
Facultad de Informática

Trabajo Fin de Grado
Grado en Ingeniería Informática
mención en ingeniería del Software

# *ESTUDIO, DEFINICIÓN E IMPLEMENTACIÓN DE PATRONES DE PRUEBA INSPIRADOS EN GoF*

**Autor/a:** Nicolás Echevarrieta Catalán
**Director/a:** Laura M. Castro Souto

OKI
Colaborador Oficial en
Soluciones de Impresión

*A Coruña, a 10 de febrero de 2016.*

# Agradecimientos

A mi familia y amigos por aguantarme y a mi tutora por guiarme en el proceso de realización del proyecto.

**Resumen**

El primer objetivo del presente proyecto es el diseño de un conjunto genérico de pruebas o esqueletos de pruebas, funcionales y no funcionales, para los patrones de diseño de la ingeniería del software. En concreto se trabajó con los descritos por Eric Gamma, Richard Helm, Ralph Johnson y John Vlissides (habitualmente conocidos como "Gang of Four" o GoF). Para que dichas pruebas sean genéricas, se abstrajeron de los efectos que el patrón puede tener más allá de su funcionalidad explícita.

El segundo objetivo es, con las pruebas diseñadas, realizar el análisis, diseño, implementación y validación de un software en dos capas que permita la generación automática de dichas pruebas. La primera capa del sistema identifica el patrón (mediante marcadores sobre el código o a partir de ficheros de salida de herramientas generadoras de UML o cualquier otro método de entrada de datos adecuado) y las características necesarias de la implementación concreta (nombres de clases, métodos, etc.) para generar una representación genérica del patrón (independiente del lenguaje de implementación de las pruebas) en XML. La segunda capa del sistema produce las pruebas de forma automatizada a partir de dicha representación, según la tecnología deseada.

En concreto, se ha implementado soporte solo para un lenguaje y tecnología (elegidos por la familiaridad que tiene con ellos el autor): Java con JUnit, a través de marcadores en código. Sin embargo, gracias a la estructura en dos capas y la representación intermedia diseñadas, se simplifica la ampliación para soportar más formas de definición de los patrones, lenguajes y tecnologías de prueba.

El software se ha pretendido que simplifique y mejore las pruebas realizadas sobre los patrones de diseño. Para que eso se cumpla, es necesario que sea usado por desarrolladores de software, por lo que es una herramienta de código abierto, de forma que pueda ser adaptada por parte de sus usuarios a necesidades más concretas de las tenidas en cuenta en su desarrollo, así como ampliada para otras tecnologías y/o lenguajes.

**Palabras clave:**

- √ Pruebas funcionales.
- √ Pruebas no funcionales.
- √ Generación automatizada.
- √ Patrones de diseño.
- √ Java
- √ JUnit
- √ Maven
- √ XML

# Índice general

**Página**

# Índice de figuras

**Figura** **Página**

# Índice de cuadros

**Tabla** **Página**

# Capítulo 1

# Introducción

## Índice general



En este capítulo se describen los rasgos generales del proyecto en su conjunto.

## 1.1. Motivación

En la actualidad los patrones de diseño de la ingeniería del software se han convertido en un conocimiento obligatorio para cualquiera que se dedique a la profesión, pues permiten resolver de forma elegante y efectiva situaciones muy habituales.

No obstante, pese a su extendido uso, no existen herramientas ni librerías de pruebas genéricas que puedan aplicarse sobre el patrón en sí, lo que en la práctica lleva a realizarlas *ad-hoc* para cada implementación mezclándose con las pruebas del problema concreto al que se ha aplicado el patrón.

Por ejemplo, la función del patrón Observador es que, al modificarse el componente observado, todo componente observador sea avisado y analice los cambios en el observado, realizando la lógica que le corresponda en caso de considerarlo necesario. Qué particularidades del cambio en el observado analice cada observador y qué lógica correspondiente debe realizar es particular de cada uso del patrón y no interfiere con la funcionalidad propia del patrón. Dicho de otro modo, sea cual sea la lógica de negocio de observado y observadores, cualquier implementación de este patrón debe ir acompañada de las pruebas correspondientes relacionadas con la adecuada creación y recepción de notificaciones.

## 1.2. Objetivos

Con la motivación presentada en la sección anterior como base, el presente trabajo pretende analizar algunos patrones de los descritos en el libro de GoF [7], catalogados dentro de la categoría "de comportamiento", con la intención de definir un conjunto de pruebas genéricas, independientes de la implementación concreta, que permitan probar cualquier implementación del patrón de forma automatizada y lo más completa posible, con la mínima interacción necesaria por parte del usuario. En concreto, se pretende realizar el soporte del sistema para los patrones Estado, Estrategia y Observador.

Con este producto, se espera motivar e incentivar el uso consciente e intencionado de los patrones por parte de los desarrolladores de software, al permitirles probar una parte importante de su software (ya que los patrones elegidos son los que más intervienen en la lógica de la aplicación) con un mínimo de esfuerzo y la fiabilidad de que son un conjunto completo y estable de pruebas con capacidad de detectar los fallos que buscan, en caso de existir.

Como consecuencia última se cree que esto contribuirá a mejorar la calidad del software en general, pues si bien las pruebas no garantizan la calidad del producto, sí que ayudan a verificar que el software alcanza unos mínimos validables, y consituye un elemento de mantenibilidad muy valioso. Además, el incentivar el uso de patrones también ayudará a mejorar la calidad del software, pues éstos

permiten solucionar problemáticas del desarrollo de forma eficaz, escalable y, gracias a herramientas como esta, fácilmente validable.

Para que el proyecto pueda ser utilizable, se va a poner a disposición de quien lo quiera mediante un repositorio git [19] público alojado en la URL:

`https://bitbucket.org/Gildarth/tfg_sw.git`

Un punto importante del trabajo es la escalabilidad del producto, de forma que añadir más lenguajes o tecnologías de pruebas automatizadas (como es el caso de JUnit para Java por ejemplo) sea lo más sencillo posible. Para ello, se separará en dos módulos comunicados por una representación intermedia abstracta e independiente. El primer módulo será el encargado de generar la representación intermedia del patrón, siendo el segundo el encargado de entenderla y generar los ficheros de prueba correspondientes. Gracias a la representación intermedia, cada módulo es independiente del otro. No obstante, se considera buena idea estandarizar una interfaz externa para cada uno, de forma que pueda ser utilizado por otros sistemas a modo de componente (por ejemplo, que pueda ser integrado dentro de un entorno de desarrollo como Eclipse [21]).

En resumen, y por orden de relevancia, los objetivos concretos establecidos son:

**Objetivo-1** Que el sistema sea escalable, de forma que añadir el soporte a nuevos patrones suponga el menor esfuerzo posible.

**Objetivo-2** Analizar el patrón Estado y diseñar un conjunto de pruebas genéricas, independientes y de generación lo más automatizada posible.

**Objetivo-3** Implementar el soporte del sistema para el patrón Estado.

**Objetivo-4** Analizar el patrón Estrategia y diseñar un conjunto de pruebas genéricas, independientes y de generación lo más automatizada posible.

**Objetivo-5** Implementar el soporte del sistema para el patrón Estrategia.

**Objetivo-6** Analizar el patrón Observador y diseñar un conjunto de pruebas genéricas, independientes y de generación lo más automatizada posible.

**Objetivo-7** Implementar el soporte del sistema para el patrón Observador.

El esfuerzo total estimado para este tipo de proyecto debe rondar las 300 horas×persona, a razón de tener una valoración de 12 créditos ECTS y el esfuerzo estimado para cada crédito ECTS ser de 25 horas×persona.

La fecha de inicio es el 17 de julio de 2015 y la de finalización el 10 de febrero de 2016 (fecha de depósito de la memoria).

## 1.3. Metodología

Debido a la estabilidad del contexto del trabajo y la forma del software planteado, se ha decidido seguir un ciclo de vida en espiral[20], en el cual a cada iteración se añadirá un nuevo patrón al sistema.

Durante todo el desarrollo se van a seguir las convenciones de nombrado que se describen en la sección 2.10.

## 1.4. Estructura de este documento

La presente memoria se estructura en: introducción (la presente parte), contextualización, descripción de requisitos, detalle de planificación, análisis y diseño, implementación y validación, seguimiento, y conclusiones.

En el capítulo de contextualización se describirán todos aquellos componentes y conocimientos necesarios para entender el presente trabajo. Aquí se incluye la sección de nomenclatura, donde se explican las diferentes convenciones de nombrado adoptadas en el proyecto.

En el capítulo de descripción de requisitos se detallan aquellos requisitos que componen el proyecto y se catalogan según si son o no funcionales y según su relevancia en el producto final. La valoración de esfuerzo para cada requisito se deja para la planificación, pues el mismo requisito común en las distintas iteraciones pueden requerir un esfuerzo diferente.

Las iteraciones se presentan repartidas en cuatro capítulos: Planificación, Análisis y diseño, Implementación y Validación, y Seguimiento; en ellos se detalla el desarrollo del proyecto siguiendo las fases del ciclo de vida en espiral.

En las conclusiones se presenta un análisis de los problemas surgidos durante el desarrollo del trabajo y las lecciones aprendidas de dichos problemas, seguido del trabajo que ha quedado más allá del alcance de este trabajo, así como las vías de desarrollo futuro que se ven para el sistema.

# Capítulo 2

# Contextualización

## Índice general

Este capítulo tiene por objetivo exponer aquellos conocimientos necesarios para entender el presente proyecto, desde el contexto histórico en el que se desarrolla hasta los conceptos, tecnologías y herramientas utilizados.

## 2.1. Historia de los patrones y definición

En esta sección se pretende dar una visión histórica global sobre los patrones de diseño y su evolución, así como diversas definiciones actuales.

Una de las primeras definiciones de lo que es un patrón de diseño fue documentada en el área de la arquitectura por Christopher Alexander en 1979 en su libro "*The Timeless Way of Building*"[2], donde escribe: "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez".

Una de las primeras apariciones en la ingeniería del software la protagonizaron Ward Cunningham y Kent Beck en 1987 en su libro "*Using Pattern Languages for OO Programs*"[5], donde usaron varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-ordenador (HCI).

Pero no sería hasta casi una década más tarde, en 1995, cuando Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, los llamados "Gang of Four" (GoF), publicaron su libro "*Design Patterns: Elements of Reusable Object-Oriented Software*", donde ofrecieron una definición ampliamente aceptada aun a día de hoy:

> *"Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias."*

También en palabras de GoF, los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores, estandarizando el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

## 2.2. Patrones de diseño software

Como hemos comentado, los patrones del diseño de software son aquéllos aplicados a la ingeniería del software y se catalogan en 3 grupos basándose en sus funciones [7]: creacionales (creación y destrucción de objetos), estructurales (relaciones entre clases e interfaces) y de comportamiento (delegación del comportamiento entre diferentes clases).

Por ser los concernientes a este proyecto, nos centraremos en los patrones encargados del comportamiento.

### 2.2.1. Patrones de comportamiento

Los patrones de comportamiento son aquéllos que están relacionados con algoritmos y con la asignación de responsabilidades de soporte a la lógica de negocio a los objetos.

Describen no solamente estructuras de relación entre objetos o de clases, sino también engloban esquemas de comunicación entre ellos y reparto de roles. Al

igual que los otros tipos de patrones, se pueden clasificar en función de que trabajen con clases (*Template Method, Interpreter*) u objetos (*Chain of Responsability, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor*).

A continuación se detallan un poco más aquellos patrones en concreto a los que se espera dar soporte en este proyecto, incluyéndose un análisis más detallado en el apartado correspondiente al análisis y diseño (capítulo 5).

#### 2.2.1.1. Estado

El patrón de diseño Estado (*State*)[7] se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo. Surgió para solucionar el caso en que un contexto que se está desarrollando requiere tener diferentes comportamientos según el estado en que se encuentra, resultando complicado manejar el cambio de comportamientos dentro del mismo bloque de código. El patrón Estado propone una solución a esta complejidad, creando un objeto por cada estado posible donde encapsular la lógica relacionada.

Se ofrece una interfaz estado genérica con un método para cada comportamiento que dependa del estado en que se encuentre el contexto. Este estado genérico se implementa para cada estado diferente en el que pueda estar el contexto y se da una implementación a cada acción variable en función del estado que se represente. El contexto entonces pasa a delegar la ejecución de estas acciones en el estado actual que contenga, siendo el propio estado el encargado de modificar el estado del contexto si procede (son los propios estados los encargados de modificar el estado interno del contexto). Con esto se consigue que el contexto pueda ejecutar sus funcionalidades y modificar el resultado de aquellas dependientes del estado durante la ejecución, sin necesidad de ser consciente de en qué estado se encuentra.

El acoplamiento entre el contexto y la jerarquía de estados es, naturalmente, fuerte. En contraste, la jerarquía de estados no es visible a los objetos clientes del contexto, lo cual también contribuye a la mantenibilidad y extensibilidad de la solución que encarna este patrón.

Un ejemplo sencillo de aplicación del patrón estado es un servicio TCP. Ante el mismo mensaje recibido, el comportamiento y respuestas devueltos dependerán del estado en el que se encuentre el servicio, y a su vez conducirán a la conexión a un estado concreto dependiendo del caso.

#### 2.2.1.2. Estrategia

El patrón Estrategia (*Strategy*)[7] permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón estrategia. Puede haber cualquier número de estrategias y cualquiera de ellas debe poder ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución, sin que el resultado final de ejecutarla varíe.

Un ejemplo de esto son los algoritmos de ordenación, los cuales todos cumplen el mismo objetivo y dan el mismo resultado, pero de diferente manera.

Esto se hace mediante una interfaz común que será implementada por cada estrategia concreta. Lo que diferencia este patrón de una simple especialización es el hecho de que cambiar la implementación concreta que se esté usando no conlleva cambios en aspectos funcionales de la aplicación, pero sí puede hacerlo en aspectos no funcionales como el rendimiento en función del caso de trabajo concreto.

Aunque la estructura de clases propuestas es similar al del patrón Estado, el acoplamiento entre el objeto cliente y las estrategias en este caso es mínimo; de hecho, las estrategias son independientes de sus clientes, y tienen vocación de ser reutilizables, por lo que no restringen su visibilidad dentro del sistema.

#### 2.2.1.3. Observador

El patrón de comportamiento Observador (*Observer*)[7] define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando el uno cambia su estado, notifica este cambio a todos los dependientes.

Este patrón también se conoce como el patrón de suscripción-publicación. Este nombre sugiere la idea básica del patrón, que es: el objeto de interés (observado), se responsabiliza de gestionar y notificar al conjunto de objetos interesados (observadores) que se pueden suscribir a él. Los observadores tienen una dependencia con el observado, ya que tras ser notificados pueden enviar peticiones al mismo, según la lógica que implementen. Eso sí, los observadores son independientes entre sí, no tienen conocimiento mutuo, y por tanto el orden de notificación de los observadores y el orden de actuación de los observadores sobre el observado debe implementarse de manera coherente en cada entorno de aplicación.

Este patrón suele aplicarse en los *frameworks* de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir *listeners* a los objetos que pueden disparar eventos. También es la clave del patrón de arquitectura Modelo Vista Controlador (MVC)[3].

## 2.3. Pruebas

Las pruebas software [8] son el principal medio que se tiene para buscar errores en un sistema o aplicación en construcción. Existen diferentes tipos de pruebas en función de en qué punto del sistema se busquen los errores, la estrategia utilizada para localizarlos, o el tipo de errores que se persiguen, pero todas tienen en común que se consideran exitosas cuando encuentran un error (ya que es entonces cuando cumplen su cometido y demuestran su valor).

Por norma general, el tamaño del software actual hace imposible probar un programa o sistema por completo, e incluso el mejor conjunto de pruebas no puede garantizar la ausencia de fallos sin revelar. Dicho de otro modo, tanto por la imposibilidad de probarlo todo, como por el hecho de que las pruebas en

sí mismas son código fuente (y por tanto, sujetas a errores en sí mismas), éstas nunca podrán demostrar que un software está libre de fallos. Lo único que pueden demostrar es o bien que el software probado tiene algún fallo, o que no se ha sido capaz de descubrir ningún fallo con esas pruebas.

Una buena práctica a la hora de desarrollar software es el tener al menos una prueba por cada unidad de implementación, funcionalidad y requisito, llegando al punto de que algunas metodologías ágiles modernas sugieren escribir las pruebas antes incluso que el programa.

Habitualmente se clasifican las pruebas en cuatro niveles:

**Pruebas unitarias** utilizadas para secciones específicas y delimitadas, sin dependencias de otras unidades de implementación (servicios, otros módulos, etc...).

**Pruebas de integración** cuya finalidad es probar la correcta interrelación de componentes.

**Pruebas de sistema** realizadas por los desarrolladores sobre el sistema software como un todo.

**Pruebas de aceptación** llevadas a cabo por el usuario final sobre el software completo.

Otras clasificaciones complementarias habituales son, según el conocimiento que se tiene del software a probar:

**Pruebas de caja blanca** cuando se realizan utilizando conocimiento de cómo es internamente.

**Pruebas de caja negra** cuando no se conoce o se tiene acceso a la estructura interna (implementación) de lo probado y solo se trabaja en base a su API externa (entradas y salidas).

Según el tipo de requisito a probar, también se puede hablar de:

**Pruebas funcionales** para probar que el software cumple con la funcionalidad de negocio esperada.

**Pruebas no funcionales** cuando lo que se prueba es cómo el software lleva a cabo sus funcionalidades de negocio en términos de calidad (por ejemplo, consumo de recursos).

Según si ejecuta o no el código fuente:

**Pruebas estática** cuando analizan código pero no lo ejecutan.

**Pruebas dinámicas** en aquellos casos que la ejecución de la prueba requiera ejecutar el código probado.

Desarrollar un buen conjunto de pruebas automatizadas es de gran ayuda, sobretodo a largo plazo, pudiendo ejecutarlas siempre que se haga alguna modificación o ampliación en el software (denominándose en esta utilidad, "pruebas de regresión"), lo que ayuda a detectar errores de forma sencilla. Esto en proyectos grandes donde cualquier modificación pueda desencadenar (con mayor probabilidad cuanto peor sea el diseño) un efecto mariposa en la parte menos esperada, es de gran utilidad, pues permite probar los fallos que puede haber.

### 2.3.1. Pruebas unitarias

Como hemos dicho, las pruebas unitarias son aquellas cuya finalidad es probar, de forma atómica e independiente del resto del sistema, un fragmento de código o módulo. Para ello suelen usarse herramientas que permiten escribir comprobaciones centradas en cada una de las funcionalidades específicas, sin dependencias con otras. Un ejemplo de prueba unitaria sería la prueba de que la inserción de datos en una estructura resulta en la modificación de ésta, de manera que una consulta sobre la misma revela que efectivamente los datos se han persistido.

### 2.3.2. Herramientas XUnit

Existen varios frameworks de ayuda al desarrollo de pruebas y comprobaciones a nivel de unidad, que han llegado a conocerse colectivamente como xUnit[25]. Tales frameworks están basados en un diseño de Kent Beck, implementados originalmente para Smalltalk como SUnit, pero están ahora disponibles para muchos lenguajes de programación y plataformas de desarrollo.

El diseño general de los frameworks xUnit[25] depende de varios componentes:

**Caso de prueba (*test case*)** es el componente elemental del framework. Todas las pruebas definidas por el desarrollador serán de este mismo tipo.

**Motor de pruebas (*test runner*)** es un programa ejecutable que corre las pruebas implementadas usando el framework XUnit y genera un informe con los resultados de dicha ejecución.

**Conjunto de pruebas (*test suite*)** está formado por el conjunto de casos de prueba agrupados en uno o varios módulos de pruebas. El orden de su ejecución no debe ser relevante.

**Accesorios de prueba (*test fixture*)** es un conjunto de precondiciones o estados necesarios para que se ejecute un caso de prueba. También se conoce a esto como contexto de prueba.

**Ejecución de prueba (*test execution*)** es la ejecución de un caso de prueba unitaria individual. Primero se prepara el contexto donde se ejecutarán los casos, seguido de la ejecución en orden no determinista de los mismos y para finalizar se limpia el contexto retornándolo a su estado anterior para que la ejecución de unas pruebas no influya en otras o en la ejecución normal del programa.

