la guía es un

real aporte a la reutilización de componentes de software, ya que inserta a los frameworks en un ciclo de evolución necesario para que se mantengan acorde a los requerimientos del dominio específico al que apuntan. Además de la aplicación de las métricas para validar la guía propuesta, se presenta un ejemplo de aplicación de la guía con un par lenguaje de patrones/frameworks real, es importante destacar que el ejemplo no abarcó todos los pasos de la guía ya que no se prentendía desarrollar el framework de caja gris, por lo tanto, sólo se llegó a su diseño.

También dentro del trabajo, se identificaron y analizaron las características de la construcción de un protótipo de una herramienta que asiste al proceso de evolución de frameworks de caja blanca hacia frameworks de caja gris basados en lenguajes de patrones, apoyando el primer paso de la guía (identificación de puntos comunes), logrando capturar de manera asistida los puntos más utilizados dentro de un par framework / lenguaje de patrón, con lo que se consiguió entender los procesos que se llevan a cabo para evolucionar frameworks desarrollados en base a un lenguaje de patrones, pero ahora de forma asistida, resaltando el apoyo que brinda la guía validada. Paso fundamental para poder desarrollar un protótipo de una herramienta acorde al problema planteado en el objetivo general.

Se construyó un protótipo de una herramienta que asista la evolución de frameworks, con el fin de ayudar a la aplicación de la guía y con ello a la reutilización de frameworks. El protótipo implementado asiste el primer punto de la guía planteada, el cual consiste en encontrar los puntos comunes entre las aplicaciones del dominio y mapear esos puntos con los patrones del framework de caja blanca para determinar que métodos y clases pueden ser modificados para así poder evolucionar el framework a caja gris.

Finalmente es posible concluir que gracias a la planificación de las actividades necesarias para cumplir con los objetivos es que se puede definir como concluido el proyecto con éxito, logrando un aporte significativo a la reutilización de componentes de software y con ello a la ingeniería de software.

Referencias.

[1] **Basset P.G.** "*Frame-Based Software Engineering*". IEEE Software, Vol. 4, nº 4, Julio 1987.

[2] **Jacobson I., Griss M. y Jonsson P.**, "*Software reuse. Architecture, Process and Organization for Business Success*", ACM Press. Addison Wesley Longman, 1997.

[3] **Braga R. T. V**., "*Um Processo para Construção e Instanciação de Frameworks baseados em uma Linguagem de Padroes para um Domínio Específico*". Instituto de Ciências Matemáticas e de Computação, 2003.

[4] **Fayad M. E y Schmidt D. C.** "*Object-oriented application frameworks*". Communications of the ACM, 1997.

[5] **Alexander C.**, "*The Timeless Way of Building*", Oxford University Press, 1979.

[6] **Coplien J. O.**, "*Software design patterns: Common questions and answers in L. Rising - The Patterns Handbook: Techniques, Strategies, and Applications*", Cambridge University Press, 1998.

[7] **Wirfs-Brock R. J. y Johnson R. E,** "*Surveying current research in Object-Oriented design*". CACM, 1990.

[8] **Johnson R. E. y Foote B.**, "*Designing reusable classes. Journal of Object-Oriented Programming*", 1998.

[9] **Fuentes L., Troya J.M. y Vallecillo A.** *"Desarrollo de Software Basado en Componentes",* Departamento de Lenguajes y Ciencias de la Computación. Universidad de Málaga. ETSI Informática. Málaga, España.

[10] **Yassin A y Fayad M. E.,** "*Application frameworks: A survey*", John Wiley & Sons, 2000.

[11] **Roberts D. y Johnson R., "***Evolving frameworks: A pattern language for developing object-oriented frameworks*", Addison-Wesley, 1998.

[12] **Beck K. y Cunningham W.,** "*Using pattern languages for object-oriented programs*". Reporte técnico no. CR-87-43, disponible en URL: http://c2.com/doc/oopsla87.html, 1987.

[13] **Gamma E., Helm R., Johnson R. y Vlissides J.**, "*Design Patterns*". Addison-Wesley, 1995.

[14] **Bosch J, Molin P., Mattsson M.; Bengtsson P. y Fayad M.,** "*Framework problem and experiences in M. Fayad, R. Johnson, D. Schmidt. Building Application Frameworks: Object-Oriented Foundations of Framework Design*", John Willey and Sons, 1999.

[15] **Brugali D. y Menga G.,** "*Frameworks and pattern languages: an intriguing relationship". ACM Computing Surveys*, 1999.

[16] **Ré R., Braga R. T. V. y Masiero P. C.,** "*A pattern language for online auctions. In: 8th Pattern Languages of Programs Conference (PLoP'2001)*", Monticello - IL, USA, 2001.

[17] **Fowler M.,** "*Analysis patterns*", Addison-Wesley, 1997.

[18] **Olsina L. A., Bertoa M. F., Lafuente G. J., Martín M. A., Katrib M. y Vallecillo A.**, *"Un Marco Conceptual para la Definición y Explotación de Métricas de Calidad"*, Universidad Nacional de La Pampa, Argentina. Universidad de Málaga, España y Universidad de la Habana, Cuba.

[19] **Calero C.,** "*MÉTRICAS DEL SOFTWARE: Conceptos básicos, definición y formalización"*, Departamento de Informática Universidad de Castilla-La Mancha, 2005.

[20] **Weyuker E.,** *"Evaluating software complexity measures"*, IEEE Transactions on Software Engineering, 1988.

[21] **Yoder J. W, Balaguer F., Johnson R. E,** *"Architecture and design of adaptative object models"*, Conference on Object-Oriented Programming System, Languages, and Applications (OOPSLA '01), ACM SIGPLAN.

[22] **CINCOM Visualworks 5i.4** no-comercial. Disponible para instalación URL: http://www.cincom.com, 2001.

[23] **MySQL versión 3.23**. Disponible para instalación en URL: http://www.mysql.com, 200

Glosario de Términos.

- **Caja blanca:** Sistema en el cual se conoce como se transforman las entradas en las salidas.

- **Caja gris:** Sistema que posee procesos caja blanca y procesos caja negra.

- **Caja negra:** Sistema en el cual sólo se conocen las interfaces para las entradas y las salidas obtenidas.

- **Complejidad:** Recursos requeridos durante el cálculo para resolver un problema.

- **Componente Software:** Unidades binarias desarrolladas, adquiridas e incorporadas al sistema de forma independiente, y que interactúan para formar un sistema funcional.

- **Encapsulación:** Es el criterio que nos lleva a agrupar en una clase todas los campos que determinan el estado de un objeto y los métodos que permiten acceder a ellos.

- **Herencia:** Característica que permite que una clase posea las características de otra, sin tener que reescribir el código.

- **Instanciación:** Proceso de creación de un objeto a partir de una clase.

- **Interfaz:** Define un tipo de datos, pero sólo indica el prototipo de sus métodos, nunca la implementación.

- **Jerarquía de herencia:** Árbol construido mediante las relaciones de herencia en las clases.

- **Scripts:** Conjunto de comandos escritos en un lenguaje interpretado para automatizar ciertas tareas de aplicación.

- **Sobrescritura:** Poseer el mismo método, pero con código distinto, en una clase base y en una clase que deriva de ella.

Listado de Acrónimos.

A.K.A = Also Know As.

AOC = Acoplamiento entre Objetos de las Clases.

AWT = Abstract Windows Toolkit.

DSOM = Distributed System Object Model.

FCM = Falta de Cohesión de los Métodos.

GoF = Gang of Four.

GREN = Framework de caja blanca basado en el lenguaje de patrones GRN.

GRN = Gestión de Recursos de Negocios.

GUI = Graphics User Interface.

JFC = Java Foundation Classes.

MPC = Métodos Ponderados por Clase.

NHC = Número de hijos por clase.

OLE = Object Linking and Embedding.

PAH = Profundidad del árbol de herencias.

RC = Respuesta a una Clase.

SRA = Sistema de Reparación de Agujeros.

TC = Tabla de Clases.

TH = Tabla Historial.

UML = Unified Modeling Language.

VLSI = Very Large Scale Integration.

XML = Extensible Markup Language.

Apéndice A. Ejemplo: construcción de un lenguaje de patrones para gestión de recursos de negocios (GRN).

Se utiliza para la construcción del lenguaje de patrones el proceso presentado en el punto 1.5.2 para el dominio de aplicación de gestión de recursos de negocios, dominio particular de sistemas de información. Como resultado obtendremos el lenguaje de patrones GRN, él cual será utilizado para la construcción de un framework para desarrollar aplicaciones en el dominio mencionado.

1. Análisis del Dominio.

Las similitudes en las aplicaciones que necesitan registrar transacciones de alquiler, comercialización y mantención de recursos. En este trabajo el concepto de "transacción" es considerado como un evento significativo que necesita ser almacenado por el sistema a través del tiempo.

El alquiler de recursos se enfoca principalmente en la utilización temporal de un bien o servicio, como por ejemplo una cinta de vídeo o el tiempo de un especialista. La comercialización de recursos se enfoca en la transferencia de propiedad de un bien, como por ejemplo una venta o subasta de productos. El mantenimiento de recursos se enfoca en la reparación o conservación de un determinado producto, utilizando mano de obra y piezas para ejecución, como es el ejemplo de un taller de reparación de electrodomésticos.

Mediante ingeniería inversa, analizando aplicaciones existentes en el dominio, se han podido encontrar tres grupos definidos de acuerdo a su propósito, como se indicó en la figura 5.2: En el grupo 1 son incluidas las actividades de identificación y posible calificación, cuantificación y almacenamiento de los recursos de negocios, en el grupo 2 las actividades que tratan con las transacciones efectuadas por el sistema y en el grupo 3 las actividades que cuidan de detalles asociados a la mayoría de las transacciones de negocio.

2. Determinación de Patrones y grafo de flujo de aplicación de Patrones.

Al analizar el modelo de análisis de dominio se han podido identificar quince patrones de análisis, algunos de los cuales son aplicaciones o extensiones de patrones recurrentes propuestos en la literatura existente, junto con las dependencias entre los patrones encontrados y el orden en el cual ellos son generalmente aplicados.

Las dependencias son mostradas y complementadas en una sección incluida en cada patrón llamada "Próximos patrones". Los principales patrones son indicados en la figura mediante un cuadro destacado. Ellos son: ALQUILAR RECURSOS, COMERCIALIZAR RECURSOS y MANTENER RECURSOS. El uso de estos patrones no es mutuamente excluyente. En la realidad, existen aplicaciones en las cuales ellos pueden ser usados conjuntamente, como por ejemplo en un "RENT A CAR" que también compra, vende y repara sus propios automóviles.

El primer patrón en ser aplicado es IDENTIFICAR EL RECURSO. Los patrones CATEGORIZAR LA TRANSACCIÓN DEL RECURSO, PAGAR POR LA TRANSACCIÓN DEL RECURSO e IDENTIFICAR EL EJECUTOR DE LA TRANSACCIÓN son mostrados dentro de una caja, denotando que son aplicables a todas las situaciones en las cuales una flecha llega hasta el borde de esa caja. La flecha sin origen que llega al patrón 11 significa que ese es el primer patrón que debe ser verificado, seguido de los patrones 12 y 13.

3. Escritura de Patrones.

La notación utilizada para expresar los patrones es UML (del inglés Unified Modelling Language). Para cada patrón es incluido un ejemplo de su instanciación, en la cual nuevos atributos pueden ser añadidos de acuerdo con aplicaciones específicas. Métodos básicos para crear un objeto, atribuir y recuperar valores de atributos, añadir y remover conexiones de objetos, recoger un objeto específico y excluir objetos no son mostrados en los modelos de clases, ya que sumarían más complejidad a esos diagramas sin traer ganancias efectivas al modelo. En vez de eso, se asume que esos métodos estén presentes en cada clase, siempre que sea apropiado. En cuanto al formato del texto del patrón, el NOMBRE DEL PATRÓN y de otros patrones mencionados en el texto es escrito utilizando capitalización de mayúsculas; cada Elemento del patrón es subrayado; nombres de métodos, clases y atributos son escritos en fuente Courier.

Para mayor claridad se muestra la escritura de uno de los quince patrones encontrados:

Patrón 2: CUANTIFICAR EL RECURSO.

Contexto

Usted identificó el recurso controlado por su aplicación y sus cualidades relevantes. Una pregunta importante a ser considerada ahora es la forma de cuantificación del recurso. Existen ciertas aplicaciones en las cuales es necesario tener control sobre instancias específicas del recurso, porque ellas son negociadas individualmente. Por ejemplo, en una biblioteca un libro puede poseer diversas copias, cada cual prestada a un lector diferente. Otras aplicaciones lidian con una cierta cantidad del recurso o con lotes de recursos. En tales aplicaciones, no es necesario saber que instancia particular del recurso fue negociada. Por ejemplo, una cierta cantidad de cobre fue vendida. En otras aplicaciones, el recurso es único, como por ejemplo un automóvil que sufre mantenimiento o un médico que examina un paciente.

Problema

¿Cómo la aplicación cuantifica el recurso de negocio?

Influencias

Es muy importante saber, ya en la fase de análisis del sistema, exactamente cual es la forma de cuantificación del recurso adoptada. Una decisión errada en ese punto puede comprometer la evolución futura.

Si es necesario controlar instancias específicas del recurso, entonces las informaciones redundantes pueden ser almacenadas para las diversas instancias del mismo recurso. Sin embargo, esa redundancia es indeseable.