**Formateador de resultados (*result formatter*)** se encarga de generar el informe de ejecución en diferentes formatos, entre ellos habitualmente HTML o XML.

**Aserciones (*assertions*)** son funciones o macros que verifican el comportamiento (o estado) de aquello que está siendo probado. Generalmente son condiciones lógicas que comparan un resultado esperado con el obtenido tras la ejecución del código a probar.

Una prueba puede terminar con uno de estos tres resultados: que sea exitosa (todas las aserciones se han evaluado como verdaderas), que haya un error (una aserción se ha determinado falsa) o que se produzca un fallo (se ha producido algún error inesperado durante al ejecución).

### 2.3.3. JUnit

JUnit[13] es un framework de pruebas de la familia XUnit para el lenguaje Java. Originalmente fue escrito por Erich Gamma y Kent Beck y actualmente es un proyecto de código abierto bajo la licencia pública de Eclipse en su versión 1.0, estando almacenado su código en SourceForge[6].

Un accesorio de prueba JUnit es un objeto Java y sus métodos de prueba han de ir precedidos de la anotación `@Test`. Contiene anotaciones `@BeforeClass` y `@AfterClass` para especificar código que se debe ejecutar al iniciar la ejecución de la suite de pruebas y al terminarla (respectivamente). De manera similar, las anotaciones `@Before` y `@After` indican código que ha de ejecutarse antes y después, respectivamente, de cada prueba unitaria. También proporciona un conjunto de aserciones, siendo algunas de ellas `AssertEquals, AssertTrue y AssertFalse`. El orden de ejecución de las pruebas, como es habitual, no está asegurado.

### 2.3.4. XMLUnit

XMLUnit[24] es un framework de la familia XUnit[25] para Java[15] y .Net que añade aserciones para validar y comparar documentos XML, diferenciando estas comparaciones entre documentos idénticos (los mismos elementos formando la misma estructura arbórea en el mismo orden), similares (los mismos elemen-

tos formando la misma estructura arbórea en distinto orden) o distintos (algún elemento es hijo de un elemento diferente al esperado).

## 2.4. Ciclo de vida

El ciclo de vida [20] es el conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o remplazado (muere).

Entre las funciones que debe tener un ciclo de vida se pueden destacar:

- Determinar el orden de las fases del proceso de software
- Establecer los criterios de transición para pasar de una fase a la siguiente
- Definir las entradas y salidas de cada fase
- Describir los estados por los que pasa el producto
- Describir las actividades a realizar para transformar el producto
- Definir un esquema que sirve como base para planificar, organizar, coordinar y desarrollar.

El proceso para el desarrollo de software, también denominado ciclo de vida del desarrollo de software, es una estructura aplicada al desarrollo de un producto de software. Hay varios modelos a seguir para el establecimiento de un proceso para el desarrollo de software, cada uno de los cuales describe un enfoque diferente para diferentes actividades que tienen lugar durante el proceso.

### 2.4.1. Ciclo de vida en espiral

El desarrollo en espiral es un modelo de ciclo de vida del software definido por primera vez por Barry Boehm en 1986 en su artículo "*A Spiral Model of Software*

*Development and Enhancement*", y se utiliza con frecuencia en la ingeniería de software. Las actividades de este modelo se conforman en una espiral, en la que cada bucle o iteración representa un conjunto de actividades. Las actividades no están fijadas a ninguna prioridad, sino que las siguientes se eligen en función del análisis de riesgo, comenzando por el bucle interior.

En cada vuelta o iteración hay que tener en cuenta:

**Los Objetivos:** qué necesidad debe cubrir el producto.

**Alternativas:** las diferentes formas de conseguir los objetivos de forma exitosa, desde diferentes puntos de vista como pueden ser:

1. Características: experiencia del personal, requisitos a cumplir, etc.
2. Formas de gestión del sistema.
3. Riesgo asumido con cada alternativa.

**Desarrollar y Verificar:** programar y probar el software.

Si al final de un ciclo el resultado no es el adecuado o se necesita implementar mejoras o funcionalidades, se planificaran los siguientes pasos y se comienza un nuevo ciclo de la espiral. La espiral tiene una forma de caracola y se dice que mantiene dos dimensiones, la radial (indica el avance del proyecto del software dentro de un ciclo) y la angular (indica la cantidad de iteraciones y por ende el aumento en tiempo y coste del proyecto).

Como se ve en la figura 2.1, para cada ciclo habrá cuatro fases:

1. *Determinar Objetivos:* Se recogen y especifican los requisitos. Se identifican los riesgos del proyecto y se plantean estrategias alternativas para evitarlos. Un caso especial de esta fase es la planificación inicial, que solo se realiza en al primera iteración.
2. *Análisis del riesgo:* Se lleva a cabo el estudio de las causas de las posibles amenazas y probables eventos no deseados, así como los daños y consecuencias que éstas puedan producir. Se evalúan alternativas.

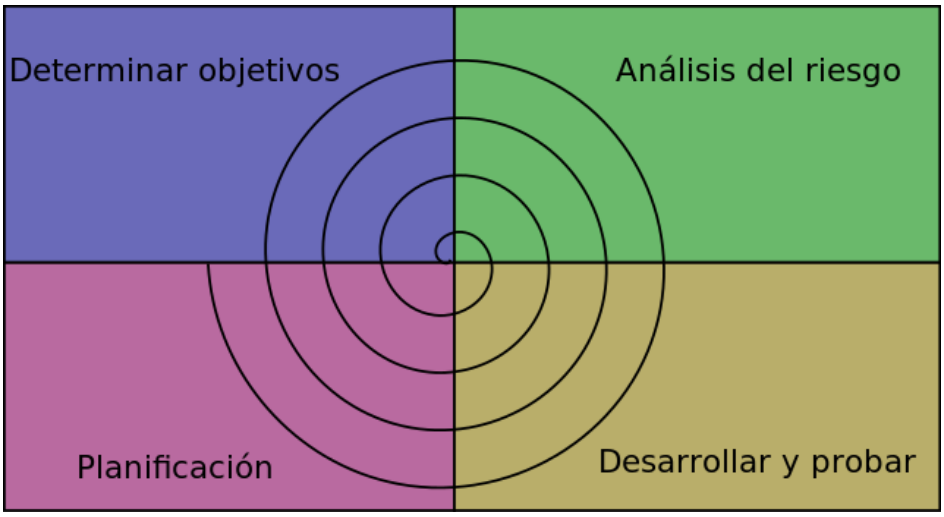


Figura 2.1: Esquema del modelo en espiral

3. *Desarrollar y probar:* Dependiendo del resultado de la evaluación de los riesgos, se elige un modelo adecuado para el desarrollo de entre los existentes. Así si por ejemplo los riesgos en la interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipos evolutivos.

4. *Planificación:* Se presenta el estado y la evolución del proyecto y durante la iteración. A partir de ello se deciden los objetivos a cumplir en la siguiente iteración, que pueden ser desde arreglar errores surgidos de la anterior iteración a incluir nuevas funcionalidades.

## 2.5. Java

Java[15] es un lenguaje de programación orientado a objetos de propósito general, que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "*write once, run anywhere*"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra, ya que la ejecución se realiza a través de una

máquina virtual. Java es, a partir de 2012, uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web, con 9 millones de usuarios-desarrolladores en todo el mundo [14].

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son generalmente compiladas a *bytecode* (código objeto) que puede ejecutarse en cualquier máquina virtual Java (JVM) escrita en código nativo de la máquina en que se ejecuta, por lo que el programa se ejecuta sin importar la arquitectura de la computadora subyacente. Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hilos o *threads*, la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el *bytecode* generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (*Just In Time*).

La compañía Sun desarrolló la implementación de referencia original para los compiladores de Java, máquinas virtuales, y librerías de clases en 1991 y las publicó por primera vez en 1995. A partir de mayo de 2007, en cumplimiento con las especificaciones del Proceso de la Comunidad Java, Sun volvió a licenciar la mayoría de sus tecnologías de Java bajo la Licencia Pública General de GNU. Otros también han desarrollado implementaciones alternas a estas tecnologías de Sun, tales como el Compilador de Java de GNU [10] y el GNU Classpath[11].

En Java el problema de fugas de memoria se evita en gran medida gracias a la recolección de basura automática (o "*automatic garbage collection*"). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java RTS) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste. Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas. Aun así, es posible que se produzcan fugas de memoria si el

código almacena referencias a objetos que ya no son necesarios, pero ya no es el programador quien ha de hacerlo explícitamente como ocurre en C o C++.

Estas facilidades al desarrollo suponen que en lo referente al rendimiento no alcanze las cotas de los ya mencionados C o C++.

## 2.6. Apache Maven

Apache Maven [4] es un software para la gestión de proyectos escritos en Java. Permite compilación, ejecución, ejecución de pruebas, generación de informes, documentación técnica... de una forma centralizada. Facilita la gestión de múltiples proyectos relacionados como un todo en lugar de tener que gestionarlos uno a uno por separado.

Fue creado en 2002 por Jason van Zyl, de Sonatype, y actualmente es desarrollado por la "Apache Software Foundation"[9].

## 2.7. Marcadores en código

A la hora de señalar fragmentos de código para algún framework, es común el uso de marcadores en el mismo código. Estos marcadores pueden ser ignorados por el compilador (por estar escritos en comentarios por ejemplo) e interpretados por otro programa que analice el código fuente, como es el caso de la herramienta Javadoc[16], o pueden desencadenar la ejecución de otro código enlazado a la anotación como es el caso de Spring[18] para tareas como la inyección de dependencias.

## 2.8. XML

Lenguaje de Marcado eXtensible ("EXtensible Markup Language") o XML es un lenguaje de marcado desarrollado por el World Wide Web Consortium (W3C)

utilizado para almacenar datos en forma legible para las personas. Deriva del lenguaje SGML (Standard Generalized Markup Language) [1] y permite estructurar documentos grandes mediante la definición de una gramática específica.

XML se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas, permitiendo compartir la información de una manera segura, fiable y fácil.

La estructura fundamental de un documento XML se basa en un árbol de trozos de información. Estos trozos se llaman elementos y se los señala mediante etiquetas, que consisten en marcas hechas en el documento señalando una porción de éste como un elemento, el cual puede contener más elementos, caracteres o ambos, o bien ser un elemento vacío. Los elementos también pueden tener atributos, que son una manera de incorporar características o propiedades a los elementos de un documento. Las etiquetas tienen la forma `<nombre atributo="valor" />` o `<nombre atributo="valor"> elemento </nombre>`, donde `nombre` es el nombre del elemento que se está señalando.

El prólogo de un documento XML contiene una declaración XML, una declaración de tipo de documento y uno o más comentarios e instrucciones de procesamiento. Ej: `<?xml version="1.0" encoding="UTF-8"?>`.

También se permite el uso de comentarios a modo informativo para el programador que han de ser ignorados por el procesador. Los comentarios en XML tienen el siguiente formato: `<!--- comentario --->`.

Un documento se considera "bien formado" si su estructura sintáctica básica está correctamente escrita según el formato XML. Ahora bien, cada aplicación de XML (es decir, cada lenguaje definido con esta tecnología), necesitará especificar cuál es exactamente la relación que debe verificarse entre los distintos elementos presentes en el documento, lo cual se define en un documento externo (expresada como DTD —Document Type Definition, 'Definición de Tipo de Documento'— o como XSchema). Crear una definición equivale a crear un nuevo lenguaje de marcado, para una aplicación específica.

Entre sus ventajas encontramos que se trata de un lenguaje extensible en el que se pueden seguir añadiendo etiquetas a una vez ya ha sido puesto en producción, el analizador es un componente estándar y por lo tanto no es necesario crear uno específico para cada lenguaje XML definido y tiene una buena compatibilidad entre aplicaciones al ser fácil de entender su estructura y contenido por parte de un tercero.

De sus inconvenientes destaca el hecho de que, al tener una estructura arbórea inherente, puede necesitar mucha memoria para tenerlo cargado en memoria y leerlo.

## 2.9. Generación automática de código

En la actualidad son cada vez más comunes las herramientas que generan código de manera automática, ayudando al desarrollador desde auto-completando el nombre de una variable hasta escribiendo fragmentos completos de código, como pueda ser una herramienta de generación de código a partir de un diagrama UML.

Dichas herramientas suponen una gran ayuda, pues no solo ahorran horas de trabajo (generalmente en tareas repetitivas y monótonas como escribir en una clase sus atributos y el conjunto genérico de métodos para acceder a ellos), sino que además se reducen las posibilidades de introducir errores en dichos fragmentos de código, en comparación con la implementación manual.

## 2.10. Sistema de nombrado

A lo largo del proyecto se han adoptado diversas convenciones a la hora de nombrar los elementos que lo componen. En esta sección se recogen dichas convenciones.

**Objetivos Objetivo-XX** Donde XX es un código numérico correlativo.

**Requisitos R[M|A][F|NF]-XX:** Donde R es de requisito, M de mínimo, A de adicional, F de funcional, NF de no funcional y XX es un código numérico correlativo de dos dígitos para diferenciar los requisitos que sean de la misma categoría y mismo tipo. Ejemplo "RMF-01" para el requisito mínimo funcional número 1.

**Pruebas P[F|NF][E|EST]-X:** Donde P es de prueba, F de funcional, NF de no funcional, E de estado, EST de estrategia y X es un código número correlativo dentro de la misma categoría para el mismo patrón. Ejemplo "PNFEST-1" para la prueba no funcional del patrón estado número 1.

**Riesgos R[E|EST]-XX:** Donde R es de Riesgo, E de estado, EST de estrategia y XX es un código numérico correlativo de dos dígitos que diferencia los riesgos dentro de los pertenecientes al mismo patrón. Ejemplo "RE-02" para el riego numero 2 identificado para la iteración del patrón estado.

**Tests del sistema itX_ftYY_ZZ_test:** Donde X es el número de la iteración donde se ha implementado, ft es de "*functional test*", YY es un código numérico correlativo dentro de la iteración y ZZ es un nombre adecuado a aquello que se esté probando. Además, **itX_ftYY** supone una clave única, y por ello en este documento se identificarán las pruebas solo por ella, sin necesidad de añadir el resto del nombre, pues éste puede llegar a ser bastante largo. Ejemplo "it1_ft12_test_name_example_test" para la prueba número 12 de la primera iteración.

En el código del proyecto, se ha utilizado el inglés para dar nombre a las clases, métodos y atributos, así como para escribir los comentarios, diseñar los marcadores y la representación intermedia. Con esto se pretende aumentar el número de desarrolladores potenciales.

Las diferentes implementaciones de interfaces comunes a cada patrón se han nombrado mediante el nombre del patrón seguido de "`Pattern`" y terminado en el nombre de la interfaz en sí. Por lo tanto, la clase que implemente una interfaz "`Parser`" para el patrón Estado se llama "`StatePatternParser`".

A las pruebas resultantes de la ejecución del sistema se las ha nombrado siguiendo la convención "testXXX_YYY ()" siendo XXX un código numérico

correlativo e YYY un nombre adecuado a la prueba concreta con "_" en lugar de espacios.

# Capítulo 3

# Requisitos

## Índice general



En este capítulo se van a detallar los requisitos del sistema a construir. Los requisitos se han clasificado de dos maneras: por un lado según si son funcionales o no funcionales, y por otro según su prioridad de implementación o importancia.

Puesto que desde el punto de vista de la planificación es más importante la prioridad, en este documento se van a presentar ordenados por dicha relevancia, y dentro de un mismo bloque de importancia, en base a si son funcionales o no funcionales.

Los requisitos se han nombrado siguiendo las directrices definidas en la sección 2.10.

## 3.1. Mínimos

Los requisitos mínimos son aquellos que se considera debe cumplir el software resultante en la fecha de la entrega.

### 3.1.1. Funcionales

Podemos definir los requisitos funcionales como el conjunto de servicios que un sistema software debe proveer, los resultados concretos que debe proporcionar como salida ante entradas particulares. [**?** ]

En otras palabras, los requisitos funcionales son la descripción a alto nivel de qué ha de hacer el sistema o aplicación.

**RMF-01** Recorrido recursivo de un directorio en busca de código Java.

**RMF-02** Análisis del código Java en busca de la información pertinente que defina un patrón soportado.

**RMF-03** Interpretación de la información del patrón concreto detectado.

**RMF-04** Representación del patrón detectado en un formato intermedio, conocido y estándar, que sirva de entrada al módulo de escritura de pruebas.

**RMF-05** Lectura de la representación intermedia por parte del módulo de escritura de pruebas.

**RMF-06** Generación de código de pruebas Java con JUnit a partir de la información contenida en la representación intermedia.

**RMF-07** Soporte para el patrón Estado.

**RMF-08** Soporte para el patrón Estrategia.

**RMF-09** Soporte para el patrón Observador.

RMF-01, RMF-02 son independientes del patrón en sí a analizar y suponen un comportamiento común que solo tendrá que implementarse una vez.

RMF-03, RMF-04, RMF-05, RMF-06 son dependientes del patrón en sí, y tendrán que cumplirse todas en las respectivas implementaciones que realicen los requisitos RMF-07, RMF-08 y RMF-09, así como para el resto de patrones a los que se dé soporte fuera del mínimo.

Estos requisitos suponen la base de las funcionalidades del sistema.

### 3.1.2. No funcionales

Los requisitos no funcionales son caracterizaciones del comportamiento de un sistema software. Se incluyen aquí restricciones de temporización, de consumo de recursos, etc. [20]

Dicho de otra forma, son propiedades se requieren del sistema, pero que no constituyen un servicio que el sistema ofrece al usuario, sino que describen cómo debe prestarse dicho servicio en términos de cualitativos.

**RMNF-01** Legibilidad del código fuente.

**RMNF-02** Facilidad de ampliación de funcionalidades.

**RMNF-03** Minimizar el acceso a disco (escalabilidad).

**RMNF-04** Simplicidad de cara al usuario final en el sistema de marcado de código.

**RMNF-05** Limpieza y legibilidad del código generado.

Para asegurar RMNF-02, es necesario que de entre los requisitos RMF-07, RMF-08 y RMF-09 se realice al menos la implementación de dos de ellos con el fin de verificar si el esfuerzo necesario para ampliar el sistema es realmente más bajo que el invertido en la primera iteración (lo cual deberá ser cuantificado).

## 3.2. Adicionales

En contrapunto a los requisitos mínimos, los requisitos catalogados como adicionales son aquéllos que no se consideran necesarios, pero sí deseables o útiles de cara al sistema final, como por ejemplo que permita devolver diferentes mensajes de error en función del idioma configurado. [20]

### 3.2.1. Funcionales

**RAF-01** Soporte para internacionalización.

**RAF-02** Mensajes de error en Multi-5 (Inglés, Español, Alemán, Francés e Italiano).

### 3.2.2. No funcionales

**RANF-01** Procesado paralelo. Que cada patrón detectado se procese en un hilo diferente (mediante un Maestro-Esclavo [3]).

# Capítulo 4

# Planificación inicial

## Índice general



En este capítulo se va a presentar la planificación general del proyecto. Además, dado el ciclo de vida elegido para gestionar el desarrollo, en el cual la planificación de una iteración se realiza al final de la iteración anterior, se va a realizar la planificación de la primera iteración aquí, considerando la captura de requisitos y planificación general como una iteración 0.

Debido a diversos motivos, la disponibilidad horaria del personal no es constante, lo que impidió una planificación temporal estable. Por ello, se ha realizado una estimación en términos de esfuerzo sin prestar especial atención a las fechas, exceptuando las de inicio y fin, así como a la fecha deseada de finalización de cada iteración.