Para evitar redundancia, una nueva clase puede ser creada, en la cual las informaciones comunes a todas las instancias del mismo recurso pueden ser almacenadas sólo una vez. Pero un precio tiene que ser pagado por lidiar con dos clases en vez de una: por ejemplo, el tiempo de procesamiento será probablemente mayor.

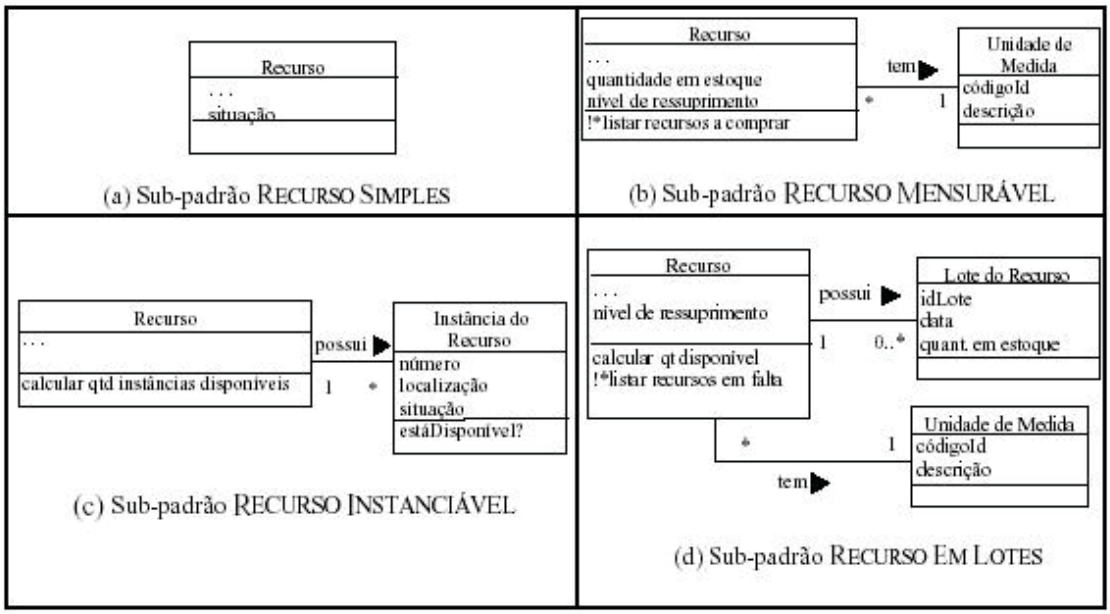
Estructura

Existen cuatro soluciones ligeramente diferentes para ese problema, dependiendo de la forma de cuantificación. La Figura muestra los cuatro sub-patrones del patrón CUANTIFICAR EL RECURSO.

Cuando el recurso es único, utilice el sub-patrón RECURSO SIMPLE (Figura A.1(a)). Cuando el recurso es tratado en cantidades específicas, utilice el sub-patrón RECURSO

MENSURABLE (Figura A.1(b)). Cuando es importante distinguir entre instancias de recursos, utilice el sub-patrón RECURSO INSTANCIABLE (Figura A.1(c)). Cuando el recurso es tratado en lotes, utilice el sub-patrón RECURSO EN LOTES (Figura A.1(d)).

Figura A.1. Patrón 2 - Cuantificar el Recurso.



Participantes

Recurso: cuando el sub-patrón RECURSO SIMPLE es utilizado, el atributo situación controla el ciclo de vida del recurso. En un taller de vehículos, la situación del vehículo podría ser: "pronto", "aguardando reparación" o "en reparación". Otro ejemplo es la inclusión de los atributos cantidad en stock y nivel de reabastecimiento para lidiar con el control de stock cuando se utiliza el sub-patrón RECURSO MENSURABLE. En ese caso, el atributo situación no se aplica, pues el sistema trabaja con cantidades grandes del recurso y, por lo tanto, no puede controlar el ciclo vida del recurso individualmente.

Instancia del Recurso: representa cada ejemplar o copia de un recurso de negocio. El atributo situación controla el ciclo de vida de cada recurso individualmente; por ejemplo, durante el ciclo de vida de una copia de un libro hay cuatro tipos diferentes de situación:"disponible", "sólo reservado", "sólo prestado" y "reservado y prestado".

Lote de Recurso: representa cada lote individual de recursos de negocios, en general compuesto de una cantidad específica de recursos. En algunos casos es importante

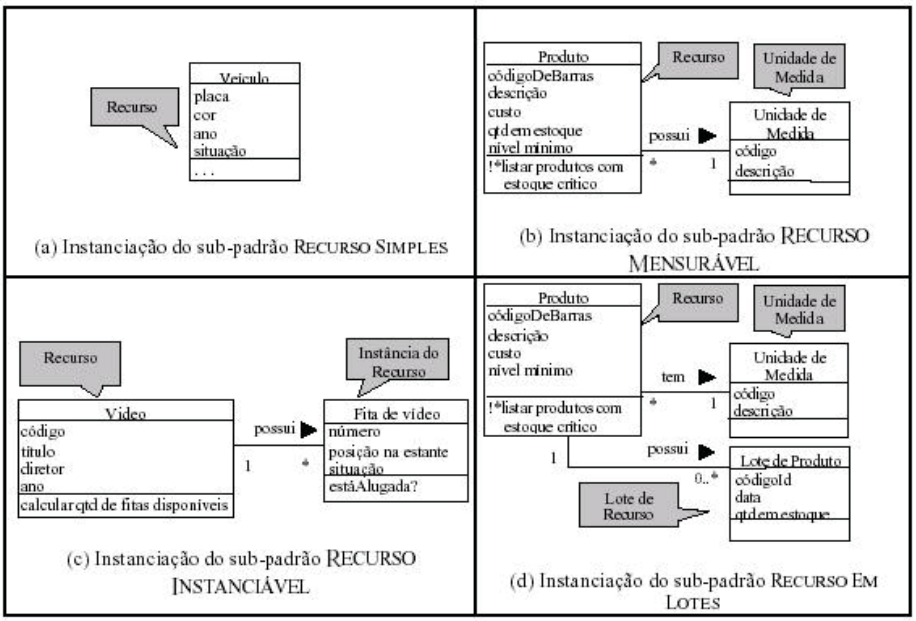
añadir un atributo fecha de vencimiento para controlar los recursos a que se han utilizado de entrada o que han sido descartados por vencimiento.

Unidad de Medida: representa todas las posibles unidades de medida por las cuales los recursos de negocio pueden ser medidos, como por ejemplo gramos, kilogramos, paquetes, etc.

Ejemplo

La figura A.2. muestra instanciones de cuatro sub-patrones del patrón CUANTIFICAR EL RECURSO.

Figura A.2. Ejemplos de uso de Patrón 2.



Próximos patrones

Después de CUANTIFICAR El RECURSO (2), examine su aplicación para verificar si es importante mantener informaciones sobre el almacenamiento de los recursos. En caso positivo, aplique ALMACENAR EL RECURSO (3). En caso negativo, continúe examinando su aplicación para verificar que tipos de transacción son efectuadas. Si la aplicación estuviera relacionada con la locación o alquiler de recursos, usted debe ALQUILAR EL RECURSO (4). Si la aplicación referirse al comercio de recursos por ejemplo compraventa o venta, usted debe COMERCIALIZAR EL RECURSO (6). Si la aplicación trata con reparación de recursos, usted debe MANTENER El RECURSO (9). Observe, sin embargo, que existen

aplicaciones en las cuales varios de esos patrones pueden ser aplicados. Por ejemplo, en un sistema de alquiler de vehículos, además de la reserva y alquiler de los vehículos, se puede efectuar el control de adquisición, mantenimiento y venta de los vehículos. Otros tipos de transacciones, no previstas por el lenguaje de patrones GRN, pueden ser efectuadas con recursos de negocios. Por ejemplo, la subasta de recursos es una transacción en la cual una administradora de subastas coloca a la venta productos pertenecientes a un propietario, que pueden ser adquiridos por un comprador. Se puede crear una extensión al lenguaje GRN para incorporar la subasta de recursos.

4. Validación del Lenguaje de Patrones.

El lenguaje de patrones para Gestión de Recursos de Negocio (GRN) fue el resultado de más diez años de práctica por parte de la doctora Braga [3] en el desarrollo de sistemas para pequeñas y medianas empresas en el dominio de gestión de recursos de negocios. Las similitudes existentes entre estos sistemas motivo la creación de este lenguaje de patrones el cual ha sido utilizado por muchos analistas y desarrolladores de Brasil para desarrollar aplicaciones en este dominio.

Apéndice B. Ejemplo: construcción de un frameworks para gestión de recursos de negocios (GREN) basado en el lenguaje de patrones GRN.

Se utiliza para la construcción del framework basado en el lenguaje de patrones GRN, el proceso presentado en el punto 1.5.4. Como resultado obtendremos el framework GREN, él cual será del tipo de caja blanca, ya que su reutilización se implementará mediante de herencia. El presente framework permitirá la creación de aplicaciones en el dominio de gestión de recursos de negocio.

1. Identificación de puntos variables.

El primer paso del proceso es identificar los puntos variables, para ello, la principal fuente es el lenguaje de patrones GRN. en la tabla 2 se muestran algunos de los cuarenta puntos variables identificados inicialmente. De ellos, treinta y seis fueron encontrados por medio de GRN. Los demás fueron encontrados al refinar y hacer una referencia cruzada de la lista de

puntos variables. Por ejemplo el segundo punto variable fue derivado del patrón 2 - CUANTIFICAR EL RECURSO (ver apéndice A), por el análisis de sus participantes, estructura y sub-patrones. Ese punto variable permite cuatro tipos diferentes de cuantificación para el recurso, de acuerdo con los cuatro sub-patrones presentados en la estructura del patrón: RECURSO SIMPLE, RECURSO MENSURABLE, RECURSO INSTANCIABLE y RECURSO EN LOTES. Es un punto variable del tipo PARTICIPACIÓN OPCIONAL, porque el usuario del framework deberá escoger cual de las clases participantes utilizará, de acuerdo con la estrategia de cuantificación deseada para el sistema objetivo.

Tabla 9. Lista parcial de puntos de variables del framework GREN.

N° P.V	**Nombre del Patrón**	**Descripción**	**Tipo**	**Fuente en GRN**	**N° Patrón**
1	Cuantificación de recursos	Un recurso puede o no tener un tipo relacionado. Puede que también tuviera múltiples tipos o tipos anidados.	PARTC-ELECCIÓN	Participantes, Variantes	1
2	Cuantificación de recursos	Un recurso puede ser único, puede tener múltiples instancias, puede ser controlado en cantidades o en lotes.	PARTC-ELECCIÓN	Participantes, Estructuras, Variantes (sub-patrones)	2
3	Mantención del recurso	Puede ser o no deseable que la aplicación controle el almacenamiento del	PATRÓN OPCIONAL	Grafo recurso de lenguaje + contexto	3

2. Diseño del framework.

El segundo paso fue proyectar el framework GREN, comenzando por el diseño de su arquitectura, la cual fue definida en tres capas: la capa de persistencia, la capa de negocios y la capa de interfaz gráfica con el usuario (GUI). La primera versión de GREN no incluye aspectos de seguridad, control de acceso, distribución e interfaz de Web, esos aspectos podrán ser incorporados en sus futuras versiones.

La capa de persistencia posee clases para cuidar de la conexión con la base de datos, administración de los identificadores de los objetos y persistencia de los objetos. La capa de negocios se comunica con la capa de persistencia siempre que necesita almacenar permanentemente un objeto. Dentro de la capa de negocios existen diversas clases derivadas directamente de los patrones de análisis que componen el lenguaje de patrones GRN, o sea, las clases y relaciones contenidas en cada patrón poseen la implementación correspondiente en esta capa. La capa de interfaz gráfica de usuario contiene las clases responsables de la entrada y salida de datos, como formularios de interfaz, ventanas y menús, que permiten la interacción del usuario final con el sistema. Se comunica con la capa de negocios para obtención de objetos que sean mostrados en la interfaz de usuario y para devolver informaciones que sean procesadas por los métodos de la capa de negocios. Existe una herramienta para la instanciación automatizada, llamada GREN-Wizard, presentada en detalle en el apéndice C, que se sitúa por encima de la capa de interfaz gráfica de usuario, ya que utiliza todas las demás capas.

Aplicaciones específicas pueden ser construidas a partir de la capa de interfaz gráfica de usuario, usando (por medio de herencia o referencia a objetos) clases de todas las capas de GREN, o pueden ser construidas con base en la capa de negocios, si la capa de interfaz gráfica de usuario no es reutilizada a partir de GREN, pero implementada separadamente. Pueden también ser implementadas por medio de GREN-Wizard, que se encarga de hacer la comunicación con las demás capas de GREN.

Definida la arquitectura de GREN, fue entonces proyectada su jerarquía de clases, comenzando por la construcción de un modelo de clases general, englobando las clases de todos los quince patrones de GRN, y después haciendo uso de los mecanismos de especialización y generalización para refinar ese modelo. A continuación fueron utilizados patrones de diseño, como por ejemplo el *Strategy* [13], para conseguir la flexibilidad necesaria para atender a los puntos variables del framework.