La estimación del esfuerzo (valorado como Alto, Medio o Bajo) para los requisitos analizados en el capítulo 3 se han dejado para el momento de la planificación en la que se van a realizar. Esto es debido a que durante la primera iteración el esfuerzo que requerirán será mayor (al partir de cero) que en las sucesivas iteraciones, donde ya se tendrá un referente, por lo que la estimación de la primera iteración es superior a la del resto de iteraciones.

| | Nombre | Duración | Inicio | Fin anticipado |
|---|---|---|---|---|
| 1 | **TGR - SW** | **208 days?** | **18/07/15 8:00** | **10/02/16 17:00** |
| 2 | Recogida de requisitos | 1 day? | 18/07/15 8:00 | 18/07/15 17:00 |
| 3 | Iteración 1 - Patrón Estado | 135 days | 19/07/15 8:00 | 30/11/15 17:00 |
| 4 | Iteración 2 - Patrón Estrategia | 31 days? | 1/12/15 8:00 | 31/12/15 17:00 |
| 5 | Iteración 3 - Patrón Observador | 31 days? | 1/01/16 8:00 | 31/01/16 17:00 |
| 6 | Memoria: Ajustes finales | 10 days? | 1/02/16 8:00 | 10/02/16 17:00 |

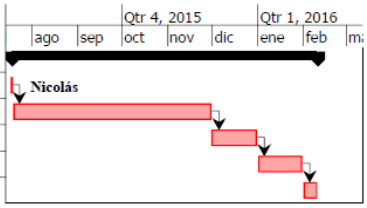


Figura 4.1: Planificación general: Diagrama de Gantt

## 4.1. Planificación general

La planificación inicial del proyecto consta de su inicio el 17 de julio de 2015 y su finalización el 10 de febrero de 2016. Se esperaba dar por terminada la primera iteración el día 30 de Noviembre del 2015, la segunda iteración para el 31 de Diciembre del 2015 y la tercera iteración para el 31 de Enero del 2016, dejando así 10 días para terminar toda la documentación y realizar una revisión completa del proyecto.

Se estima un esfuerzo total (análisis, diseño, implementación, pruebas y documentación) de aproximadamente 300 horas × persona. Se considera que el desvío aceptable debiera ser un máximo de 20 horas × persona arriba o abajo. El coste final (incluyendo equipo, personal, instalaciones, impuestos y ganancias) en horas establecido es de 30€/hora, tarifa final de cara al cliente bastante estándar para un proyecto con un analista/programador junior como recurso, con lo que, en caso de tener que ofertárselo a un cliente, incurriría en un coste total del proyecto de 9.000€.

## 4.2. Iteración 0: Planificación de la iteración 1 - Patrón *State*

La primera iteración precisa que se desarrolle la base del sistema, formada por los requisitos de *recorrido recursivo de un directorio* (RMF-01) y de *análisis del código en busca de la información pertinente* (RMF-02) (que solo serán implementados una vez), junto con una primera aproximación al resto del sistema (que

deberá ser ampliada para cada patrón incorporado en sucesivas iteraciones) mediante los requisitos de: *interpretación de la información del patrón* (RMF-03), *representación del patrón en un formato intermedio* (RMF-04), *lectura de la representación intermedia por parte del módulo de escritura de pruebas* (RMF-05) y *generación de código de pruebas* (RMF-06). Durante esta primera iteración se ha decidido realizar el requisito de *Soporte para el patrón Estado* (RMF-07). Estos requisitos y sus estimaciones de esfuerzo para esta iteración pueden observarse en la tabla 4.1.

| **Requisito** | **Descripción** | **Esfuerzo** |
|---|---|---|
| RMF-01 | Recorrido recursivo de un directorio | Bajo |
| RMF-02 | Análisis del código en busca de la información pertinente | Alto |
| RMF-03 | Interpretación de la información del patrón | Alto |
| RMF-04 | Representación del patrón en un formato intermedio | Medio |
| RMF-05 | Lectura de la representación intermedia por parte del módulo de escritura de pruebas | Medio |
| RMF-06 | Generación de código de pruebas | Alto |
| RMF-07 | Soporte para el patrón Estado | Medio |
| RMNF-01 | Legibilidad del código fuente | Bajo |
| RMNF-02 | Facilidad de ampliación de funcionalidades | Medio |
| RMNF-03 | Minimizar el acceso a disco | Bajo |
| RMNF-04 | Simplicidad de cara al usuario final en el sistema de marcado de código | Medio |
| RMNF-05 | Limpieza y legibilidad del código generado | Bajo |
| RAF-01 | Soporte para internacionalización | Bajo |

Cuadro 4.1: Resumen de requisitos y estimaciónes de la primera iteración

De los requisitos funcionales adicionales se ha decidido implementar desde el principio el requisito de *soporte a la internacionalización* (RAF-01), pues el esfuerzo para implementarlo en una etapa temprana es casi despreciable, mientras que en una etapa más madura sería medio o incluso alto, en función de la cantidad de mensajes que deban internacionalizarse. Además, simplifica el mantenimiento de las pruebas unitarias al cotejar el mensaje devuelto por el sistema con el relacionado al identificador, de forma que modificar los mensajes en sí no influye en las pruebas.

Con el fin de conseguir cumplimiento del requisito de *facilidad de ampliación de funcionalidades* (RMNF-02), se ha de conseguir que los requisitos comunes a cada patrón (RMF-03, RMF-04, RMF-05 y RMF-06) implementados en la primera iteración puedan usarse como plantilla o guía para la ampliación del sistema con el soporte para más patrones. Esto repercute en la primera iteración haciendo especialmente necesario un diseño robusto y reutilizable.

El cumplimiento del requisito de *Minimizar el acceso a disco* (RMNF-03) recae sobre los requisitos funcionales RMF-01 y RMF-02 (sobre todo este segundo) en los cuales se accede a los ficheros de forma que no sean necesarias posteriores lecturas.

Debido a la necesidad de desarrollar completamente los requisitos RMF-01 y RMF-02, así como por ser la primera vez que se implementan los requisitos RMF-03, RMF-04, RMF-05 y RMF-06 (y por tanto no existir una estructura por la que guiarse), esta iteración se estimó que necesitaría un esfuerzo muy alto en comparación con el resto de iteraciones.

La planificación desglosada en tareas puede verse en el diagrama de Gantt de la figura 4.2.

| | Nombre | Trabajo | Predecesores |
|---|---|---|---|
| 1 | Inicio Iteración 1 - Patrón Estado | 0 horas | |
| 2 | **Iteración 1 - Patrón Estado** | **198 horas** | **1** |
| 3 | Iteración 1 - Análisis | 5 horas | 1 |
| 4 | **Subsistema 1 - Analyzer** | **77 horas** | **3** |
| 5 | Diseño | 15 horas | 3 |
| 6 | **Implementación** | **29 horas** | **5** |
| 7 | Etapa 1 - Crowler | 6 horas | 5 |
| 8 | Etapa 2 - Agrupamiento de ficheros | 3 horas | 12 |
| 9 | Etapa 3 - Parser | 10 horas | 13 |
| 10 | Etapa 3 - Writer | 10 horas | 14 |
| 11 | **Validación** | **33 horas** | **7** |
| 12 | Etapa 1 - Crowler | 8 horas | 7 |
| 13 | Etapa 2 - Agrupamiento de ficheros | 5 horas | 8 |
| 14 | Etapa 3 - Parser | 10 horas | 9 |
| 15 | Etapa 3 - Writer | 10 horas | 10 |
| 16 | **Subsistema 2 - TestMaker** | **60 horas** | **4** |
| 17 | Diseño | 20 horas | 4 |
| 18 | **Implementación** | **20 horas** | **17** |
| 19 | Builder desde XML | 5 horas | 17 |
| 20 | PFE-1 - Transiciones | 5 horas | 24 |
| 21 | PFE-2 - Camino | 5 horas | 25 |
| 22 | PNFE-1 - Acceso concurrente | 5 horas | 26 |
| 23 | **Validación** | **20 horas** | **19** |
| 24 | Builder desde XML | 5 horas | 19 |
| 25 | PFE-1 - Transiciones | 5 horas | 20 |
| 26 | PFE-2 - Camino | 5 horas | 21 |
| 27 | PNFE-1 - Acceso concurrente | 5 horas | 22 |
| 28 | Pruebas de integración y Validación | 15 horas | 16 |
| 29 | Redacción de memoria | 30 horas | 28FF;15S |
| 30 | **Tareas adicionales** | **11 horas** | **6SS** |
| 31 | Implementaciónnternacionalización | 3 horas | 6SS |
| 32 | Estudio librerías de comparación de XML | 4 horas | 14SS |
| 33 | Migración a Apache Maven | 4 horas | 4 |




Figura 4.2: Planificación de la primera iteración: Diagrama de Gantt

# Capítulo 5

# Análisis y Diseño

## Índice general



En este capítulo se describen, para cada iteración, las fases del ciclo de vida en cascada de "Determinación de objetivos" (análisis del patrón, sus pruebas funcionales y no funcionales y qué información es necesaria) y "Análisis de riesgo". En caso de que el análisis de riesgos dé como conclusión que no es viable el soporte de un patrón por este sistema, se da por terminada esa iteración.

El software se ha diseñado como un proyecto Maven (`TestGenerator`) compuesto de varios subproyectos Maven siguiendo la estructura descrita en la figura 5.1. En ella se puede observar un proyecto para el módulo de análisis (`CodeAnalyzer`) de código, otro para el módulo de generación de código (`TestMaker`), uno que englobe ambos proporcionando una interfaz externa del software en su conjunto (`PatternTestGenerator`), uno para las pruebas de integración y validacion (`IntegrationTesting`) y otro con implementaciones de ejemplo para las pruebas (`PatternImplementationExamples`).

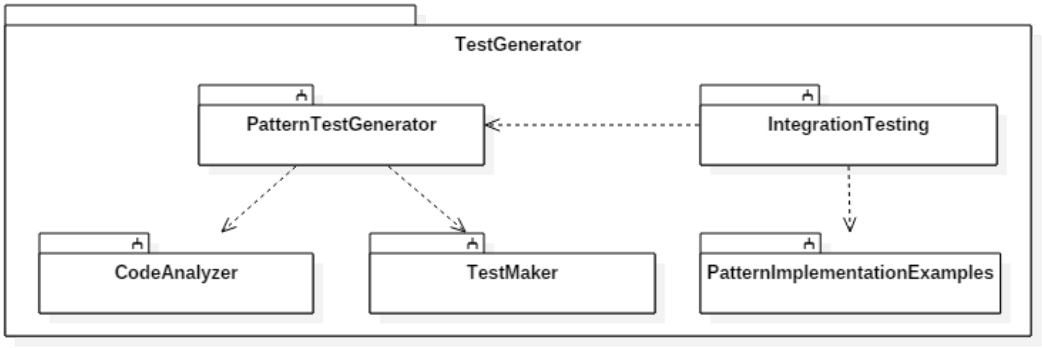


Figura 5.1: Estructura de proyectos y subproyectos

## 5.1. Iteración 1: *Estado*

Durante esta iteración, se ha analizado del patrón Estado y diseñado tanto un conjunto de pruebas (o esqueleto de pruebas) como una representación abstracta en formato XML. Se ha diseñado también un conjunto de marcadores que permitan extraer toda la información necesaria para realizar las pruebas.

Con eso, se diseñó la primera versión del módulo software que se encarga de recorrer recursivamente una estructura de directorios abriendo ficheros de código fuente Java y analizándolos en busca de los marcadores definidos, con los cuales componer una estructura interna con los datos del patrón, la cual se parsea a un árbol DOM que se guardada como XML.

A partir de la definición de las pruebas y de la representación abstracta, se diseñó el segundo módulo software que transforma la información del XML en uno o más ficheros con las pruebas unitarias.

Por la necesidad añadida de diseñar de cero la estructura básica de los dos módulos (recorrido y análisis de ficheros, así como generación de código de pruebas), se esperaba que ésta fuera la iteración que más esfuerzo conllevase.

Para el desarrollo del sistema se decidió usar Java EE con JUnit debido a la familiaridad con el lenguaje y entornos de desarrollo, así como por la existencia

de gran cantidad de librerías estables (como son las librerías de manejo de XML) que simplifican y agilizan el desarrollo.

### 5.1.1. Análisis del patrón

El propósito del patrón Estado es permitir que un objeto pueda cambiar de comportamiento sin cambiar de clase (y, por tanto, manteniendo la integridad referencial con respecto a otros objetos que actúen como clientes). Para ello, separa el comportamiento que depende del estado del que no, y el primero se externaliza a una jerarquía de estados que únicamente el objeto conoce, y con la que se mantiene una relación de asociación que gestiona los propios objetos estado.

Un objeto Contexto con un atributo que referencia a un objeto Estado e invoca los métodos de este para llevar a cabo sus casos de uso. Con ello se genera un comportamiento dependiente de la lógica del EstadoConcreto al que hace referencia el atributo, el cual ademas podrá transitar cambiando dicho EstadoConcreto a otro distinto (con lo que cambiaría el comportamiento en la siguiente llamada a un método que dependa del estado) o dejando el mismo que estaba. Es importante que el contexto ofrezca una forma de modificar su estado interno por parte del EstadoConcreto, generalmente teniendo un método set y pasándose a sí mismo como parámetro en las acciones del estado.

Un ejemplo de ejecución sería: se llama a un método del contexto, el cual en algún momento de su ejecución delega en un método del EstadoConcreto que tenga instanciado en ese momento independientemente de cual sea, por lo que en función del EstadoConcreto, se conseguirá un comportamiento u otro.

Esta estructura puede observarse en la figura 5.2.

La forma más sencilla de este patrón puede verse como un autómata determinista en el cual el alfabeto de entrada son las posibles acciones, el alfabeto de salida no es controlable a nivel general (resultado directo de la acción), los estados (EstadoConcreto) son las implementaciones del EstadoAbstracto, el estado

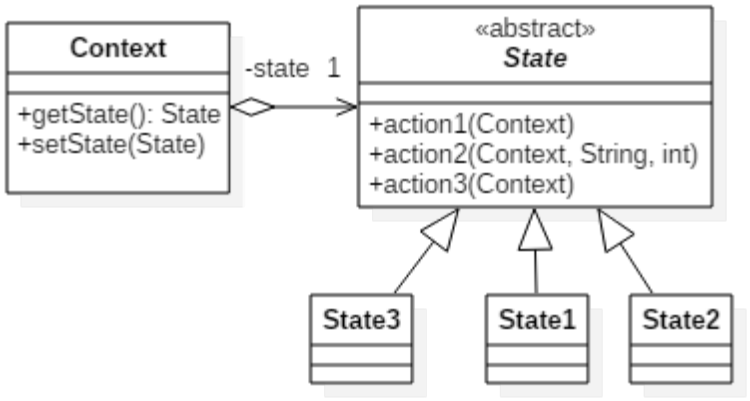


Figura 5.2: Diagrama de clases general del patrón Estado.

inicial y los estados finales están definidos y las producciones son las transiciones de un estado con una acción a otro estado.

No obstante, esto puede complicarse añadiendo factores como:

- No determinismo en función de una llamada a un sistema externo, o del entorno de ejecución.
- No determinismo en función del valor de los parámetros que reciba la acción.

### 5.1.2. Análisis de pruebas funcionales sobre el patrón

**PFE-1 - Transiciones**

Las pruebas más básicas sobre este patrón son comprobar que, dado cualquier estado y cualquier acción, se transita al estado correcto (entendiendo como transición al mismo estado cuando no se transita).

Para ello, suponiendo determinismo no dependiente de los parámetros ni sistemas expertos sino solo de las acciones y el estado actual, lo más sencillo y efectivo es invocar cada acción desde cada estado y comprobar a dónde transita. Los resultados deberán ser la transición a un EstadoConcreto para aquellas transiciones

definidas, y la transición al mismo estado para aquellas no definidas. Cualquier comportamiento que no se ajuste a ello, es considerado un error.

**PFE-2 - Camino**

Otra prueba (que requiere más información) es partiendo del estado inicial y con una cadena de acciones válida, comprobar que se llega a un estado final. De la misma manera partiendo del estado inicial y con una cadena de acciones no válida, que no se llegue a un estado final. Existen sistemas con estado que no están pensados para "morir" nunca mientras sigue el sistema en ejecución, en cuyo caso el patrón no tiene estado final y esta prueba no tiene sentido.

### 5.1.3. Análisis de pruebas no funcionales sobre el patrón

**PNFE-1 - Acceso concurrente**

Una prueba no funcional es la comprobación de la respuesta del patrón ante el acceso concurrente. Se trata de acceder al mismo Contexto desde diferentes hilos, lo cual impide saber en qué orden llegarán las peticiones aunque sepamos en qué orden se generan en cada hilo.

Para comprobar la robustez ante este comportamiento, se pueden ejecutar las mismas operaciones en diferente orden, y ver si llevan al mismo estado, pero no en todos los dominios eso es el comportamiento correcto.

Puesto que el resultado de esta prueba se debe interpretar de forma diferente en diferentes dominios, deberá especificarse el comportamiento deseado, siendo el por defecto que el estado final no deba ser dependiente del orden de las interacciones. En caso de que el estado final sí dependa del orden de las acciones, el resultado de esta prueba debe ser interpretado al revés.

### 5.1.4. Información necesaria (mínimo)

En este apartado se expone la información mínima que es necesario conocer para poder llevar a cabo PFE-1, que son las pruebas básicas del patrón, en su forma más sencilla (transiciones sólo dependientes de la acción invocada).

A partir de aquí nos referiremos al nombre del paquete seguido del nombre de la clase como nombre completo, el cual se corresponde con el "*domain name*" en terminología Java.

- Identificador del patrón Estado dentro del proyecto (por si hay más de uno en la implementación del sistema o componente).
- Nombre completo de la clase contexto contenedora del patrón Estado e identificador del patrón Estado dentro del proyecto. Es necesario pues los métodos de los estados reciben el objeto contexto para transitar de ser preciso.
- Nombre completo de la clase abstracta/interfaz EstadoAbstracto.
- Nombre completo de cada clase EstadoConcreto.
- En cada EstadoConcreto, si una acción implica una transición, indicar a qué estado transita. De no indicarse se asume que no transita a otro estado y se mantiene el actual.
- Ejemplo de instanciación de toda clase implicada que no se instancia mediante un constructor vacío.

Puesto que para la realización de las pruebas es necesario poder saber el EstadoConcreto que contiene el contexto en un momento dado, se ha impuesto la pre-condición de que el contexto contenga un método "`+getStateHeader (): StateHeader`" siendo *StateHeader* el nombre de la clase correspondiente al EstadoAbstracto.

También es necesario de cara a las pruebas poder inyectar un EstadoConcreto en el contexto para poder controlar el estado de partida, pero por motivos del

patrón en sí ya ha de haber ya un *setter* del estado, "`+setStateHeader (state: StateHeader): void`" siendo *StateHeader* el nombre de la clase correspondiente al EstadoAbstracto, para que al ejecutar una acción del estado actual, este pueda modificar el estado que contiene el contexto que se pasa como parámetro (provocando una transición).

En el desarrollo de todas las pruebas, existe el problema de las dependencias con los paquetes que se necesiten importar en el código. Es fácil obtener los datos para incluir aquellas referentes a las clases implicadas, pero no tanto, por ejemplo, si en un constructor definido por el usuario se instancia una clase que no forma parte del patrón en sí (por ejemplo, un `GregorianCalendar`). En ese caso para poder incluir sus dependencias en las pruebas resultantes, se debería o bien analizar por completo el código del constructor, localizar las dependencias y buscarlas entre aquellas importadas por la clase analizada, o bien incluir todas las dependencias usadas en todas las clases participantes en el patrón.

Ante este problema, puesto que no supone un gran esfuerzo al usuario y muchos IDEs lo hacen automáticamente con las dependencias más comunes, se ha decidido que sea el usuario quien, de ser necesarias, las introduzca en el código resultante, dejando como mejora futura el que sea el propio sistema quien los identifique y añada.

### 5.1.5. Información adicional

La información necesaria para ejecutar las pruebas sobre versiones más complejas de este patrón es:

- En cada EstadoConcreto, si una acción implica una transición en función de sus parámetros, con que valores de los parámetros se deberá comprobar.
- En cada EstadoConcreto, si una acción implica una transición en función de factores externos y de qué manera. Así, aunque no puedan probarse estas transiciones, el sistema sabrá que no tiene control sobre ellas y no las considerará errores.

- Ejemplos de una sucesión de acciones que lleven la máquina de estados desde el estado inicial a un estado final, así como una sucesión que deba llevar a un estado no final.

Si es determinista o no, se deduce de la definición de sus transiciones, de forma que si cada acción de un EstadoConcreto solo tiene una transición posible, será entendido como determinista.

### 5.1.6. Análisis de riesgo

Se le ha asignado un identificador a cada riesgo se han nombrado siguiendo las directrices definidas en la sección 2.10.