La Figura 5.3 muestra parte del diagrama de clases de proyecto de GREN. En ese ejemplo en particular, son mostradas las clases relacionadas a la clase *Resource*. La clase *PersistentObject* fue creada para modelar la capa de persistencia, conforme el patrón *PersistenceLayer*. Las clases *StaticObject* y *QualifiableObject* fueron creadas durante la etapa de generalización/especialización, en vista que diversas clases pueden heredar su comportamiento. El patrón de diseño STRATEGY fue utilizado para modelar el segundo punto variable de GREN. Cada objeto de la clase *Resource* se refiere a un objeto de la clase, la cual es responsable por sus aspectos de cuantificación, permitiendo que el framework implemente las cuatro diferentes soluciones requeridas.

3. Implementación del framework.

El tercer paso es la implementación de las clases diseñadas en el punto anterior, en este ejemplo se utilizó el lenguaje Smaltalk, más específicamente la versión no-comercial 5i.4 [22]. La base de datos (relacional) utilizada en la persistencia de los objetos es la MySQL [23]. La primera versión de GREN posee aproximadamente 160 clases y 30 mil líneas de código. La jerarquía de clases de GREN, así como algunos diagramas de clases, es presentada en el apéndice D.

El lenguaje de patrones GRN fue utilizada para implementar GREN de forma gradual, siguiendo secuencialmente sus quince patrones. La implementación del primer patrón, IDENTIFICAR EL RECURSO, dio origen a un pequeño framework que, al ser instanciado, originaba aplicaciones para mantenimiento de registro de recursos, como por ejemplo, un sistema para control de discos compactos (CDs) de un individuo. Algunas operaciones posibles en ese sistema son: dar de alta los CDs, imprimir listados de CDs en orden alfabético, eliminar CDs, etc. A continuación fue implementado el segundo patrón, CUANTIFICAR EL RECURSO, que extendió el framework para tratar con las posibles formas de cuantificar el recurso, como las ya mencionadas anteriormente en esta sección. Así, en el caso del sistema de control de CDs, sería ahora posible dar de alta varias copias de un mismo CD. Usando ese mismo raciocinio, cada uno de los demás patrones de GRN fue implementado, culminando con la versión final del framework GREN.

Para completar el paso de implementación de GREN, fue entonces elaborada su documentación según las directrices del proceso propuesto, lo que resultó en el Manual de

Instanciación de GREN [3], denominado "Cookbook de GREN". Él contiene cuatro tablas que permiten mapear GRN a GREN. Las dos primeras tablas corresponden a las "Tablas de Mapeo de clases" y las dos últimas tablas a las "Tablas de Mapeo de los Métodos". Esas tablas fueron divididas para separar las clases de la capa de negocios de las clases de la capa de interfaz gráfica de usuario.

Una muestra de esas dos primeras tablas es presentada en las Tablas 3 y 4. La Tabla 3 es utilizada en la creación de las clases concretas de la capa de negocios y la Tabla 4 en la creación de las clases de la capa GUI. Las clases de la capa de negocios se preocupan sólo de la funcionalidad del sistema. Cada clase de esa capa puede tener una o más clases correspondientes en la capa GUI, lidiando con los aspectos de entrada y salida de datos. Se debe notar que esas tablas fueron construidas con base en el lenguaje de patrones GRN.

Tabla 10. Ejemplo de documentación de GREN - Tabla usada para identificar nuevas clases de aplicación.

Patrón	Variante	Clases de Patrón	Clases de GREN	Cod. Ref.
1. Identificar el Recurso	Todas	Recurso	Resource	N1
	Default, Tipos Múltiples	Tipo de Recurso	SimpleType	N2
	Tipos Anidados	Tipo de Recurso	NestedType	N2
2. Cuantificar el Recurso	Recurso Instanciable	Instancia de Recurso	ResourceInstance	N3
	Recurso Mensurable	Unidades de Recurso	MeasureUnity	N4
	Recurso en Lote	Lote de Recurso	ResourceLot	N5
	Recurso Simple	*nada que hacer*	-	-
9. Mantener el Recurso	Todas	Mantención de Recurso	ResourceMaintenance	N21
		Origen	SourceParty	N6
		Destino	DestinationParty	N14

Tabla 11. Ejemplo de documentación de GREN - Tabla para identificar nuevas clases GUI.

Patrón	Variante	Cod. Ref.	Clase GREN
1. Identificar el Recurso	Default, Tipos Múltiples	N2	StaticObjectForm
	Tipos Anidados	N2	QualificableObjectForm
2. Cuantificar el Recurso	Recurso Simple	N1	SingleResourceForm
	Recurso Instanciable	N1	MeasurableResourceForm
	Recurso Mensurable	N1	InstantiableResourceForm
	Recurso en Lote	N1	LotableResourceForm
	Recurso Instanciable	N3	ResourceInstanceForm
	Recurso Mensurable	N4	MeasureUnityForm
	Recurso en Lote	N5	*aún no implementado*
9. Mantener el Recurso	Aplicando patrones 14 y 15	N21	OneResourceMaintWPWTForm
	Sin aplicar patrones 14 y 15	N21	OneResourceMaintNPNTForm
	Todas	N6	SourcePartyForm
	Todas	N14	DestinationPartyForm

Un ejemplo de las tablas de mapeo de los métodos de GREN es presentado en la Tabla 4. Ella lista los métodos que son sobrepuestos por las nuevas clases creadas y debe ser recorrida secuencialmente, usando informaciones sobre la jerarquía de las nuevas clases de la aplicación siendo instanciada.

Cuando se encuentra un método que pertenece a una clase abstracta de GREN, especializada durante la instanciación, entonces ese método debe ser sobrepuesto. La columna "Instancia/Clase" determina si el método será invocado por instancias de la clase o por la propia clase. La columna "Comentarios" describe la función del método de forma suficiente para que el desarrollador pueda codificarlo.

El "Cookbook de GREN" posee también algoritmos para que sean utilizados en conjunto con las tablas, para permitir el uso disciplinado de esas tablas en la obtención de las clases y métodos que son implementados.

Tabla 12. Ejemplo de métodos que son sobrepuesto en GREN.

Super-Clase	Método	Instancia/Clase	Comentario
QualifiableObject	typeClass	C	retorna un objeto List con las clases que representan el tipo del recurso (cero o más).
ResourceMaintenance resource	Class	C	Retorna la clase que representa el recurso siendo mantenido.
	hasSourceParty	C	Retorna un booleano, siendo true si el participante "Origen" fue utilizado durante la instanciación del patrón 2, o false de lo contrario, ya que ese participante es opcional en ese
	sourcePartyClass	C	Retorna el nombre de la clase de la aplicación que desempeña el papel de Destino en el patrón. Ese método solamente debe ser sobrepuesto si el participante Destino fue utilizado durante la instanciación de ese patrón.
	destinationPartyClass	C	Retorna el nombre de la clase de la aplicación que desempeña el papel de Destino en el patrón.