**RE-01** Tener que rehacer trabajo por descubrir algún problema o una forma más efectiva de hacer algo (ya implementado) a la hora de construir una base genérica que aproveche todas las similaridades en la forma de trabajar para analizar los patrones.

- Riesgo: alto
- Repercusión: alta
- Acción preventiva: realizar un diseño exhaustivo
- Acción correctiva: rehacer la parte afectada

**RE-02** No encontrar una forma factible de recopilar los datos del patrón para PFE-1.

- Riesgo: bajo
- Repercusión: alta
- Acción preventiva: NA
- Acción correctiva: abortar patrón por ser la prueba básica

**RE-03** No encontrar una forma factible de recopilar los datos del patrón para PFE-2.

- Riesgo: medio
- Repercusión: media
- Acción preventiva: NA
- Acción correctiva: dejar un espacio en el código final, señalado, para que el usuario complete con la secuencia de acciones

**RE-04** No encontrar una forma factible de recopilar los datos del patrón para PNFE-1.

- Riesgo: medio
- Repercusión: media
- Acción preventiva: NA
- Acción correctiva: no implementar PNFE-1

**RE-05** No encontrar una forma factible de implementar PFE-1.

- Riesgo: bajo
- Repercusión: media
- Acción preventiva: NA
- Acción correctiva: abortar patrón por ser la prueba básica

**RE-06** No encontrar una forma factible de implementar PFE-2.

- Riesgo: bajo
- Repercusión: media
- Acción preventiva: NA
- Acción correctiva: no implementar PFE-2

**RE-07** No encontrar una forma factible de implementar PNFE-1.

- Riesgo: bajo
- Repercusión: media
- Acción preventiva: NA

| **Riesgo / Repercusión** | **Alta** | **Media** | **Baja** |
|---|---|---|---|
| **Alto** | 1 | 0 | 0 |
| **Medio** | 0 | 2 | 0 |
| **Bajo** | 1 | 3 | 0 |

Cuadro 5.1: Iteración 1 Recuento de riesgos

- Acción correctiva: no implementar PNFE-1

Resumen:

A pesar de existir un riesgo alto con un gran impacto (RE-01), se ha decidido seguir adelante con el diseño, implementación y pruebas de esta iteración, pues dicho riesgo se corresponde a la base de la aplicación, que es imprescindible e inevitable.

Para prevenir que se dé el riesgo RE-01, se le ha dado especial importancia a la fase de diseño. En caso de que el riesgo se produzca, salvo que su repercusión sea muy pequeña a corto y largo plazo, se deberá volver al diseño y refinarlo reiteradamente hasta que se solvente el problema descubierto.

### 5.1.7. Definición del lenguaje XML

Se ha decidido usar XML en la representación intermedia debido a la capacidad de personalización que permite, la existencia de multitud de librerías para trabajar con él y la facilidad para manipularlo (por ejemplo, realizando recorridos de árbol).

La parte del lenguaje XML referente al patrón Estado se muestra en la figura 5.3 y sigue la siguiente definición:

Dentro de una etiqueta *statePattern* se define:

**Id** identificador de la implementación concreta para el sistema de generación de tests..

**orderdependent OR nonorderdependent** indica si es o no determinista con respecto al orden de ejecución de sus acciones.

**generalstate** contiene la información a cerca del EstadoAbstracto del que heredan los estados concretos. Se indica su *package* y *classname*.

**actions** tiene un *action* por cada método (público) del EstadoGenérico (uno o varios).

**context** indica la clase que contiene al patrón Estado. Contiene su *package*, *classname* y *builder* (este último es opcional).

**states** contiene un *package* y el conjunto de *state* que cuelgan de este.

**state** representa un EstadoConcreto con su *classname*, *initialstate*, *finalstate*, *builder* y un *transitions*

**transitions** tiene cero o varias *transition*.

**transition** puede depender de cero o más parámetros. Un atributo *position* indica el orden de los parámetros en la firma del método. Si solo recibe el contexto, no hace falta especificarlo. Si recibe más de un parámetro, en la posición del contexto se pondrá "context". La posición empieza a contar en 0.

**package** es la ubicación de la clase dentro de la estructura del proyecto.

**classname** es el nombre de la clase.

**builder** es el fragmento de código que se usará para instanciar la clase. Si no se especifica, se entiende constructor vacío.

**initialState** indica que el estado es el inicial. Ha de haber uno y solo uno. Ha de ser un EstadoConcreto.

**finalState** indica que el estado es final. Puede haber varios, uno o ninguno. Ha de ser un EstadoConcreto.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<patterns>
    <statePattern>
        <id>identificator</id>
        [<orderdependent /> | <nonorderdependent />]
        <generalstate>
            <package>testExamples.parser.state.correct</package>
            <classname>StateHeader</classname>
        </generalstate>
        <actions>
            <action>action name</action>
        </actions>
        <context>
        <package>testExamples.parser.state.correct</package>
            <classname>StateContext</classname>
            [<builder>new StateContext (new StateConcrete())</builder>]
        </context>
        <states>
            <package>testExamples.parser.state.correct</package>
            <state>
                <classname>StateConcrete</classname>
                [<initialstate/>]
                [<finalstate/>]
                [<builder>new StateConcrete()</builder>]
                [<transitions>
                    <transition>
                        <action>action1</action>
                        <destiny>State1Concrete2</destiny>
                    </transition>
                    <transition>
                        <action>action2</action>
                        <destiny>State1Concrete2</destiny>
                        <parameters>
                            <parameter position="0">"test"</parameter>
                            <parameter position="1">context</parameter>
                            <parameter position="2">5</parameter>
                        </parameters>
                    </transition>
                </transitions>]
            </state>
            <state> ... </state>
        </states>
    </statePattern>
</patterns>
```

Figura 5.3: Ejemplo del lenguaje XML para el patrón Estado

### 5.1.8. Diseño de marcadores

Se ha decidido utilizar marcadores dentro de comentarios encima de las líneas de interés del código (cabeceras de clases o métodos) frente a otras fuentes como podría ser un diagrama UML, debido al amplio uso de marcadores similares en Java (`@override` por ejemplo) y las diferentes herramientas populares entre los desarrolladores que usan este lenguaje (JUnit, Hibernate, Spring, JavaDoc, etc.). Otro motivo de esta decisión es que permite mayor flexibilidad al hacerse las pruebas en función del código concreto, evitando problemas por inconsistencia entre el diseño (diagrama) y la implementación final.

Los marcadores deben estar dentro de un comentario de bloque ( /* */ ). No se permite separar marcadores relacionados en distintos bloques. Que se haga dentro de un comentario cuando java permite añadir comportamiento mediante anotaciones precedidas de "@" se decidió para evitar la interferencia con otras herramientas en caso de que coincidan los nombres. Al hacerse dentro de /* */ se evita interferencia con JavaDoc que lo hace en /** */, y con todos los que marcan sin comentario, como Java o Hibernate. En un bloque marcado no deberá haber nada que no sean los marcadores y sus datos asociados. Un mismo marcador puede dividirse en varias líneas consecutivas mientras no se corten palabras.

Entre el bloque de comentarios anotado y la línea de código de interés no puede haber otra cosa que líneas en blanco, otros comentarios y anotaciones del lenguaje y otras herramientas que empiecen por "@".

Existen varias alternativas sobre que marcador usar: se puede usar el mismo que usa Java y otras herramientas de Java "@"que será más habitual al desarrollador, o uno diferente para una identificación más sencilla como pueda se "$".

Se ha decidido realizarlo con "@"por ser el más usado.

**@pattern State <identificator>**

Identifica una parte de un patrón Estado.

Ha de ir seguido de un *@patternElement*.

*Identificator* es el identificador del patrón y es opcional en casi todos los

casos. En caso de "*@patternElement State*" si no se indica, se tomará del nombre de la clase. En caso de "*@patternElement ConcreteState*" si no se indica, se buscará en el extends o implements. "*@patternElement Context*" deberá llevar siempre identificador.

**@patternElement**

Puede ser de cuatro tipos: *State*, *Context* y *ConcreteState*, que son mutuamente excluyentes, y *Builder*, combinable con las otras tres.

**@patternElement Context**

Identifica la clase Contexto que contiene el estado como atributo y se pasa a los métodos de ése.

Ha de haber uno, y solo uno.

**@patternElement State [orderdep | nonorderdep]**

Identifica la clase como Estado abstracto.

Orderdep o nonorderdep (dependiente o no del orden de las acciones) es opcional. Por omisión es no dependiente.

A efectos de verificación de los datos, los métodos públicos o protegidos de esta clase que reciban el Contexto como parámetro serán considerados acciones del patrón.

**@patternElement ConcreteState [initial] [final]**

Identifica la clase como EstadoConcreto y lo clasifica como inicial y/o final.

Todos los EstadoConcreto han de estar en el mismo "package".

Ha de haber un estado inicial.

Sólo puede haber un estado inicial.

No tiene por qué haber un estado final.

Puede haber más de un estado final.

**@patternElement Builder <builderInvocation>**

Recibe un ejemplo de instanciación de la clase, el cual se usa en las pruebas.

BuilderInvocation se toma como un literal (deberá ir entre < y >). De momento no se permite > dentro del constructor.

Ha de ser una sola sentencia y no ha de terminar con ";".

De no ponerse, se usa un constructor vacío.

Puede ir precedido de cualquiera de los patternElement Context y ConcreteState.

**@patternAction Transition ConcreteState <ParamLine>**
Identifica una acción dentro de la clase EstadoConcreto. Ha de corresponderse con una acción reconocida de la clase EstadoAbstracto.
*ConcreteState* es el nombre de la clase EstadoConcreto a la que se realizará la transición. *ParamLine* es una sucesión de valores separados por comas que serán parseados y usados en el mismo orden al especificado. En la posición donde se pasa el contexto, deberá ponerse "*context*".
Si no se especifica *ParamLine*, se entiende que solo recibe el contexto.

Un ejemplo de marcadores sobre código se muestra en las figuras 5.4 y 5.5.

La información recabada se guarda en la estructura de clases descrita en la figura 5.6.

### 5.1.9. Diseño del módulo de recogida de información

La forma más sencilla es el recorrido recursivo de una estructura de ficheros, a modo de *crowler*, analizando (con un *parser*) los ficheros fuente en busca de los marcadores pertinentes (si sobre la cabecera de la clase no hay uno pasará al siguiente fichero) y guardando la información en una estructura interna, la cual luego se convertirá a la representación XML.

El módulo necesitará como parámetros el directorio raíz donde iniciar el *crowling* y el nombre (preferiblemente absoluto) a ponerle al fichero XML.

Como parte genérica para todos los patrones implementados en sucesivas iteraciones, se ha diseñado un conjunto de clases e interfaces que encapsulen en la medida de lo posible el patrón con el que está trabajando:

**Marker** contiene la información de un marcador. Lo procesa y almacena de forma que no sea necesario otra lectura del fichero.

```
/*
* @pattern State <State>
* @patternElement Context
* @patternElement Builder <new State ()>
*/
public class State .... {

    private State state;

    public Context () {
        ...
    }

    public void setState (State state) {
        this.state = state;
    }

    public State getState () {
        return state;
    }

    ...
}


/*
* @pattern State
* @patternElement State
*/
public abstract State ... {
    ...
    public abstract void action1 (Context context);
    public abstract void action2 (String text, Context context, int value);
    ...
}
```

Figura 5.4: Ejemplo de código con marcadores para el patrón Estado

```
/*
* @pattern State
* @patternElement ConcreteState initial
*/
@Transactional // Se permite el uso de anotaciones de java
public class State1 implements Comparable State ... {

    /*
    * @patternAction Transition State2 <"test", context, 5>
    */
    /**
    * Aquí podría ir el javadoc u otros comentarios
    */
    @override
    public void action2 (String text, Context context, int value) {
             ...
    }
}

/*
* @pattern State <StateHeader>
* @patternElement ConcreteState final
* @patternElement Builder <new State2 ()>
*/
public class State2 implements Comparable State ... {
    ...
}
```

Figura 5.5: Ejemplo de código con marcadores para el patrón Estado

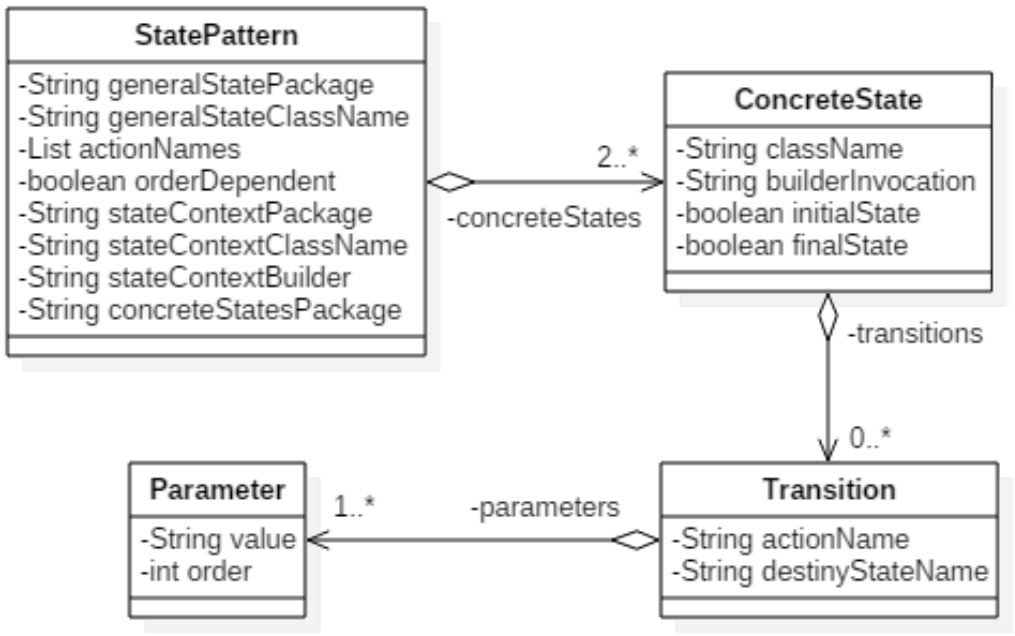


Figura 5.6: Diagrama de clases de la representación interna del patrón Estado

**MarkedBlockComment** representa a un comentario de bloque con marcadores, y contiene tanto el comentario de bloque en sí, como la lista de *Marker* producto de procesarlo.

**ParsedFile** se corresponde a un fichero procesado. Contiene al propio fichero (por si fuera necesario en algún caso concreto), una lista con sus dependencias y una lista de *MarkedBlockComment* con la información de sus marcadores.

**PatternParsedFiles** es la estructura que guarda la información de entrada de cada instancia de patrón reconocido por el sistema. Contiene que patrón es, su identificador, una lista de *ParsedFile* con los ficheros definidos con ese identificador y el *ParsedFile* de la clase "cabecera" (separado para identificarlo más rápidamente ya que su procesado suele ser diferente).

**Pattern** es una clase abstracta la cual extienden las clases representativas de los patrones. Contiene un método abstracto "`+getWriter():PatternWriter`" de forma que cada patrón sea el encargado de saber que clase lo traduce a

XML. Contiene a su vez un método "`+validate(): String`" que devuelve un *String* vacío si el patrón pasa una validación interna y un mensaje con que ha vulnerado la comprobación en caso contrario. Cada implementación de *Pattern* debiera implementar un método estático "`+getParser(): PatternParser`" de forma que sea la propia implementación la que conozca que *PatternParser* concreto ha de procesarla.

**PatternParser** es la interfaz de los *parsers* concretos para cada patrón. Contiene un método "`+parse(info: PatternParsedFiles): Pattern`" que recibe la lista de ficheros que contienen la información encontrada.

**Parser** es una clase de parseo general que más tarde llama a un *parser* concreto para cada patrón. Trabaja en tres etapas: primero recorre todos los ficheros devueltos por el *crowler* y recoge el tipo de patrón y el identificador de cada patrón marcado, creando un objeto *PatternFiles* para cada uno. En la segunda etapa se añadieron los ficheros con el mismo identificador al *PatternFiles* correspondiente. En la tercera etapa, se pasa el *PatternParsedFiles* por el *PatternParser* patrón al que pertenece, el cual que devuelve un objeto *Pattern*. Una vez parseados todos los *PatternParsedFiles*, se devuelve la lista de *Pattern* resultante.

**PatternWriter** interfaz la cual implementan las distintas clases encargadas de, dado un objeto de la clase correspondiente que herede de *Pattern*, traducirlo a XML según el formato definido. Contiene un método público *patternToDomElement* que recibe la información del patrón y devuelve un nodo XML con el árbol DOM que representa el patrón. Sus implementaciones, debido a que su función no depende de nada más que de la información recibida como parámetro y no tiene estado interno que pueda modificarse, debieran programarse siguiendo el patrón *Singleton*.

**Crowler** es la clase encargada de, dada una ruta inicial, recorrer la estructura de ficheros de forma recursiva recogiendo aquellos de código fuente que sean de interés y guardándolos en una lista. Se ha decidido hacerlo así de forma que si en un futuro se amplia el sistema a otros formatos o lenguajes, se generalice como una interfaz que implementen los distintos *crowler*.

**Analyzer** clase principal de control de flujo. Ejecutable tanto importándola como librería externa, como desde línea de comandos.

**SystemUtil** es una clase creada ante la necesidad de gestionar los mensajes de error de forma centralizada. Se han agrupado todos los mensajes de error ubicándolos en un único fichero de texto, donde cada línea puede ser: una línea en blanco, su primer carácter es % y es ignorada o sigue la estructura *identificador = mensaje*. La clase *SYstemUtil* ofrece un método *public static String getMessageById (String identifier)* que busca en dicho fichero el mensaje que se corresponda con el identificador y lo devuelve. En caso de no encontrarlo o no poder abrir el fichero devuelve una *SYstemUtilException*. Con esto se ofrece a mayores la herramienta necesaria para poder internacionalizar el sistema (cumpliendo así el requisito RAF-01). Puesto que prácticamente todas las clases principales hacen uso de esta clase, se ha omitido en los diagramas de UML.

La parte específica del patrón Estado lo componen las clases:

**StatePattern** extiende *Pattern* y permite la representación de la información necesaria por el sistema como se mostró en al figura 5.6

**StatePatternParser** es el *Parser* concreto del patrón Estado. Extiende *PatternParser*.

**StatePatternWritter** es el *Writter* concreto del patrón Estado. Extiende *PatternWritter*. Implementa el patrón *Singleton*.

Estas clases y sus relaciones pueden observarse en el diagrama de la figura 5.7. Aquellos componentes que van a ser extendidos o implementados en cada iteración para sus correspondientes patrones se han destacado en color.

Tras recopilar la información de los patrones, estos pasan una verificación.

En el caso del patrón Estado se verifica:

- Todas las transiciones se han definido sobre acciones (métodos públicos de la clase *State*).

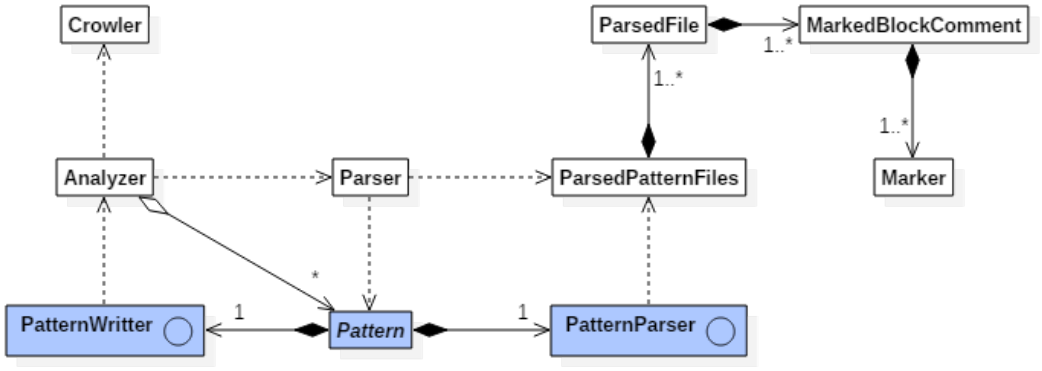


Figura 5.7: Diagrama de clases del módulo de recogida de información

- Todas las transiciones transitan a uno de los estados marcados.
- Existe estado inicial.
- No existen estados no alcanzables. En una primera pasada se mira si hay estados a los que no transita ninguna transición. En una segunda pasada, se comprueba que se puede llegar a todos los estados desde el inicial con alguna secuencia de acciones.

No se verifica:

- Los constructores y parámetros, pues en caso de ser incorrectos, el propio Java lanza una excepción suficientemente explicativa.
- En caso de haber estados finales, no se comprueba explícitamente que existe una secuencia de transiciones desde el estado inicial a ese estado pues ya esta implícito al comprobar que no haya estados no alcanzables.
- Que el autómata se termine en un estado final, pues ese es el problema de la parada que es irresoluble.

A continuación se procede a generar la representación intermedia para cada patrón. La clase *PatternWritter* proporciona la interfaz general para este proceso, siendo los *PatternWritters* concretos (*StatePatternWritter* en este caso) los encargados de recorrer el objeto *StatePattern* y escribir el XML correspondiente.

### 5.1.10. Diseño del módulo de generación de código

La generación de código tiene dos etapas: lectura de la representación intermedia para generar una representación interna y escritura de las pruebas correspondientes usando la información de la estructura interna.

La etapa de lectura genera una representación intermedia completa, en la cual también se hace explícito el conocimiento implícito por omisión, como es que si no está definido un "builder", se usa un constructor vacío.

La forma de trabajar se ha generalizado y estandarizado mediante las siguientes clases e interfaces:

**Pattern** Clase casi idéntica (pero independiente) a la definida para el primer módulo y de igual manera será extendida por la clase general que represente un patrón concreto. De forma análoga al módulo de análisis en el que las implementaciones de esta clase conocen su *PatternParser* y *PatternWritter* correspondientes, en este módulo las implementaciones de esta clase conocen su *PatternBuilder* (con un método estático) y su lista de *PatternTest* asociados.

**PatternTest** Clase abstracta con un atributo *Pattern* que se le pasa al constructor y deberá coincidir con el patrón asociado a cada implementación que se le dé. Tiene un método abstracto al que llamar el cual devuelve un *String* con el código completo de la clase con las pruebas que le corresponde construir sobre el patrón.

**PatternBuilder** interfaz que se implementa para cada patrón soportado y contiene un método "`+getPatternFromXML(rootElement: Elemen):Pattern`" cuya implementación deberá coger un elemento XML, que será la raíz del árbol de una implementación de un patrón, y generar el objeto *Pattern* correspondiente.

**Builder** es la clase encargada de coger la definición en XML de los patrones identificados por el primer módulo, y devolver una lista de objetos *Pattern*.

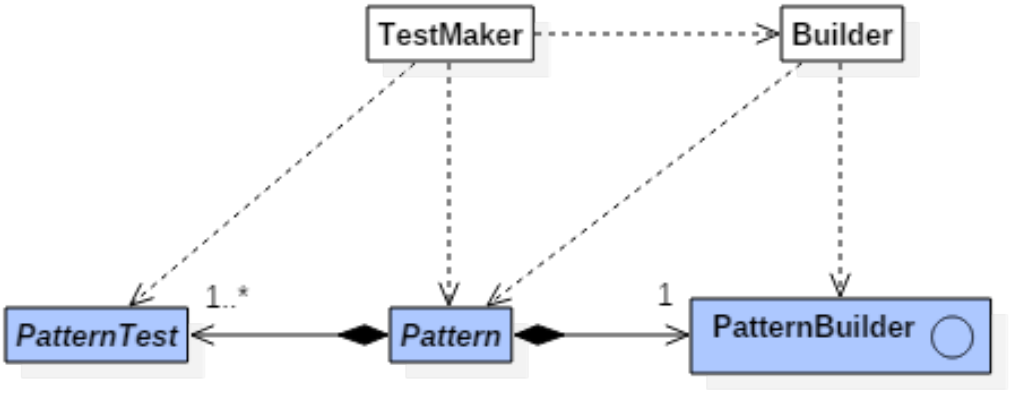


Figura 5.8: Diagrama de clases del módulo de generación de código

Para cada sub-árbol del XML, identifica qué clase de patrón contiene e invoca al *PatternBuilder* correspondiente.

**TestMaker** clase principal de control de flujo. Ejecutable tanto importándola como librería externa, como desde línea de comandos.

Estas clases y sus relaciones pueden observarse en el diagrama 5.8. De nuevo aquellos componentes que van a ser extendidos o implementados en cada iteración para sus correspondientes patrones se han coloreado.

La parte correspondiente del patrón Estado serían pues las clases:

**StatePattern** clase principal de una estructura encargada de representar internamente la información contenida en el XML leído. Hereda de *Pattern*. Puesto que la información es la misma, este conjunto de clases es muy similar al del primer módulo, con diferencias como un método auxiliar que genera las sentencias con las dependencias necesarias para usar las clases del patrón a probar.

**StatePatternBuilder** clase encargada de transformar una definición XML en un objeto *StatePattern*. Hereda de *PatternBuilder*.

**StatePatternTestsPFE1** implementación de *PatternTest* que escribe las pruebas referentes a *PFE-1: transiciones*.

**StatePatternTestsPFE2** implementación de *PatternTest* que escribe las pruebas referentes a *PFE-2: secuencias de transiciones que deben llevar la máquina de estados a un estado final.*

**StatePatternTestsPNFE1** implementación de *PatternTest* que escribe las pruebas referentes a *PNFE-1: invariante frente al orden de las acciones para llegar de un estado a otro.*

Además, se hace necesario tener en cuenta los paquetes que haya que importar para ponerlos al principio del fichero resultante. Para ello se dividió la responsabilidad de saber que `imports` eran necesarios entre la implementación de la clase Pattern, y cada implementación de PatternTest, siendo la primera la encargada de proporcionar un *string* con los `imports` de sus propias clases, y la segunda la encargada de saber qué `imports` adicionales se han de incluir en la prueba.

Se ha partido del supuesto que el XML de entrada, al ser proporcionado por el primer módulo, es correcto y no se han preparado mensajes de error al leerlo ni se han realizado validaciones espaciales. Aun así, se ha preparado para que dé un error genérico en caso de que encuentre un error de formato como que no se encuentre un tag obligatorio, aparezca más de una vez un tag único o se detecte algún tag desconocido.

#### 5.1.10.1. Diseño de PFE-1 - Transiciones

Esta prueba ha decidido hacerse de forma atómica, con un test por transición, comprobando todas las acciones posibles desde todos los estados, siendo el estado origen igual al estado destino para aquellas acciones sin transición definida.

En la figura 5.9 puede verse un ejemplo siguiendo el código de las figuras 5.4 y 5.5.

```
@Test
public void test001TransitionState1Action1State2 ()
    throws Exception {

    StateHeader initialState = new ConcreteState1 ();
    StateHeader resultState = new ConcreteState2 ();
    Context context = new StateContext (8080, "pruebas");

    context.setState1Header (initialState);
    initialState.action1 (context, true, "transicion");

    assertEquals (resultState, context.getState1Header ());
}
```

Figura 5.9: Ejemplo de código de PFE-1

#### 5.1.10.2. Diseño de PFE-2 - Camino

Al no haberse encontrado una forma aceptable para definir la secuencia de acciones, se decidió diseñar un código de pruebas lo más sencillo posible con huecos que deberán ser completados por el usuario.

Para ello, se decidió dotar a cada transición de un identificador y hacer que el usuario solo tenga que indicar la secuencia de identificadores que quiere usar. Para dar a conocer dichos identificadores al usuario, se mostrará un texto comentado al principio de la función a rellenar.

Al usuario se le pedirá que, si quiere que se realice la prueba, introduzca los identificadores en una lista por orden, ejecutándose luego esa lista en otra función que devolverá si ha llegado o no a un estado final.

La idea es que el usuario repita el código del @Test tantas veces como quiera con diferentes secuencias.

Un ejemplo de este algoritmo ya completado puede encontrarse en las Figuras 5.10 y 5.11.

La prueba, antes de ser modificada por el usuario, finaliza dando un error al lanzar una excepción `UnsupportedOperationException` con el mensaje *"The*

```
@Test
public void testTransitionSecuenceToFinalState01 ()
    throws Exception {

    List<int> transitionOrder = new ArrayList<int> ();

    /*
    *    Identifiers of each defined transition for each action
    *         set in order as in the example:
    *              transitionOrder.add(0);
    * action1
    *    1 - (context, true, "transicion")
    *    2 - (context, false, "transicion")
    *
    * action2
    *    3 - (context)
    */

    // insert using: transitionOrder.add(identifier);

    transitionOrder.add(1);
    transitionOrder.add(3);
    transitionOrder.add(2);
    transitionOrder.add(1);
    transitionOrder.add(3);

    assertTrue (executeActionSecuence (transitonOrder));
}
```

Figura 5.10: Ejemplo de código de PFE-2 parte 1

*user must finish this test. This exception must be removed after that."* para indicar que no está terminado y que deberá ser eliminado por el usuario una vez añada las transiciones deseadas.

#### 5.1.10.3. Diseño de PNFE-1 - Acceso concurrente

Para esta prueba, se realiza una lista que contiene una variación con repetición de las n acciones cogiendo 2n acciones. Éstas se ejecutarán todas en secuencia de forma no determinista varias veces. En caso de estar definido el patrón como independiente del orden de ejecución, se busca que todas las secuencias terminen

```
private boolean executeActionSecuence (List<int> transitonOrder){

    StateContext context = new State1Context (8080, "pruebas");

    // If the context doesn't begin in the initial state
    if (!context.getState().equals(new StateConcrete1 ()))
        return false;

    // Execute the transitions in the defined order
    for (int i: transitonOrder) {
        switch (i) {
            case 1:
                context.getState().action1(context, true, "transicion");
                break;
            case 2:
                context.getState().action1(context, false, "transicion");
                break;
            case 3:
                context.getState().action2(context);
                break;
            default:
                return false;
        }
    }

    // Conpare the final state obtained with the final states defined
    StateHeader state = context.getState();
    return (getFinalStateClassNames().contains(state.getClass()));

}

// Function that return a list with an instance of each final state
private List<StateHeader> getFinalStateClassNames()
{
    ...
}
```

Figura 5.11: Ejemplo de código de PFE-2 parte 2

en el mismo estado. En caso de estar definido como dependiente del orden, se busca que al menos una termine en un estado diferente al resto.

Esta prueba supone un problema casuístico en cualquier caso por introducir no determinismo (aleatoriedad). En caso de que todas las secuencias fueran iguales (altamente improbable pero no imposible) esta prueba sería inútil. Para resolverlo, se ha decidido que la secuencia de acciones se seleccione al ejecutar la prueba, de forma que al ejecutar dos o tres veces la prueba, la probabilidad de que salgan las mismas secuencias dentro de cada ejecución es despreciable.

La aproximación para ejecutar las distintas transiciones es equivalente a la de PFE-2: se les da un identificador a las posibles transiciones de un estado y se decide un conjunto de identificadores a ejecutar, como se observa en las figuras 5.12 y 5.13.

## 5.2. Iteración 2: *Estrategia*

Durante esta iteración, se ha analizado del patrón Estrategia y diseñado tanto un conjunto de pruebas (o esqueleto de pruebas) como una representación abstracta en formato XML. Se ha diseñado también un conjunto de marcadores que permitan extraer toda la información necesaria para realizar las pruebas.

Con eso, se amplió la primera versión del módulo software de análisis de código y el de generación de código.

### 5.2.1. Análisis del patrón

El patrón Estrategia (Strategy) se clasifica como patrón de comportamiento porque determina cómo se debe realizar el intercambio de mensajes entre diferentes objetos para resolver una tarea. Permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier

```
public void test () {

    List<List<int>> transitionsList = new ArrayList<List<int>> ();

    List<int> transitionCombination = new ArrayList<int> ();
    List<int> transitionCombinationTmp = new ArrayList<int> ();
    List<int> permutedTransitions = new ArrayList<int> ();

    int transitionsNumber = transitions.size();
    Random random = new Random ();

    // Get a random combination of actions
    for (int i = 0; i < permutationSize; i++) {
        int k = random.nextInt(transitionsNumber) + 1;
        transitionsCombination.add (k);
    }

    // permute the random combination permutation times and
    // add ir to transitionsList
    for (int i = 0; i < permutations; i++) {
        transitionCombinationTmp = transitionCombination.clone();
        permutedTransitions = new ArrayList<int> ();
        for (int j = 0; j < permutationSize; j++) {
            int k = random.nextInt(transitionCombinationTmp.size()) + 1;
            permutedTransitions.add (transitionCombinationTmp.get(k));
            transitionCombinationTmp.remove(k);
        }
        transitionsList.add(permutedTransitions);
    }

    // Execute the transition permutations
    ConcreteState execution, executionBefore;
    boolean hasDiferent = false;
    executionBefore = executeTransitionList (transitionsList.get(0)));
    for (int i = 1; i < permutations; i++) {
        execution = executeTransitionList (transitionsList.get(i)));
        // Compare results
        hasDiferent = hasDiferent ||
            (execution.getClass() == executionBefore.getClass ());
        executionBefore = execution;
    }

    // Is not order dependent and has diferent
    assertFalse (hasDiferent);

}
```

Figura 5.12: Ejemplo de código de PNFE-1 parte 1

```
private ConcreteState executeActionSecuence (List<int> transitonOrder){

    StateContext context = new State1Context (8080, "pruebas");

    // If the context doesn't begin in the initial state
    if (!context.getState().equals(new StateConcrete1 ()))
        return null;

    // Execute the transitions in the defined order
    for (int i: transitonOrder) {
        switch (i) {
            case 1:
                context.getState().action1(context, true, "transicion");
                break;
            case 2:
                context.getState().action1(context, false, "transicion");
                break;
            case 3:
                context.getState().action2(context);
                break;
            default:
                return false;
        }
    }