4. Validación del framework.

El cuarto y último paso de la construcción de GREN es su validación, con el objetivo de verificar que las aplicaciones producidas a partir de él funcionaban correctamente. GREN fue probado, por la doctora Braga [3], para tres aplicaciones diferentes, escogidas cuidadosamente de modo de ejercitar los quince patrones de GRN. Sin embargo, no fue posible, durante esas pruebas, validar todas las variantes y sub-patrones, lo que se pretende

hacer en futuros esfuerzos. Las aplicaciones utilizadas en la prueba fueron: un taller de reparación de vehículos, una alquiladora de vídeos y una tienda de venta y alquiler de productos para fiestas. Después del término de esas pruebas, fueron hechos algunos estudios con alumnos de graduación y postgrado, los cuáles recibieron la tarea de desarrollar dos aplicaciones, más específicamente para un hotel y para una alquiladora de automóviles, con base en requisitos previamente suministrados.

Apéndice C. Ejemplo: GREN-Wizard una herramienta de soporte a GREN y a GRN [3].

GREN-Wizard [3] es una herramienta para auxiliar la instanciación del framework GREN (descrito en la sección anterior), con base en el lenguaje de patrones GRN (descrito en el apéndice A). GREN-Wizard fue proyectado de forma que sus usuarios puedan generar, sólo con el conocimiento sobre GRN, aplicaciones específicas en el dominio de gestión de recursos de negocios. La herramienta fue implementada usando el ambiente VisualWorks 5i.4, con datos obtenidos de una base de datos MySQL, teniendo 30 clases y aproximadamente veinte mil líneas de código en Smalltalk. La jerarquía de clases de GREN-Wizard es presentada en el Apéndice D.

GREN-Wizard es alimentado con informaciones sobre el uso de GRN en el modelado de la aplicación específica y retorna como resultado el código-fuente, en Smalltalk, de las clases especializadas a partir de GREN. Se crean, también, las tablas en la base de datos MySQL, de acuerdo con los patrones de GRN que fueron utilizados. Para ejecutar la aplicación resultante basta juntarse, al código de GREN, el código de las clases generadas por GREN-Wizard.

La interacción del ingeniero de aplicaciones con la GUI de GREN-Wizard es basada en los conceptos de GRN, por lo tanto, se debe informar de cuáles patrones son usados, cuáles variantes son más adecuadas para cada aplicación específica y cuáles clases desempeñan cada papel en la variante escogida. Después de aplicar un determinado patrón, se debe escoger cual es el patrón siguiente a aplicar, respetando las posibilidades de navegación pre-establecidas por el ingeniero de dominio que construyó el lenguaje de patrones. GREN-Wizard es utilizado paralelamente a GRN, es decir, se utiliza GRN para modelar el sistema, produciéndose un modelo de análisis y un histórico de patrones y variantes aplicadas. Esa

información es utilizada para llenar las tablas de GREN-Wizard y producir el código Smalltalk necesario para adaptar el GREN a la aplicación específica.

Apéndice D. Ejemplo de documentación adicional de GREN y GREN-Wizard [3].

1. Jerarquía de clases de GREN.

Las clases de VisualWorks aparecen en itálico. Las demás son las clases de GREN. Los atributos de cada clase aparecen entre paréntesis. Para dar la noción de jerarquía, fueron incluidos tres trazos para cada nivel de especialización (por ejemplo, InterestRate es especializada de ExactNumberCalculator, que es especializada de AbstractCalculator, y así por delante).

Capa de Persistencia.

Object ()

---ConnectionManager ()

---OIDManager ('className' 'lastIdCode')

---PersistentObject ('isChanged' 'isPers isted')

Capa de Negocios.

Object()

---DatePeriod('initialDate' 'finishingDate')

---DiscreteList ('values')

---PersistentObject ('isChanged' 'isPersisted')

------AbstractCalculator ('percentageRate')

---------ExactNumberCalculator ('number')

----------InterestRate ()

---------NumberRangeCalculator ('lowerNumber' 'upperNumber')

------------FineRate ()

------BusinessResourceTransaction ('number' 'date' 'observation' 'status' 'totalPrice' 'totalDiscount' 'finalTotal' 'destinationParty' 'sourceParty' 'resource' 'items' 'executor' 'transQuantification')

---------BasicMaintenance ('faultsPresented' 'tasks' 'parts' 'totalTasks' 'totalParts')

------------MaintenanceQuotation ('expirationDate' 'maintenanceNumber')

------------ResourceMaintenance ('exitDate' 'quotationNumber')

---------BasicNegotiation ('freight' 'taxes' 'oldQtty')

------------BasicDelivery ('tradeNumber')

---------------PurchaseDelivery ()

---------------SaleDelivery ()

------------ResourceTrade ('quotationNumber')

---------------BasicPurchase ()

---------------BasicSale ()

---------ResourceRental ('finishingDate' 'returnDate' 'reservationNumber' 'fineValue')

---------ResourceReservation ('period')

------BusinessResourceQuotation ('number' 'date' 'observation' 'status' 'sourceParty' 'quotationItems' 'executor')

------Commission ('transaction' 'executor' 'date' 'aValue')

------LocationZoneStCond ('locationZoneIdCode' 'stCond' 'condition')

------MaintenancePart ('transaction' 'part' 'qtty' 'cost')

------MaintenanceTask ('transaction' 'problemToSolve' 'laborDescription' 'hoursSpent' 'cost' 'executor')

------NToNRelationship('obj1' 'obj2')

------Payment ('dueDate' 'paymentDate' 'installmentNumber' 'paymentType' 'value' 'status')

------PaymentStrategy ('transaction' 'installmentNumber')

---------ImmediateReceiving ()

------------Cash ('suppliedValue')

------------MoneyOrder ('bank')

------------ElectronicTransfer ('cardNumber' 'bank')

---------LaterReceiving ()

------------CashOnDelivery ('paymentForm')

------------Check ('accountNumber' 'bank' 'checkNumber')

------------CreditCard ('cardNumber' 'type' 'bank' 'expirationDate' 'paymentDay')

------------Invoice ('invoiceNumber')

------QuotationItem ('transaction' 'destinationParty' 'resource' 'price' 'expirationDate' 'quantity')

------ResourceInstance ('resourceIdCode' 'code' 'allocation' 'status')

------ResourceLot ('number' 'qtty')

------ResourceStCond ('resourceIdCode' 'stCond' 'condition')

------StaticObject ('idCode' 'description')

---------DestinationParty ()

---------LocationZone ('capacity' 'storageConditions' 'storageLocationIdCode')

---------MeasureUnity ()

---------QualifiableObject ('types')

----------NestedType ()

----------Resource ('quantification')

----------StockResource('cost' 'salePrice')

----------StorableResource ('locationZone' 'storageConditions')

----------SimpleType()

----------Warehouse ('storageLocations')

---------SourceParty ()

---------StorageCondition ()