    return context.getState();
}
```

Figura 5.13: Ejemplo de código de PNFE-1 parte 2

momento, incluso en tiempo de ejecución. Si muchas clases relacionadas se diferencian únicamente por su comportamiento, se crea una superclase que almacene el comportamiento común y que hará de interfaz hacia las clases concretas. Además, la jerarquía de clases que implementa el algoritmo encarnado por la estrategia, es potencialmente reutilizable, esto es, usable por diferentes clientes/contextos.

Consta de un Contexto que usa los algoritmos delegando en las estrategias, una Estrategia genérica que define la interfaz común que usará el contexto para todos los algoritmos soportados, y un conjunto de Estrategias Concretas, cada una de las cuales implementa uno de los algoritmos.

Como contrapartida, aumenta el número de objetos creados, por lo que se produce una penalización en la comunicación entre estrategia y contexto (hay una indirección adicional).

Un ejemplo claro de uso de este patrón es el caso de los diferentes algoritmos de ordenación. Tenemos una interfaz común que ofrece un método ordenar el cual recibe una lista de elementos (con una relación de orden definida para ellos) y devuelve otra lista con los mismos elementos ordenados, y luego diferentes implementaciones de dicha interfaz, cada una usando un algoritmo de ordenación diferente (burbuja, merge-sort, quick-sort, etc.). Desde el contexto se llama al método ordenar pasando la lista, por ejemplo de enteros, y sea cual sea el algoritmo implementado en la Estrategia Concreta, el resultado será la lista ordenada,pero dependiendo del caso concreto de los datos de entrada y el algoritmo concreto usado, la eficiencia puede variar mucho (por ejemplo, para Burbuja la complejidad varia de $O(n)$ mejor caso y $O(n^2)$ en el peor caso).

Esta estructura puede observarse en la figura 5.2.

### 5.2.2. Análisis de pruebas funcionales sobre el patrón

**PFEST-1 - Mismo resultado**

La prueba más obvia es, en caso de poder comparar el resultado obtenido con los diferentes algoritmos, ejecutarlos sobre el mismo conjunto de datos inicial

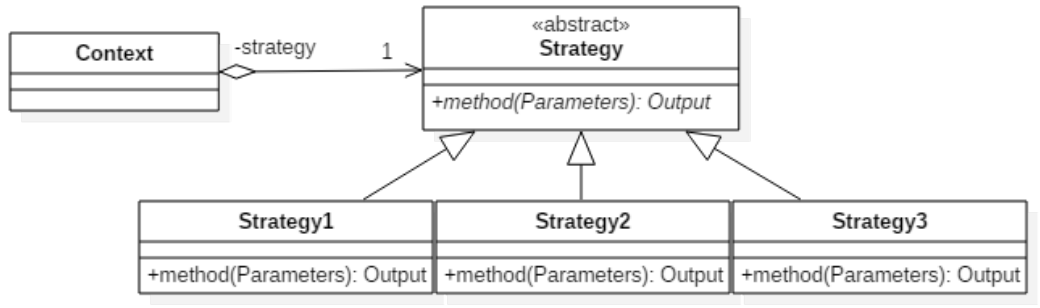


Figura 5.14: Diagrama de clases general del patrón Estrategia

y comprobar que los resultados son acordes. Volviendo al ejemplo de los algoritmos de ordenación, que pasando la misma lista desordenada a dos (o más) algoritmos diferentes, el resultado es el mismo (una lista de elementos ordenados) independientemente del algoritmo concreto.

Para ello es necesario contar con una manera de comparar los valores devueltos por los diferentes algoritmos. Para los tipos básicos del lenguaje, puede usarse la operación `==`. Para el resto de resultados, existen tres formas de compararlos (pues `==` compara igualdad referencial en ese caso): mediante el método "equals" de la clase devuelta, si la clase devuelta extiende `Comparable<T>` se puede usar el método `compareTo` (buscando que devuelva 0 como resultado) y por último mediante una clase externa que extienda `Comparator<T>`.

El caso más sencillo es aquel en el que los métodos de la EstrategiaGenerica reciben uno o varios parámetros simples (tipos básicos ya definidos en el lenguaje con función de comparación también definida), y devuelven otro tipo simple que es comparable mediante `==`.

Puede darse el caso de que el resultado se "devuelva" dentro de uno de los parámetros (modificando el propio parámetro) y no con una instrucción "return".

Se han diferenciado tres versiones de esta prueba:

**Básica:** Resultado mediante "return" de tipo simple comparable con `==`.

**Intermedia:** Resultado mediante "return" de una clase de usuario que o bien tenga implementado equals, extienda `Comparable<T>` o se proporcione un `Comparator<T>`.

**Compleja:** Lo mismo que la Intermedia, pero el resultado es devuelto a través de uno de los parámetros y no mediante el "return".

### 5.2.3. Análisis de pruebas no funcionales sobre el patrón

Puesto que el comportamiento funcional de las diferentes estrategias ha de ser el mismo, lo que las diferencie ha de ser factores no funcionales como el uso de memoria, velocidad, etc.

Por ello, no se ha diseñado ninguna prueba no funcional que pueda efectuarse a este patrón de forma genérica.

### 5.2.4. Información necesaria (mínimo)

- Identificador del patrón Estrategia dentro del proyecto (por si hay más de uno en la implementación del sistema o componente).
- Nombre completo de la clase abstracta/interfaz EstrategiaGenerica.
- Nombre completo de cada clase EstrategiaConcreta.
- Para cada método de la EstrategiaGenerica, que el tipo devuelto: comparable usando ==, tenga una implementación válida de equals, que implemente `Comparable<T>` o que se ofrezca una clase externa que implemente `Comparator<T>`.
- Ejemplo de instanciación de toda clase implicada que precise instanciarse y que no se instancie mediante un constructor vacío.
- Un ejemplo de parámetros de entrada de los métodos del Estado genérico que puedan llegar a descubrir discordancias entre los resultados de los algoritmos.

Un detalle a investigar es si no se indica como comparar los resultados, que comportamiento por defecto adoptar.

El método "equals", por defecto viene heredado de "Object", por lo que toda clase lo tiene implementado por defecto, y fuera de los tipos básicos del lenguaje la implementación por defecto hace una comparación de igualdad referencial, lo cual salvo casos concretos que así lo requieran no sirve a nuestro propósito. Por lo tanto, si el usuario no lo ha redefinido para su clase, deberá hacerlo de cara a poder usar las pruebas generadas. Esta es una aproximación intrusiva de cara al código del usuario, pues se le obliga a implementar algo en sus clases que puede no querer tener.

La opción de la interfaz `Comparable<T>` tiene el mismo problema de ser intrusiva que el `equals`, pero además en sí el método "compareTo" no esta pensado para saber si dos objetos son iguales, sino para establecer una relación de orden entre ellos, en la cual viene incluido el caso de que sean iguales.

El uso de un `Comparator<T>` es la opción menos intrusiva con el código del usuario, al ser una clase externa cuyo acceso puede restringir, pero incluye el problema de la opción de usar la interfaz `Comparable<T>` encuanto a que el método que incluye pretende determinar una relación de orden entre los objetos comparados.

Una vez más nos encontramos ante la problemática de las dependencias, y en este caso no solo con el contenido de los marcadores "Builder" sino también con los tipos devueltos por las acciones. Es más sencillo extraer el tipo devuelto por las acciones y buscar alguna dependencia en el fichero de código que coincida, pues no hay que hacer un análisis interno para saber que dependencias buscar. En este caso, y por ser relativamente fácil de implementar, se incluirá en el diseño, dejando excluido el caso en que se referencia al espacio de nombres del que cuelga y no la dependencia concreta.

### 5.2.5. Información adicional

- Más ejemplos de parámetros de entrada de los métodos de la Estrategia genérica que puedan llegar a descubrir discordancias entre los resultados de los algoritmos.

### 5.2.6. Análisis de riesgo

De forma análoga a la primera iteración, se ha asignado un identificador a cada riesgo siguiendo lo descrito en la sección 2.10 correspondiente a nomenclatura.

**REST-01** Tener que rehacer trabajo por descubrir algún problema o una forma más efectiva de hacer algo (ya implementado).

- Riesgo: medio
- Repercusión: alta
- Acción preventiva: realizar un diseño exhaustivo
- Acción correctivo: rehacer la parte afectada

**REST-02** No encontrar una forma factible de recopilar los datos del patrón para una versión simple de PFEST-1.

- Riesgo: bajo
- Repercusión: alta
- Acción preventiva: NA
- Acción correctiva: abortar el patrón y pasar a la It-3.

**REST-03** No encontrar una forma factible de recopilar los datos del patrón para una versión intermedia de PFEST-1.

- Riesgo: medio
- Repercusión: media

- Acción preventiva: NA
- Acción correctiva: no hacer casos de prueba de complejidad intermedia o dejarlos para que el usuario los complete

**REST-04** No encontrar una forma factible de recopilar los datos del patrón para una versión compleja de PFEST-1.

- Riesgo: alto
- Repercusión: baja
- Acción preventiva: NA
- Acción correctiva: no hacer casos de prueba complejos o dejarlos para que el usuario los complete

**REST-05** No encontrar una forma factible de implementar una versión simple de PFEST-1.

- Riesgo: bajo
- Repercusión: alto
- Acción preventiva: NA
- Acción correctiva: dejar partes de la prueba para que el usuario los complete o, de no ser factible tampoco eso, se creará el esqueleto más completo posible y se indicará que es lo que el usuario debe completar para que la prueba funcione.

**REST-06** No encontrar una forma factible de implementar una versión intermedia de PFEST-1.

- Riesgo: medio
- Repercusión: media
- Acción preventiva: NA
- Acción correctiva: no hacer casos de prueba de complejidad media o dejarlos para que el usuario los complete.

**REST-07** No encontrar una forma factible de implementar una versión compleja de PFEST-1.

- Riesgo: alto
- Repercusión: bajo
- Acción preventiva: NA
- Acción correctiva: no hacer casos de prueba complejos o dejarlos para que el usuario los complete.

Resumen:

| Riesgo / Repercusión | Alta | Media | Baja |
|---|---|---|---|
| **Alto** | 0 | 0 | 2 |
| **Medio** | 1 | 2 | 0 |
| **Bajo** | 2 | 0 | 0 |

Cuadro 5.2: Iteración 2 Recuento de riesgos

El riesgo de tener que rehacer algo ya implementado siempre va a estar presente y tener una alta repercusión, por lo que no será tenido en cuenta para la decisión de si llevar a cabo la implementación o no.

Los riesgos referentes a la versión básica de la prueba (REST-02 y REST-05) tienen una exposición al riesgo baja y una repercusión alta, pero es de esperarse pues es la versión más sencilla, y si ni esa es posible, aumentar la complejidad no facilitará su implementación.

Los riesgos de la versión intermedia (REST-03 y REST-06) son de exposición y repercusión media, siendo deseable que puedan estar en la versión final, mas no indispensable.

Los riesgos con mayor exposición (REST-04 y REST-07) son referentes a la parte de la versión compleja de la prueba, y por lo tanto es, hasta cierto punto, opcional, por lo que tienen una repercusión media y en caso de darse, no sería necesario encontrarles solución, pudiendo decidirse no implementar la versión más compleja sin que eso afecte a la versión básica.

Se ha decidido que estos riesgos son asumibles y por lo tanto se da luz verde a la implementación de las pruebas para este patrón.

### 5.2.7. Definición del lenguaje XML

En lo que respecta al patrón abordado en esta iteración, la definición del lenguaje XML diseñado se ha extendido con la etiqueta *strategyPattern* y su contenido, descrito a continuación. Un ejemplo de esta etiqueta se puede observar en la figura 5.15.

Dentro de una etiqueta *strategyPattern* encontraremos:

**Id** identificador de la implementación concreta para el sistema de generación de pruebas.

**generalStrategy** contiene la información referente a la estrategia genérica de la implementación del patrón. Se indica su *package* y *classname*.

**strategyActions** conjunto de métodos públicos que expone la interfaz genérica y que por ende podrán ser invocados en las diferentes estrategias concretas. Se compone de uno o más *strategyAction*.

**strategyAction** cada uno de los métodos del conjunto de métodos públicos de la estrategia. Se compone de: *returnType*, *returnTypePackage*, *name*, *parameterSet* y *comparation*.

**parameterSet** conjunto de parámetros de prueba proporcionados por el usuario para el método en cuestión. Se compone de uno o más *parameters*, que son cadenas de caracteres que se copiarán sin modificación al llamar al método. Han de ir sin paréntesis.

**comparison** forma de comparación de los resultados del método, pudiendo ser *equals* cuando se ha de usar el susodicho método o la comparación con ==, *comparable* cuando la clase resultado implementa la interfaz o *comparator* con su *package*, *className* y *builder*. En caso de ser *comparator* y no indicarse *className*, se entenderá que no hay tal clase y se proporcionará junto

a la prueba un esqueleto que implemente *Comparator* y que deberá ser completado por el usuario.

**concreteStrategies** contiene cada una de las estrategias concretas a probar, que han de implementar la estrategia genérica y estar todas bajo el mismo *package*.

**concreteStrategy** cada una de las estrategias concretas. Contiene un *className* y un *builder*.

**package** es la ubicación de la clase dentro de la estructura del proyecto.

**classname** es el nombre de la clase.

**builder** es el fragmento de código que se usará para instanciar la clase. Si no se especifica, se asume constructor vacío.

### 5.2.8. Diseño de marcadores

En el diseño de marcadores se han seguido las mismas normas que en la iteración 1, por los mismos motivos y por coherencia del propio sistema. A continuación, se procederá a definir los marcadores correspondientes a esta iteración. Aunque no se reiteren los ya definidos en la anterior iteración, se entiende que siguen vigentes sin modificaciones, siendo estos una ampliación.

**@pattern Strategy <identificator>**

Identifica una parte de un patrón Estrategia.
Ha de ir seguido de un *@patternElement*.
*Identificator* es el identificador del patrón y es opcional aunque recomendado. En caso de "*@patternElement Strategy*" si no se indica, se tomará del nombre de la clase. En caso de "*@patternElement ConcreteStrategy*" si no se indica, se buscará en el extends o implements.

**@patternElement**

Puede ser de tres tipos: *Strategy* y *ConcreteStrategy*, que son mutuamente excluyentes, y *Builder*, combinable con *ConcreteStrategy*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<patterns>
    <strategyPattern>
        <id>identificator</id>
        <generalStrategy>
            <package>testExamples.parser.state.correct</package>
            <classname>StateHeader</classname>
            <strategyActions>
                <strategyAction>
                    <returnType> ... </returnType>
                    <returnTypePackage> ... </returnTypePackage>
                    <name> ... </name>
                    <parameterSet>
                        <parameters> ... </parameters>
                    </parameterSet>
                    <comparison>
                    [
                        <equals />
                      | <comparable />
                      | <comparator>
                            [
                             <package> .. </package>
                             <classname> ... </classname>
                             <builder> ... </builder>
                            ]
                        </comparator>
                    ]
                    </comparison>
                </strategyAction>
            </strategyActions>
        </generalStrategy>
        <concreteStrategies>
            <package> ... </package>
            <concreteStrategy>
                <classname> ... </classname>
                <builder> ... </builder>
            </concreteStrategy>
        </concreteStrategies>
    </strategyPattern>
</patterns>
```

Figura 5.15: Ejemplo del lenguaje XML para el patrón Estrategia

**@patternElement Strategy**

Identifica la clase como Estrategia genérica.

A efectos de verificación de los datos, los métodos públicos serán considerados acciones del patrón.

**@patternElement ConcreteStrategy**

Identifica la clase como Estrategia Concreta.

Todas las Estrategias Concretas han de estar en el mismo "package".

**@patternElement Builder <builderInvocation>**

Recibe un ejemplo de instanciación de la clase, el cual se usa en las pruebas.

BuilderInvocation se toma como un literal (deberá ir entre < y >). De momento no se permite > dentro del Builder.

De no ponerse, se usa un constructor vacío.

Solo puede ir precedido de *@patternElement ConcreteStrategy*.

**@patternElement Action**

Acción implementada por cada Estrategia Concreta.

**@patternElement Comparison**

Se refiere al modo de comparar los resultados de las acciones. Puede ser *Equals*, *Comparable* o *Comparator*.

Si no se indica, se entiende por defecto el valor *Equals*.

**@patternElement Comparison Equals**

Indica que la forma de comparación es mediante el método "equals" de la propia clase resultante (o su *wrapper* si es un tipo básico como `int` cuyo *wrapper* es `Integer`).

**@patternElement Comparison Comparable**

Con esto se entiende que el resultado implementa la interfaz `Comparable<T>` y por lo tanto se invocará al método "compareTo" buscando que devuelva 0.

**@patternElement Comparison Comparator [<package> <className> <Builder>]**

Indica que la comparación se hace mediante una clase externa que implementa la interfaz `Comparator<T>`.

Si "Builder" se omite, se tomará por defecto un constructor vacío.
"className" ha de ser una palabra sola, que empiece con mayúscula, sin signos de puntuación.
Si "package", "Builder" y "className" son omitidos, se entenderá que junto a la prueba se deberá crear un esqueleto del `Comparator<T>` a rellenar por el usuario.

**@patternAction Execution <ParamLine>**
Identifica un conjunto de parámetros, en orden, que se pasarán a la acción de la Estrategia.
*ParamLine* es una sucesión de valores separados por comas que serán parseados y usados en el mismo orden especificado.
Si no se especifica ningún *Execution*, se entiende que el método se ejecuta sin parámetros.
Puede haber tantas acciones de ejecución como quiera el usuario.

Un ejemplo de marcadores sobre código se muestra en la figura 5.16 (Estrategia genérica) y en la figura 5.17 (Estrategias concretas).

La información recabada se guarda en la estructura de clases descrita en la figura 5.18.

### 5.2.9. Diseño del módulo de recogida de información

Gracias al énfasis en la reutilización realizado durante el diseño de la primera iteración, para esta segunda iteración solo será necesario implementar la representación interna del patrón, una clase que extienda `PatternParser` y otra que extienda `PatternWriter`, siendo el resto del algoritmo, salvo un par de puntos donde ha de añadirse la nueva opción en cláusulas "switch", independiente del patrón en sí.

Así, tenemos las clases:

**StrategyPattern** extiende *Pattern* y permite la representación de la información necesaria por el sistema como se mostró en al figura 5.18.

```
/*
* @pattern Strategy
* @patternElement Strategy
*/
public interface StrategyExample {

    /*
    * @patternElement Action
    * @patternElement Comparison Equals
    * @patternElement Execution < new Date (2017, 01, 01), true >
    * @patternElement Execution < new Date (2017, 12, 31), true >
    */
    public Date action1 (Date date, boolean future);

    /*
    * @patternElement Action
    * @patternElement Comparison Comparable
    */
    public UsersClassComparable action2 ();

    /*
    * @patternElement Action
    * @patternElement Comparison Comparator
    *     <tfg.sw.test.integration.pattern.strategy>
    *     <UsersComparator> < new UsersComparator (3); >
    * @patternElement Execution < new UsersClassComparable (); >
    */
    public UsersClassComparator1 action3 (UsersClassComparable input);

    /*
    * @patternElement Action
    * @patternElement Comparison Comparator
    */
    public UsersClassComparator2 action4 ();
}
```

Figura 5.16: Ejemplo de código con marcadores para el patrón Estrategia

```
/*
* @pattern Strategy <StrategyExample>
* @patternElement ConcreteStrategy
* @patternElement Builder < new Strategy1 () >
*/
public class Strategy1 implements StrategyExample {
    ...
}

/*
* @pattern Strategy
* @patternElement ConcreteStrategy
*/
public class Strategy2 implements StrategyExample {
    ...
}
```

Figura 5.17: Ejemplo de código con marcadores para el patrón Estrategia

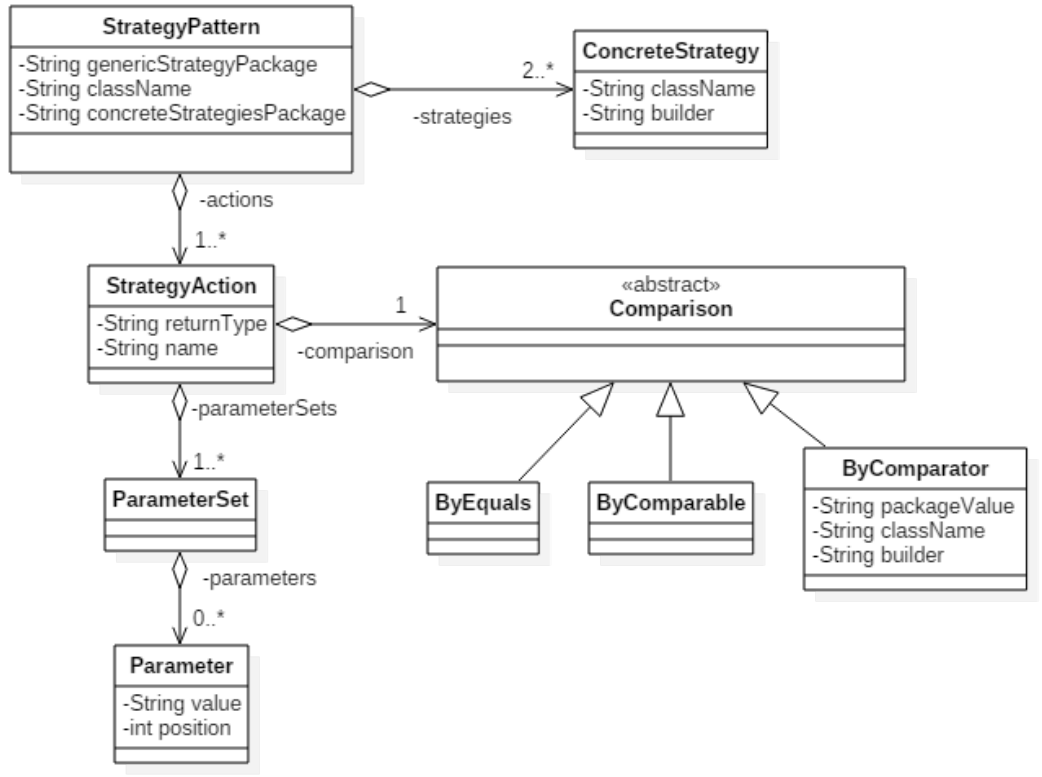


Figura 5.18: Diagrama de clases de la representación interna del patrón Estrategia

Figura 5.19: Diagrama de clases del módulo de generación de código

**StrategyPatternParser** es el *Parser* concreto del patrón Estrategia. Extiende *PatternParser*.

**StrategyPatternWritter** es el *Writter* concreto del patrón Estado. Extiende *PatternWritter*.

### 5.2.10. Diseño del módulo de generación de código

De la misma forma que en la primera iteración, se ha separado este módulo en dos partes, lectura de la información desde un XML, y escritura de las pruebas.

Puesto que se ha reutilizado toda la parte común de este módulo (como son las clases StatePattern, Builder y PatternTest), solo ha sido necesario implementar las partes concretas correspondientes al patrón Estrategia:

**StrategyPattern** Clase principal de una estructura encargada de representar internamente la información contenida en el XML leído. Hereda de "Pattern". Puesto que la información es la misma, este conjunto de clases es muy similar al del primer módulo, con añadidos como un método auxiliar que genera las sentencias import necesarias para usar las clases del patrón a probar.

**StrategyPatternBuilder** Clase encargada de transformar una definición XML en un objeto "StrategyPattern". Hereda de "PatternBuilder".

**StrategyPatternTestsPFEST1** Implementación de PatternTest que escribe las pruebas referentes a PFEST-1: Mismo resultado.

Estas clases y sus relaciones pueden observarse en el diagrama 5.8.

También es necesario tener en cuenta las dependencias necesarias, responsabilidad la cual una vez más se repartió entre la implementación de Pattern y cada la de PatternTest.

```
@Test
public void test001_action1_Strategy1_Strategy3 ()
    throws Exception {

    StrategyExample strategy1 = new Strategy1 (new Date (2017, 01, 01), true);
    StrategyExample Strategy2 = new Strategy3 (new Date (2017, 01, 01), true);

    Date result1 = strategy1.action1 ();
    Date result2 = strategy2.action1 ();

    // @patternElement Comparison Equals
    assertEquals (result1, result2);
}
```

Figura 5.20: Ejemplo de código de PFEST-1 - Equals

Similarmente, se ha partido del supuesto que el XML de entrada, al ser proporcionado por el primer módulo, es correcto y no se han preparado mensajes de error al leerlo ni se han realizado validaciones espaciales. Aun así, se ha preparado para que dé un error genérico en caso de que encuentre un error de formato como que no se encuentre un tag obligatorio, aparezca más de una vez un tag único o se detecte algún tag desconocido.

#### 5.2.10.1. Diseño de PFEST-1 - Mismo resultado

Esta prueba se basa en la ejecución de un mismo método de la Estrategia genérica desde distintas Estrategias concretas y la comparación del resultado según la forma de comparación definida.

Se ha decidido hacer una prueba por cada combinación de dos Estrategias concretas y método, de forma que si tenemos 3 estrategias concretas y 2 métodos, tendremos 6 pruebas resultantes (3 formas de combinar 2 a 2 las 3 Estrategias por dos métodos a probar).

Las pruebas han sido nombradas según la nomenclatura siguiendo las directrices de la sección 2.10.

Un ejemplo de esta prueba puede verse en la figura 5.20.

```
@Test
public void test001_action2_Strategy1_Strategy3 ()
    throws Exception {

    StrategyExample strategy1 = new Strategy1 (new Date (2017, 01, 01), true);
    StrategyExample Strategy2 = new Strategy3 (new Date (2017, 01, 01), true);

    UsersClassComparable result1 = strategy1.action2 ();
    UsersClassComparable result2 = strategy2.action2 ();

    // @patternElement Comparison Comparable
    int comparisonResult = result1.compareTo(result2);
    assertEquals (0, comparisonResult);
}
```

Figura 5.21: Ejemplo de código de PFEST-1 - Comparable

En caso de comparación mediante la interfaz "Comparable" los resultados se compararán como en la figura 5.21 y en caso de "Comparator" definido, se hará como en la figura 5.22. En caso de definirse como "Comparator" pero no ofrecerse ninguna implementación, se implementa un esqueleto de la implementación que al comparar lanza un "UnsupportedOperationException" y deberá ser completada por el usuario.

## 5.3. Iteración 3: *Observador*

Al final esta iteración no pudo realizarse debido a los desvíos respecto a la planificación inicial del proyecto.

```
@Test
public void test001_action3_Strategy1_Strategy3 ()
    throws Exception {

    StrategyExample strategy1 = new Strategy1 (new Date (2017, 01, 01), true);
    StrategyExample Strategy2 = new Strategy3 (new Date (2017, 01, 01), true);

    UsersClassComparator1 result1 = strategy1.action3 (new UsersClassComparable ());
    UsersClassComparator1 result2 = strategy2.action3 (new UsersClassComparable ());

    // @patternElement Comparison Comparator
    Comparator comparator = new UsersComparator (3);
    int comparisonResult = comparator.compare(result1, result2);
    assertEquals (0, comparisonResult);
}
```

Figura 5.22: Ejemplo de código de PFEST-1 - Comparator

```
@Test
public void test001_action4_Strategy1_Strategy3 ()
    throws Exception {

    StrategyExample strategy1 = new Strategy1 (new Date (2017, 01, 01), true);
    StrategyExample Strategy2 = new Strategy3 (new Date (2017, 01, 01), true);

    UsersClassComparator2 result1 = strategy1.action4 ();
    UsersClassComparator2 result2 = strategy2.action4 ();

    // @patternElement Comparison Comparator
    Comparator comparator = new Action4Comparator ();
    int comparisonResult = comparator.compare(result1, result2);
    assertEquals (0, comparisonResult);
}

private class Action4Comparator implements
    Comparator<UsersClassComparator2> {

    @override
    public int compare(UsersClassComparator2 o1, UsersClassComparator2 o2) {
        throw new UnsupportedOperationException
            (‘‘The user must finish this test. This exception must be "
            + "removed after that.’’);
    }
}
```

Figura 5.23: Ejemplo de código de PFEST-1 - Comparator (esqueleto)

# Capítulo 6

# Implementación y Validación

## Índice general



En este capítulo se describe, para cada iteración, la parte de implementación y problemas surgidos en la fase de ciclo de vida en cascada de "Desarrollar y probar", siguiendo el diseño del capítulo de "Análisis y diseño".

## 6.1. Iteración 1: Estado

A continuación se relata, en lo referente a la primera iteración, lo acaecido durante el proceso de implementación y se describe el proceso de validación, todo siguiendo lo definido en la sección correspondiente de análisis y diseño (5.1).

### 6.1.1. Implementación

El módulo de análisis de código se ha refinado 5 veces, evolucionando desde múltiples lecturas de cada fichero en la primera tentativa, a un sistema en 3

etapas con más de una lectura (aunque menos que antes), y finalmente al sistema de 3 etapas actual con una sola lectura de cada fichero durante la primera etapa, en la cual se extrae la información de todos los marcadores pertinentes en un único paso. Además, en el último refinamiento, se decidió rediseñar las pruebas unitarias desde cero, para asegurar que no se pasaba nada por alto en las sucesivas adaptaciones de las pruebas iniciales.

También se añadió al principio del desarrollo el sistema de internacionalización, que no pertenecía a los requisitos prioritarios, pero requería poco esfuerzo de implantar y simplificaba en gran medida la coherencia de los mensajes del sistema, estando todos localizados en un fichero y no embebidos en el código, así como el mantenimiento de las pruebas unitarias (que es por lo que se surgió la idea), pues si se cambiaba mínimamente un mensaje de error, su prueba asociada también debía ser adaptada, cosa que no pasa con el sistema actual.

Otros refinamientos de menor impacto en la funcionalidad del sistema, son los referentes a su documentación, siendo un ejemplo el añadir un código a cada prueba unitaria, posteriormente a su implementación pues no se pensó antes, para simplificar la relación entre la documentación y el código.

Otro problema surgido durante la implementación fue la necesidad de incluir la librería XMLUnit. En su documentación se explica cómo añadirla a través de Maven, el cual no estaba siendo usado hasta el momento, por lo que se tuvo que migrar de un proyecto Java normal, a un proyecto Java gestionado con Maven, para lo cual se partió de un proyecto nuevo sin ningún arquetipo, se copiaron los fiheros de código y se añadió la librería en el fichero de configuración.

En lo referente al módulo de generación de pruebas, se ha implementado un `Builder` de XML a la representación interna de forma que si en un estado una acción no tiene definida una transición, se crea explícitamente un objeto con la transición implícita que le deja en el mismo estado. Para solucionar el problema del paso de parámetros, se buscará en el resto de estados concretos transiciones sobre esa acción para coger un ejemplo de parámetros. También, en caso de que una transición no tenga definido ningún parámetro, se creará el parámetro implícito "context" de forma explícita. Análogo para los "Builder".

Durante las pruebas de integración y validación, hubo que solucionar el problema de, una vez ejecutado el sistema sobre un conjunto de clases y generadas las pruebas para esas clases, comprobar de forma automatizada que las pruebas fallan si el patrón contiene los errores que se buscan en las pruebas. En un principio se intentó una aproximación compilando y ejecutando el código mediante comandos desde una prueba JUnit, pero tras varias horas invertidas en esa problemática sin resultados, se llegó a la conclusión de que se podía separar aun más los proyectos, teniendo uno propio para las pruebas de integración y validación y otro para los ejemplos de implementación de los patrones y sus pruebas generadas, ambos gestionados por Maven. Una vez los proyectos estuvieron separados, desde el proyecto de pruebas de integración y validación, se ejecuta el comando "mvn test -fae -f RUTA", cogiendo la salida que produce y buscando los resultados esperados en ésta. La opción "-fae" produce que, aunque una prueba falle, se sigan ejecutando el resto, y "-f" selecciona el directorio raíz del proyecto Maven sobre el que ejecutar el comando.

### 6.1.2. Validación

La validación se realizó, para cada módulo por separado mediante pruebas unitarias y de sistema, efectuando después unas pruebas de integración y validación globales.

Se buscó que los resultados obtenidos sean consistentes con los esperados, sin necesidad de que sean exactamente iguales: el orden dentro de las colecciones de objetos no nos interesa, y de los mensajes de error solo se comprueba el principio, que es fijo, pudiendo incluir información adicional a continuación (como puede ser la ruta del fichero donde está el fallo).

Por la agrupación e identificación de los mensajes de error, en las pruebas se buscará el texto correspondiente al identificador de interés, con lo que si se cambia el mensaje en sí la prueba no fallará.

Para la identificación de las diferentes pruebas realizadas, se ha utilizado un código siguiendo lo descrito en la sección 2.10 correspondiente a nomenclatura.

#### 6.1.2.1. Subsistema de recogida de información

Las pruebas se han dividido en las diferentes etapas, asumiendo que, al probar una etapa, las anteriores funcionan correctamente. Así mismo, se asume que la sintaxis Java es correcta, por lo que no se comprobará errores como por ejemplo que no se ponga la palabra reservada *extends* antes que la palabra reservada *class*.

Se han creado los ficheros con los ejemplos necesarios para realizar las pruebas, habiendo un conjunto que supone un ejemplo correcto y completo, y luego diferentes ficheros, cada uno con un fallo concreto, que se usan para comprobar los diferentes errores.

Como última prueba, se ejecuta el módulo en un conjunto de ficheros que contienen un patrón estado completo y correctamente marcado, comprobándose luego el contenido del XML resultante.

**6.1.2.1.1. Primera etapa** Esta etapa es independiente del patrón concreto con el que se trabaje, por lo que solo será probada en esta iteración.

Sus pruebas se dividen en aquellas generales que solo miran el marcador, y aquellas que analizan los parámetros de los marcadores independientes del patrón:

- Las pruebas relacionadas con la recogida de los ficheros pueden observarse en la tabla 6.1
- En la tabla 6.2 se recogen las pruebas relacionadas con el marcador `@pattern`.
- La tabla 6.3 contiene las pruebas sobre el marcador `@patternElement` *type*.
- El marcador `@patternElement` Builder se ha probado con las pruebas definidas en la tabla 6.4

**6.1.2.1.2. Segunda etapa** Esta etapa también es independiente del patrón en sí y tampoco será probada en iteraciones posteriores. Se comprueba que los ficheros se agrupan correctamente, y sus pruebas se recogen en la tabla 6.5.

**6.1.2.1.3. Tercera etapa** Esta etapa depende completamente del patrón concreto con el que se trabaje, en este caso el patrón Estado, por lo que las pruebas se centrarán en comprobar las relaciones entre las clases con el mismo identificador, así como la validez de la información del patrón como puede ser que no haya sido definido un contexto. Se ha dividido en tres sub-apartados:

- En la tabla 6.6 se recogen las pruebas sobre el marcador `@patternElement` *type*.
- En la tabla 6.7 se recogen las pruebas sobre el marcador `@patternAction` Transition.
- La validación de la máquina de estados se ha recogido en la tabla 6.8.

Además, se ha llevado a cabo una comprobación de la generación del XML comparando el generado con uno creado a mano que contiene el resultado esperado. Para ello se ha usado la librería XMLUnit [24], buscando que el resultado de la comparación sea "similar" (it1_ft48).

Por último, se realiza una ejecución completa del algoritmo llamando a "Analyzer.execute (sourceDirectory, targetFile);" (it1_ft49).

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft01 | `@pattern` y `@patternElement` *type* | OK |
| it1_ft02 | Fichero sin marcadores | PatternFile con *isMarked () == false* |
| it1_ft03 | Fichero sin marcadores antes de la cabecera de la clase, pero si despues | PatternFile con *isMarked () == false* |
| it1_ft04 | Más de 3 marcadores antes de la cabecera de la clase | Error: *Demasiados marcadores en la cabecera* |
| it1_ft05 | Menos de 2 marcadores antes de la cabecera de la clase | Error: *Insuficientes marcadores en la cabecera* |
| it1_ft06 | Primer marcador no es `@pattern` | Error: *El primer marcador ha de ser* `@pattern` |
| it1_ft07 | Segundo marcador no es `@patternElement` | Error: *El segundo marcador ha de ser* `@patternElement` |
| it1_ft08 | Tercer marcador no es `@patternElement` | Error: *El tercer marcador ha de ser* `@patternElement` |

Cuadro 6.1: It 1 - Pruebas - Primera etapa Generales

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft09 | Sin parámetros | Error: `@pattern` *ha de ir seguido de un "type" válido* |
| it1_ft10 | Con más de dos parámetros | Error: `@pattern` *solo puede ir seguido de un "type" y un identificador* |
| it1_ft11 | Primer parámetro no reconocido | Error: `@pattern` *con "type" no reconocido* |
| it1_ft12 | Segundo parámetro no entre < y > | Error: *el identificador ha de ir entre < y >* |
| it1_ft13 | Segundo parámetro con caracteres especiales | Error: *El identificador no puede contener caracteres especiales* |

Cuadro 6.2: It 1 - Pruebas - Primera etapa `@pattern`

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft14 | Sin parámetros | Error: `@patternElement` *ha de ir seguido de un "type" válido* |

Cuadro 6.3: It 1 - Pruebas - Primera etapa `@patternElement` *type*

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft15 | Sin parámetros adicionales | Error: `@patternElement Builder` *ha de ir seguido de un ejemplo de instanciación entre < y >* |
| it1_ft16 | Con más de dos parámetros adicionales | Error: `@patternElement Builder` *solo puede ir seguido de un ejemplo de instanciación entre < y >* |
| it1_ft17 | Segundo parámetro no entre < y > | Error: *El ejemplo de instanciación en* `@patternElement Builder` *ha de ir entre < y >* |

Cuadro 6.4: It 1 - Pruebas - Primera etapa `@patternElement Builder`

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft18 | Un fichero con marcadores queda sin agrupar | Error: *El identificador de un fichero marcado no coincide con ninguno de los patrones detectados* |
| it1_ft19 | Un grupo queda sin más ficheros que el general | Error: *Un patrón detectado solo cuenta con la clase principal* |
| it1_ft20 | Un fichero contiene más de un identificador activo entre el extends y los implements | Error: *Un fichero coincide con más de un patrón detectado* |
| it1_ft21 | Un fichero tiene un `@patternElement` *type* diferente al del general de su grupo | Error: *Un fichero esta definidio como un patrón diferente al que corresponde a su identificador* |
| it1_ft55 | Para un mismo identificador, hay mas de un fichero anotado como cabecera (por ejemplo `@patternElements State` para el patrón estado) | Error: *Se ha anotado más de una clase cabecera para el mimso identificador* |

Cuadro 6.5: It 1 - Pruebas - Segunda etapa

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft22 | Todo correcto | OK |
| it1_ft23 | `@patternElement` State sin métodos *public* o *protected* | Error: *la interfaz Estado no contiene acciones* |
| it1_ft24 | `@pattern` State pero `@patternElement` *type* distinto de State, Context y ConcreteState | Error: *"type" no válido* |
| it1_ft25 | Más de un fichero con `@patternElement` Context en un grupo | Error: *cada patrón estado solo puede tener un contexto asociado* |
| it1_ft26 | Un grupo sin ningún fichero con `@patternElement` Context | Error: *el patrón estado debe tener un contexto asociado* |
| it1_ft27 | Un grupo sin ningún `@patternElement` ConcreteState | Error: *el patrón estado debe tener estados concretos entre los que transitar* |
| h1003 it1_ft28 | `@patternElement` State con más de un parámetro adicional | Error: `@patternElement State` *solo puede ir seguido de "orderdep" o "nonorderdep"* |
| it1_ft29 | `@patternElement` State con parámetro adicional diferente de *orderdep* o *nonorderdep* | Error: `@patternElement State` *solo puede ir seguido de "orderdep" o "nonorderdep"* |
| it1_ft30 | `@patternElement` ConcreteState con mas de dos parámetros adicionales | Error: `@patternElement ConcreteState` *solo puede ir seguido de "initial" y/o "final"* |
| it1_ft31 | `@patternElement` ConcreteState con parámetros adicionales distintos de *initial* o *final* | Error: `@patternElement ConcreteState` *solo puede ir seguido de "initial" y/o "final"* |
| it1_ft32 | `@patternElement ConcreteState` con parámetros adicionales los dos *initial* | Error: *parámetro "initial" repetido* |
| it1_ft33 | `@patternElement ConcreteState` con parámetros adicionales los dos *final* | Error: *parámetro "final" repetido* |
| it1_ft34 | `@patternElement Context` con parámetros adicionales | Error: *patternElement Context no puede tener parámetros adicionales* |

Cuadro 6.6: It 1 - Pruebas - Estado - patternElement *type*

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft35 | `@patternAction Transition` con menos de un parámetro adicional | Error: `@patternAction Transition` *debe ir seguido por lo menos del estado al que transita* |
| it1_ft26 | `@patternAction Transition` con más de dos parámetros adicionales | Error: `@patternAction Transition` *solo puede ir seguido del estado al que transita y un ejemplo de los parámetros* |
| it1_ft37 | `@patternAction` con parámetro distinto de Transition | Error: *El patrón estado solo admite la acción Transición* |
| it1_ft38 | `@patternAction Transition` con segundo parámetro adicional no entre < y > | Error: `@patternAction Transition` *debe contener el ejemplo de parámetros a usar entre < y >* |
| it1_ft39 | `@patternAction Transition` cuyo segundo parámetro adicional no contiene "context" | Error: `@patternAction Transition` *debe contener "context" entre sus parámetros* |
| it1_ft40 | `@patternAction Transition` cuyo segundo parámetro adicional contiene más de un "context" | Error: `@patternAction Transition` *solo puede contener un "context" entre sus parámetros* |

Cuadro 6.7: It 1 - Pruebas - Estado - patternAction Transition

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft41 | No hay estado inicial definido | Error: *el patrón estado debe tener estado inicial* |
| it1_ft42 | Hay más de un estado inicial definidos | Error: *el patrón estado solo peude tener un estado inicial* |
| it1_ft43 | Hay al menos un `@patternElement ConcreteState` no alcanzable desde el estado inicial | Aviso: *hay estados no alcanzables* |
| it1_ft44 | `@patternAction Transition` con primer parámetro adicional no coincidente con el nombre de ningun `@patternElement ConcreteState` | Error: *no se reconoce el estado al que ha de transitar la acción* |
| it1_ft45 | Se define una transición dos veces exactamente igual (mismos estados de origen y destino, misma acción y mismos parámetros) | Aviso: `@patternAction Transition` *repetida* |
| it1_ft46 | Se define un `@patterAction` sobre un método que no coincide con ningún método *public* o *protected* del `@patternElement State` | Error: `@patternAction Transition` *definido sobre un método que no es una acción del Estado genérico* |
| it1_ft47 | Una acción del `@patternElement State` no tiene asociado ningún `@patterAction Transition` | Error: *Ningún* `@patternAction Transition` *definido sobre una acción del Estado genérico* |

Cuadro 6.8: It 1 - Pruebas - Estado - Validación máquina de estados

#### 6.1.2.2. Módulo de generación de código

Para la validación de este módulo, se han realizado pruebas unitarias de la parte de lectura y parseo del XML a la representación interna, así como de que la generación del código resultado no genera errores. Que el código resultante sea compilable y detecte los errores que esta diseñado a detectar, se ha dejado como prueba de validación del sistema. Las pruebas de este módulo se describen en la tabla 6.9. Puesto que se supone que la entrada de este módulo ha sido producida por el primer módulo, el cual debe haber sido probado que produce una salida acorde a la definición del XML, no se hacen comprobaciones sobre la validez de éste, y por lo tanto solo se prueba que el método da el resultado esperado.

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it1_ft48 | Parseo correcto de XML a la representación interna por la clase "StatePatternBuilder" | OK |
| it1_ft49 | Completar todos los datos de la representación interna como constructor vacio de una clase si este se omite en el XML | OK |
| it1_ft50 | Devuelve todos los imports referentes a partes de la implementación del patrón | OK |
| it1_ft51 | Ejecución del método "getPatternTests" de "StatePatternTestsPFE1" | OK |
| it1_ft52 | Ejecución del método "getPatternTests" de "StatePatternTestsPFE2" | OK |
| it1_ft53 | Ejecución del método "getPatternTests" de "StatePatternTestsPNFE1" | OK |
| it1_ft54 | Ejecución del método principal del módulo | OK |

Cuadro 6.9: It 1 - Pruebas - Estado - Subsistema de generación de código

#### 6.1.2.3. Prueba de integración y validación

Para la prueba de integración y validación se han implementado en un proyecto unos ejemplos del patrón estado con el fin de poder anotarlos y ejecutar el sistema sobre ellos. "State1", "State2" y "State3" son los tres estados que implementan la interfaz. "action1", "action2" y "action3" son los métodos públicos del Estado genérico. Las implementaciones y sus anotaciones han sido realizadas pensando

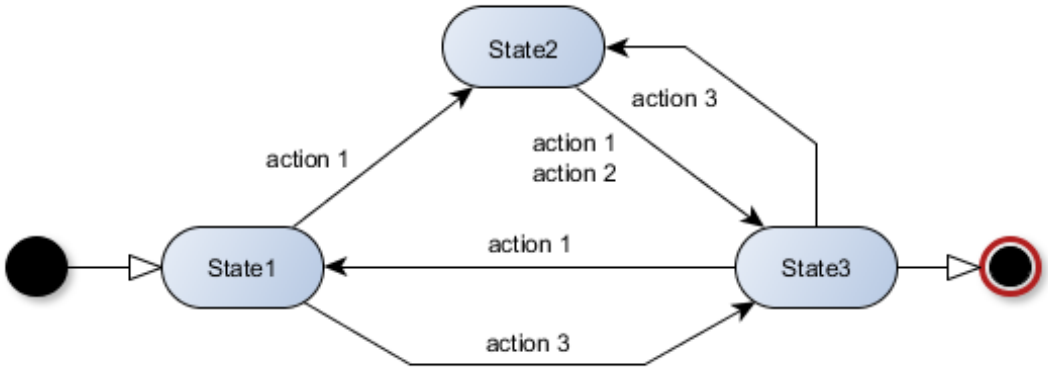


Figura 6.1: Ejemplo básico usado para validación

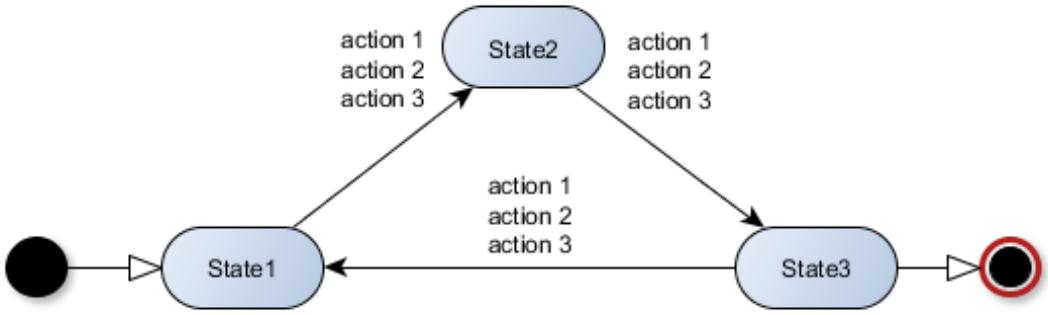


Figura 6.2: Ejemplo no dependiente del orden usado para validación

en que, para las pruebas que no requieren ser completadas por el usuario, se dé al menos un caso de prueba fallida (no encuentra el error correspondiente) y otro de prueba satisfactoria (encuentra el error correspondiente). En las figuras 6.1, 6.2 y 6.3 se muestran los diagramas de estados correspondientes a dichos ejemplos para pruebas. En todos los casos,"State1" es el estado inicial y "State3" es el estado final. En el diagrama 6.4 se muestra los cambios efectuados en la implementación de las transiciones respecto al patrón base, cambios los cuales no se han realizado en las anotaciones de las transiciones para detectar como errores en las pruebas.