---------StorageLocation ('locationZones' 'warehouseIdCode')

---------TransactionExecutor ('specialty' 'percentage' 'minimumValue' 'salary')

------TransactionItem ('transaction' 'resource' 'aValue' 'itemQuantification')

------QuantificationStrategy ('resource')

---------InstantiableResource ('instances')

---------MeasurableResource ('quantityInStock' 'reSupplyLevel' 'measureUnity')

----------LotableResource ('lotNumber')

---------SingleResource ('status')

------TransactionQuantificationStrategy ('transaction')

---------InstantiableResTransaction ('instanceCode')

---------ItemQuantificationStrategy ('transactionItem')

----------InstResTransItem ('instanceCode')

----------MeasResTransItem ('quantity')

----------LotResTransItem ('lotNumber')

----------SingleResTransItem ()

---------MeasurableResTransaction ('quantity' 'unitaryValue')

----------LotableResTransaction ('lotNumber')

---------SingleResTransaction ()

---Label ('linesPerPage' 'spaces' 'rows' 'columns' 'title' 'header' 'showDate')

---Report ('header' 'title' 'showDate' 'showPageNumber' 'linesPerPage' 'spaces' 'rows' 'columns')

---ReportColumn ('number' 'label' 'width' 'group' 'classify' 'totalize' 'totalizeGroup' 'calculatePercentage')

2. Jerarquía de clases de GREN-Wizard.

Object ()

---GRNHistory ('aHistory' 'appIdCode')

---*PersistentObject* ('isChanged' 'isPersisted')

------AppClass ('appClassIdCode' 'appIdCode' 'patternNumber' 'variantNumber' 'classNumber' 'name' 'plural')

---------TempAppClass()

------AppClassAttribute ('appIdCode' 'appClassName' 'sequentialNb' 'patternAttributeName' 'attributeName'

'attributeType' 'attributeLength' 'canDelete')

------AppliedPattern ('patternNumber' 'variantNumber')

------AppReport ('appIdCode' 'sequentialNb' 'reportNb' 'title')

------AttributeMapping ('p_name' 'p_type' 'p_length' 'app_name' 'app_type' 'app_length' 'canDelete')

------ClassMethod ('seq' 'className' 'methodName' 'methodType' 'protocolName' 'preCondition' 'methodBody')

---------ClassAtrMethod ()

---------ClassSpecia lTypeMethod('atrType')

------Element ('appIdCode' 'sequentialNb' 'patternNumber' 'variantNumber' 'transactionClassName' 'resourceClassName' 'assocTransClassName')

---------OldElement ('visited' 'toBeDeleted')

------EquivalentPatternClass ('patternNumber' 'variantNumber' 'classNumber' 'ePatternNumber' 'eVariantNumber' 'eClassNumber')

------GRENApplication ('idCode' 'name' 'databaseName' 'language' 'databaseLocation' 'aGRNHistory' 'oldGRNHistory')

------GRENReport ('superClassName' 'grenReportMethod' 'methodNb' 'reportType')

------NextPattern ('patternNumber' 'nextPattern')

------Pattern ('patternNumber' 'patternName' 'specName' 'mandatory')

------PatternClass ('patternNumber' 'variantNumber' 'classNumber' 'className' 'pluralClassName' 'superclassName' ‘refCode' 'canIncludeAttribute' 'mandatory')

'attributeLength' 'isMandatory')

------PatternClassForm ('refCode' 'seq' 'patternNumber' 'variantNumber' 'superclassName' 'specName' 'addToMenu' 'nextHorizontal' 'nextVertical')

------PatternClassAttribute ('patternNumber' 'variantNumber' 'classNumber' 'attributeName' 'attributeType'

------PatternReport ('patternNumber' 'className' 'operation' 'reportNb' 'grenReportMethod'

'grenReportTitleMethod' 'methodNb')

------PatternVariant ('patternNumber' 'variantNumber' 'description')

------SelectableReport ('number' 'name' 'title' 'wasSelected')

------*StaticObject* ('idCode' 'description')

---------AttributeType ('initializeString')

------TypeAppClass('appIdCode' 'appClassName' 'plural ')

---*Model* ('dependents')

------*ApplicationModel* ('builder' 'uiSession')

---------*GRENApplicationMainForm* ()

------------GrenWizardGUI ('aPatForm' 'editing' 'app' 'appsList' 'sequenceNb' 'transName' 'transPlName'

'resName' 'resPlName' 'assocTransName' 'lastNonLeafSeqNb' 'updateRes' 'updateTrans' 'isChanged'

'lastVisited' 'newGeneratedApp' 'aGenWin')

---------*GRENApplicationModel* ('parentApplication')

------------AttributeMappingForm ('anAttribute' 'aClassName' 'allAttributes' 'app_length' 'p_type' 'app_type' 'app_name'

p_name' 'sequentialNb' 'p_length' 'isChanged' 'rolePlayed')

------------AttributesImportation ('classesPlayingRole' 'listOfAddedAttributes' 'aSuperClassName')

----------GenerationWindow ('consistency' 'application' 'attributes' 'interface' 'menus' 'methods')

----------GrenWizardCodeGenerator ('app' 'appNameSpace' 'menuClass' 'aConcreteClassName' 'anAddedAttribute'

aClassMethod')

---------- PatternForm ('wasApplied' 'wasSkipped' 'patternNumber' 'variantApplied' 'allVariants' 'variantChosen'

nextPattern' 'nextPatternChosen' 'allNextPatterns' 'currentElement' 'adding' 'onlyRevisiting'

class1Name' 'class2Name' 'class3Name' 'class4Name' 'class5Name' 'class6Name' 'class7Name'

class8Name' 'class9Name' 'class10Name' 'class11Name' 'class12Name' 'plClass1Name' 'plClass2Name'

plClass3Name' 'plClass4Name' 'plClass5Name' 'plClass6Name' 'plClass7Name' 'plClass8Name'

plClass9Name' 'plClass10Name' 'plClass11Name' 'plClass12Name')

----------ReportsSelection ('aTitle' 'history' 'appliedPatterns' 'selectedReports' 'listOfReports' 'listOfClasses'

isChanged')

----------HtmlForms ('introductionText')

----------AttributeListForm('listApp_type' 'list_tables' 'list_attributes' 'isChanged')

---------- AttributeDiscreteListForm()

---------- AttributeTableListForm()

----------CreateListForm('nameList' 'valuesList' 'isChanged')

----------DiscreteListForm()

----------TableListForm()

---------PatternLanguageDatabase ()

---------*SimpleDialog* ('close' 'accept' 'cancel' 'preBuildBlock' 'postBuildBlock' 'postOpenBlock' 'escapeIsCancel'

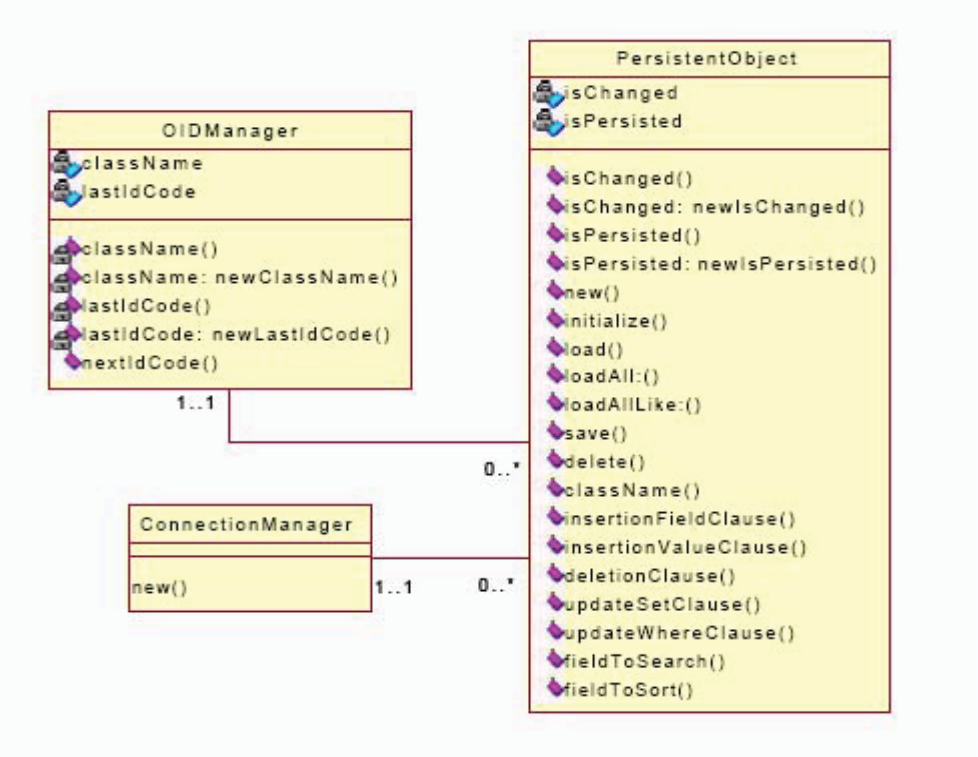
parentView')

----------GRENApplicationForm ('anApplication' 'idCode' 'name' 'databaseName' 'language' 'databaseLocation')

3. Diagrama de clases de GREN.

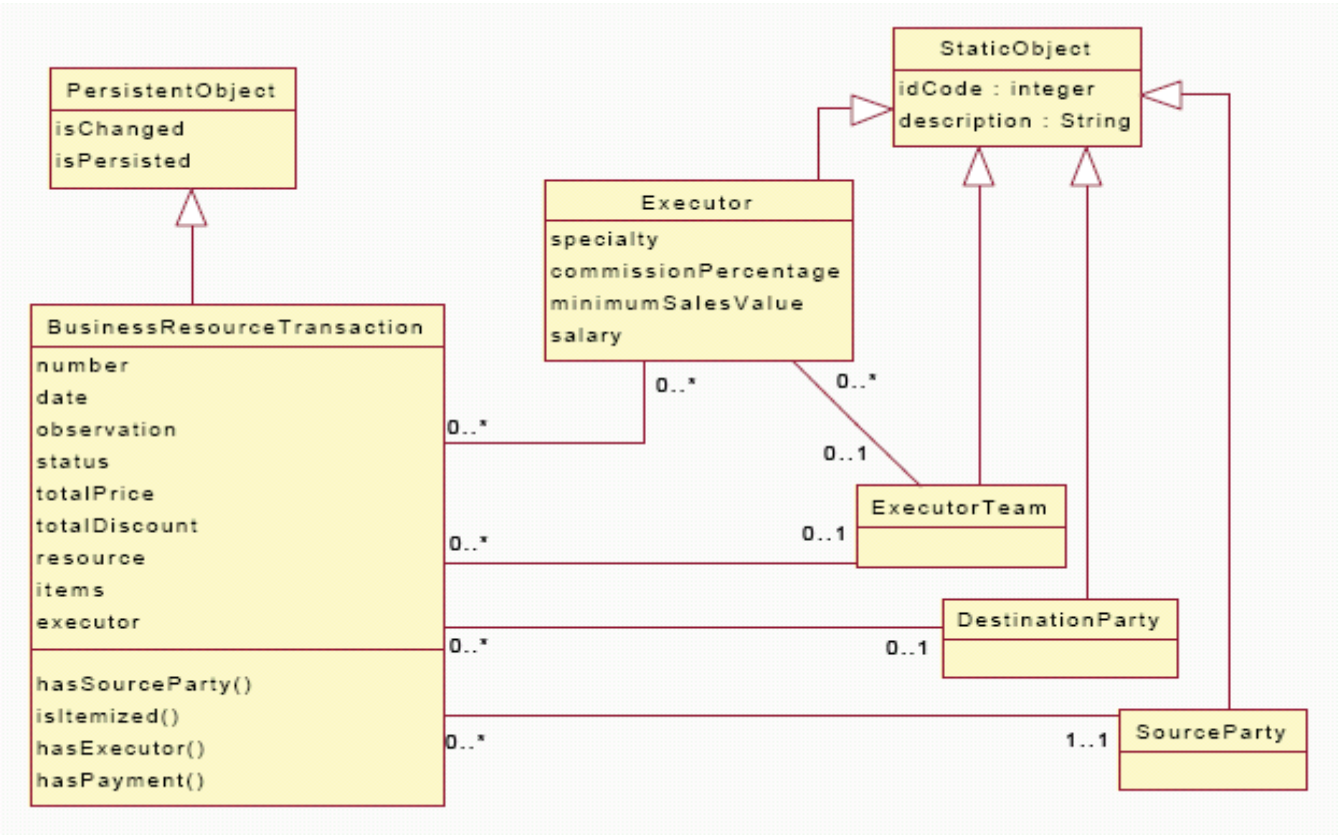
Capa de Persistencia.

Figura D.1. Capa de persistencia del framework de caja blanca GREN [3].



Capa de Negocios - Transacciones de Negocios - Patrón 13.

Figura D.2. Capa de negocios del framework de caja blanca GREN [3].


StaticObject
idCode : integer
description : String
PersistentObject
isChanged
isPersisted
Executor
specialty
commissionPercentage
minimumSalesValue
salary
BusinessResourceTransaction
number
date
observation
status
totalPrice
totalDiscount
resource
items
executor
hasSourceParty()
isItemized()
hasExecutor()
hasPayment()
0..*
0..*
0..*
0..1
0..*
0..1
0..*
0..1
0..*
1..1
ExecutorTeam
DestinationParty
SourceParty