La prueba se basa en llamar a la función "execute(sourcePath, resultPath, resultPackage)" de la clase "PatternTestGenerator" que es donde se encuentra el Main del subproyecto "PatternTestGenerator", indicando al sistema dónde están

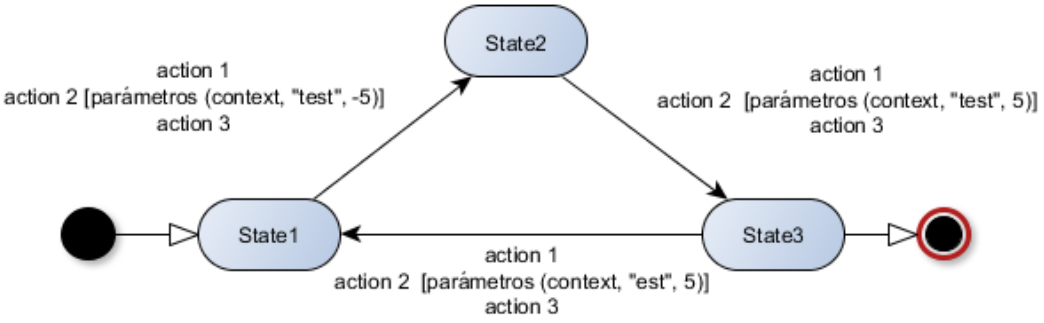


Figura 6.3: Ejemplo dependiente del orden usado para validación

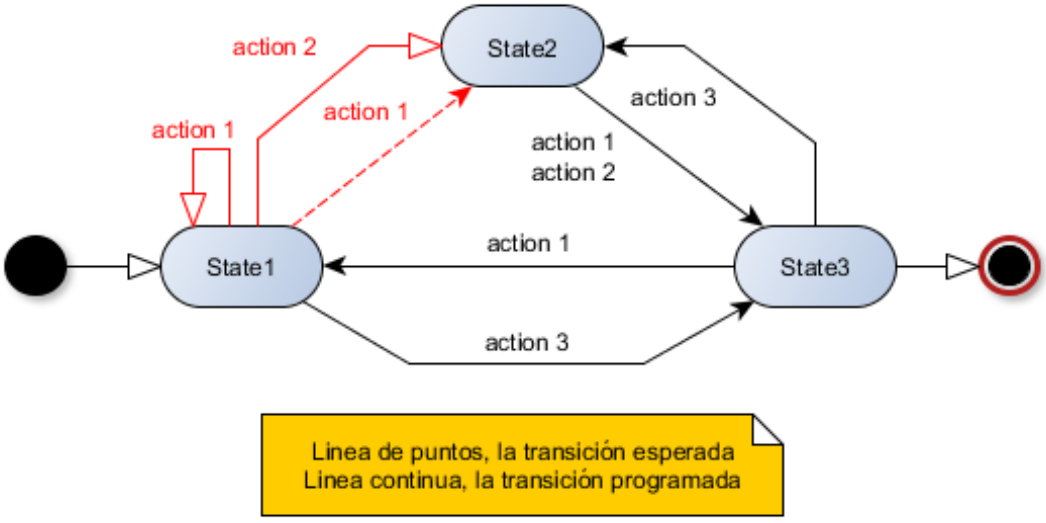


Figura 6.4: Ejemplo con transiciones erróneas usado para validación

las clases anotadas, donde ha de generar las pruebas según la estructura Maven y el "Package" que les ha de poner. Después de esto, se ejecuta el comando

```
mvn test -fae - f ../PatternImplementationExamples
```

desde el raíz del proyecto Maven con los ejemplos de implementación, capturando la salida de dicho comando, la cual se analiza buscando la existencia del mensaje

```
Tests run: 66, Failures: 5, Errors: 6, Skipped: 0
```

para comprobar que fallan el número esperado de pruebas, y si eso es correcto se buscan los mensajes de los errores y fallos concretos, que siguen el formato *Nombre del test(Nombre completo de la clase de test)* para los fallos y *Nombre del test(Nombre completo de la clase de test): Causa del error* para los errores como por ejemplo:

```
orderDependencyTest(tfg.sw.test.integration.pattern.state.
ExampleOrederDependentWrongStatePatternOrderDependentTest)
```

y

```
testTransitionSecuenceToFinalState(tfg.sw.test.integration.pattern.
state.ExampleCorrectStatePatternAceptedTransitionSecuenceTest):
The user must finish this test. This exception must be removed
after that.
```

De cara a poder llevar a cabo esta prueba, hay que asegurarse que Maven se encuentra en el PATH del sistema operativo para poder ejecutar el comando desde terminal (para windows http://www.mkyong.com/maven/how-to-install-maven-in-windows/).

## 6.2. Iteración 2: Estrategia

Vamos a presentar a continuación los elementos reseñables de los procesos de implementación y validación de la segunda de las iteraciones en que se ha dividido este proyecto, los cuales se realizaron siguiendo lo establecido en el apartado de análisis y diseño 5.2.

### 6.2.1. Implementación

Debido al cumplimiento del requisito no funcional referente a la reutilización y facilidad de ampliación del sistema (RMNF-02), en esta iteración no ha surgido ningún imprevisto en la implementación, pues como se ha usado la primera iteración como guía, se han evitado todos los problemas que surgieron en ella.

### 6.2.2. Validación

Para la identificación de las diferentes pruebas realizadas, se han seguido las directrices de la sección 2.10.

#### 6.2.2.1. Subsistema de recogida de información

En esta iteración solo se ha probado lo referente al nuevo *parser* y el *writer* creados para el patrón Estrategia, pues el resto del sistema común con la primera iteración, y por lo tanto ya ha sido validado. Se ha dividido en los siguientes sub-apartados:

- Los marcadores de la cabecera se han probado según lo descrito en la tabla 6.10.
- Las pruebas sobre el marcador `@patternElement Strategy` se recogen la tabla 6.11.

- Las pruebas sobre el marcador `@patternElement ConcreteStrategy` se recogen la tabla 6.12.
- Las pruebas sobre el marcador `@patternElement Action` se recogen la tabla 6.13.
- Las pruebas sobre el marcador `@patternElement Comparison` se recogen la tabla 6.14.
- Las pruebas sobre el marcador `@patternElement Execution` se recogen la tabla 6.15.
- La validación de la máquina de los elementos del patrón se recoge en la tabla 6.16.

De la misma manera que en la anterior iteración, se ha llevado a cabo una comprobación de la generación del XML comparando el generado con uno creado a mano que contiene el resultado esperado usado la librería XMLUnit (it2_ft23).

Por último, se realiza una ejecución completa del algoritmo llamando a "Analyzer.execute (sourceDirectory, targetFile);" (it2_ft24).

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it2_ft01 | Todo correcto | OK |
| it2_ft02 | `@pattern Strategy` pero `@patternElement` *type* distinto de Strategy o ConcreteStrategy | Error: *"type" no válido* |
| it2_ft03 | `@patternElement` Strategy con tres marcadores en la cabecera | Error: *Demasiados marcadores en la cabecera* |

Cuadro 6.10: It 2 - Pruebas - Strategy pattern parser - Cabecera

#### 6.2.2.2. Subsistema de generación de código

La validación de este módulo se ha llevado a cabo mediante una prueba de parseo de XML correcta por los mismos motivos que en la iteración 1 (la entrada es la salida del módulo 1, que ya está probado). De la misma forma, con respecto a las pruebas resultantes solo se ha probado que se generan, dejando su ejecución y

| Código | SITUACIÓN | RESULTADO ESPERADO |
|---|---|---|
| it2_ft04 | `@patternElement Strategy` que, dentro de la clase, tiene algo con un marcador diferente de `@patternElement` | Error: *La clase anotada como Strategy solo puede contener marcadores patternElement* |
| it2_ft05 | `@patternElement Strategy` que, de entre el resto de bloques de marcadores de la clase, hay alguno que no empieza con `@patternElement Action` | Error: *La clase anotada como Strategy solo puede contener marcadores de acciones* |

Cuadro 6.11: It 2 - Pruebas - `@patternElement Strategy`

| Código | SITUACIÓN | RESULTADO ESPERADO |
|---|---|---|
| it2_ft06 | `@patternElement ConcreteStrategy` cuyo fichero tiene más de un grupo de marcadores | Error: *Marcadores extra. Las estrategias concretas solo precisan los marcadores de la cabecera* |

Cuadro 6.12: It 2 - Pruebas - `@patternElement ConcreteStrategy`

comprobación de que detectan los errores a las pruebas de validación e integración del sistema completo. Estas pruebas se describen en la tabla 6.17.

#### 6.2.2.3. Prueba de integración y validación

Se han ampliado las pruebas de validación implementadas en la primera iteración, añadiendo los ficheros de código de un ejemplo del patrón Estrategia que contiene todos los casos posibles, y se han sumado los mensajes de error y fallo provocados a los que ya había del patrón Estado, así como el recuento total de pruebas ejecutadas, no superadas y erróneas (cf. sección 2.3.3), con lo que al ejecutarse el proyecto de pruebas de integración, se ejecuta el sistema sobre un directorio con múltiples implementaciones de dos patrones, entre los cuales hay dos patrones diferentes, comprobándose así la robustez de la parte común del sistema ante la ampliación de funcionalidades para soportar más patrones.

| Código | SITUACIÓN | RESULTADO ESPERADO |
|---|---|---|
| it2_ft07 | Existe más de un `@patternElement Action` en un mismo bloque | Error: *Solo puede haber un* `@patterElement Action` *en el mismo bloque de marcadores* |
| it2_ft08 | `@patternElement Action` con más de un parámetro | Error: `@patterElement Action` *No puede tener ningún parámetro extra* |

Cuadro 6.13: It 2 - Pruebas - `@patternElement Action`

## 6.3. Iteración 3: Observador

Esta iteración no ha podido ser abordada por falta de tiempo.

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it2_ft09 | Existe más de un `@patternElement Comaprison` en un mismo bloque | Error: *Solo puede haber un tipo de comparación definido por acción* |
| it2_ft10 | `@patternElement Comaprison` sin más parámetros adicionales | Error: *Ha de contener almenos un tipo de entre Equals, Comparable o Comparator* |
| it2_ft11 | `@patternElement Comaprison` *type* con type diferente de Equals, Comparable o Comparator | Error: *Solo Equals, Comparable y Comparator son reconocidos como comparadores válidos* |
| it2_ft12 | `@patternElement Comaprison Equals` con más parámetros | Error: *Equal no puede llevar más parámetros adicionales* |
| it2_ft13 | `@patternElement Comaprison Comparable` con más parámetros | Error: *Comparable no puede llevar más parámetros adicionales* |
| it2_ft14 | `@patternElement Comaprison Comparator` con solo un parámetro extra | Error: *Comparator solo puede ir con ninguno, dos o tres parámetros adicionales* |
| it2_ft15 | `@patternElement Comaprison Comparator` con más de tres parámetros extra | Error: *Comparator solo puede ir con ninguno, dos o tres parámetros adicionales* |
| it2_ft16 | `@patternElement Comaprison Comparator` con primer parámetro extra no entre < y > | Error: *Los parámetros adicionales de Comparator han de ir entre < y >* |
| it2_ft17 | `@patternElement Comaprison Comparator` con segundo parámetro extra no entre < y > | Error: *Los parámetros adicionales de Comparator han de ir entre < y >* |
| it2_ft18 | `@patternElement Comaprison Comparator` con tercer parámetro extra no entre < y > | Error: *Los parámetros adicionales de Comparator han de ir entre < y >* |

Cuadro 6.14: It 2 - Pruebas - `@patternElement Comparison`

| **Código** | **SITUACIÓN** | **RESULTADO ESPERADO** |
|---|---|---|
| it2_ft19 | `@patternElement Execution` sin parámetros adicionales | Error: *Debe ir seguido de una lista de valores entre < y >* |
| it2_ft20 | `@patternElement Execution` con más de un parámetro adicional | Error: *No puede tener más de un parámetro extra* |
| it2_ft21 | `@patternElement Execution` con parámetro adicional no entre < y > | Error: *La lista de valores ha de ir entre < y >* |

Cuadro 6.15: It 2 - Pruebas - `@patternElement Execution`

| Código | SITUACIÓN | RESULTADO ESPERADO |
|---|---|---|
| it2_ft22 | Se han definido menos de dos Estrategias Concretas | No hay suficientes estrategias concretas para comparar sus resultados. Se necesitan al menos 2 |

Cuadro 6.16: It 2 - Pruebas - Validación de la información del patrón

| Código | SITUACIÓN | RESULTADO ESPERADO |
|---|---|---|
| it2_ft23 | Parseo correcto de XML a la representación interna por la clase "StrategyPatternBuilder" | OK |
| it2_ft24 | Completar todos los datos de la representación interna como constructor vacio de una clase si este se omite en el XML | OK |
| it2_ft25 | Devuelve todos los imports referentes a partes de la implementación del patrón | OK |
| it2_ft26 | Ejecución del método "getPatternTests" de "StrategyPatternTestsPFEST1" | OK |

Cuadro 6.17: It 2 - Pruebas - Estrategia - Subsistema de generación de código

# Capítulo 7

# Seguimiento y planificación

## Índice general



En este capítulo de la memoria se describe la última etapa del ciclo de vida en espiral, en la que se hace un seguimiento de la iteración en curso, y se planifica la siguiente. A este efecto, el capítulo se divide en una sección para seguimiento y otra de planificación para cada iteración, finalizando en una sección de seguimiento global llamada "Estado final".

En los apartados de seguimiento se muestra el diagrama de Gantt con la desviación en esfuerzo y se expresa la desviación temporal que se haya sufrido.

En los apartados de planificación se decide qué requisitos se van a desarrollar en la siguiente iteración, y se planifica, reflejando dicha planificación en otro diagrama de Gantt.

## 7.1. Iteración 1: Seguimiento

La primera iteración se terminó el 21 de diciembre con un esfuerzo de 229 horas × persona frente a la planificación inicial que era terminar el 30 de noviembre tras dedicar 198 horas × persona. Esto supone una desviación de 21 días, lo cual es preocupante debido a que la fecha de entrega es fija, y de 21 horas × persona, lo cual supone un 10'6 % respecto a la estimación inicial, lo cual está dentro del margen aceptable (ver figura 7.1).

## 7.2. Iteración 1: Planificación de la iteración 2

En siguiente iteración (la segunda) se va a tratar el requisito de *Soporte para el patrón Estrategia* (RMF-08), teniendo que cumplirse en la ampliación los requisitos comunes de cada patrón (RMF-03, RMF-04, RMF-05 y RMF-06). Si el requisito de *Facilidad de ampliación de funcionalidades* (RMNF-02) ha sido cumplido con éxito en esta iteración, el esfuerzo requerido para cumplir los requisitos comunes en la nueva iteración debiera ser muy inferior al requerido en esta iteración, por lo que han sido estimados como esfuerzo medio.

El requisito adicional de *Soporte para internacionalización* (RAF-01) se mantiene durante la nueva iteración por el mismo motivo que se incluyó en esta iteración (esfuerzo de implementación menor si se hace desde etapa temprana y simplicidad de mantenimiento de las pruebas unitarias).

La división de la iteración en tareas y su estimación de esfuerzo puede observarse en el diagrama de Gantt de la figura 7.2.

Debido al desvío en tiempo sufrido durante esta iteración, las fechas de la segunda iteración ha de ser adaptadas respecto a la planificación temporal original, empezándose el día 21 de diciembre y esperando su final para el 21 de enero, manteniendo la planificación temporal de ésta en un mes de duración.

| | Nombre | Trabajo | Predecesores |
|---|---|---|---|
| 1 | Inicio Iteración 1 - Patrón Estado | 0 horas | |
| 2 | **Iteración 1 - Patrón Estado** | **229 horas** | **1** |
| 3 | Iteración 1 - Análisis | 7 horas | 1 |
| 4 | **Subsistema 1 - Analyzer** | **103 horas** | **3** |
| 5 | Diseño | 10 horas | 3 |
| 6 | **Implementación** | **42 horas** | **5** |
| 7 | Etapa 1 - Crowler | 5 horas | 5 |
| 8 | Etapa 2 - Agrupamiento de ficheros | 9 horas | 12 |
| 9 | Etapa 3 - Parser | 20 horas | 13 |
| 10 | Etapa 3 - Writer | 8 horas | 14 |
| 11 | **Validación** | **51 horas** | **7** |
| 12 | Etapa 1 - Crowler | 10 horas | 7 |
| 13 | Etapa 2 - Agrupamiento de ficheros | 8 horas | 8 |
| 14 | Etapa 3 - Parser | 25 horas | 9 |
| 15 | Etapa 3 - Writer | 8 horas | 10 |
| 16 | **Subsistema 2 - TestMaker** | **51 horas** | **4** |
| 17 | Diseño | 15 horas | 4 |
| 18 | **Implementación** | **22 horas** | **17** |
| 19 | Builder desde XML | 5 horas | 17 |
| 20 | PFE-1 - Transiciones | 6 horas | 24 |
| 21 | PFE-2 - Camino | 5 horas | 25 |
| 22 | PNFE-1 - Acceso concurrente | 6 horas | 26 |
| 23 | **Validación** | **14 horas** | **19** |
| 24 | Builder desde XML | 6 horas | 19 |
| 25 | PFE-1 - Transiciones | 4 horas | 20 |
| 26 | PFE-2 - Camino | 2 horas | 21 |
| 27 | PNFE-1 - Acceso concurrente | 2 horas | 22 |
| 28 | Pruebas de integración y Validación | 20 horas | 16 |
| 29 | Redacción de memoria | 34 horas | 28FF;15S |
| 30 | **Tareas adicionales** | **14 horas** | **6SS** |
| 31 | Implementaciónternacionalización | 2 horas | 6SS |
| 32 | Estudio librerías de comparación de XML | 6 horas | 14SS |
| 33 | Migración a Apache Maven | 6 horas | 4 |




Figura 7.1: Seguimiento de la primera iteración: Diagrama de Gantt

| Requisito | Descripción | Esfuerzo |
|---|---|---|
| RMF-03 | Interpretación de la información del patrón | Medio |
| RMF-04 | Representación del patrón en un formato intermedio | Bajo |
| RMF-05 | Lectura de la representación intermedia por parte del módulo de escritura de pruebas | Bajo |
| RMF-06 | Generación de código de pruebas | Medio |
| RMF-08 | Soporte para el patrón Estrategia | Medio |
| RMNF-01 | Legibilidad del código fuente | Bajo |
| RMNF-02 | Facilidad de ampliación de funcionalidades | Medio |
| RMNF-03 | Minimizar el acceso a disco | Bajo |
| RMNF-04 | Simplicidad de cara al usuario final en el sistema de marcado de código | Medio |
| RMNF-05 | Limpieza y legibilidad del código generado | Bajo |
| RAF-01 | Soporte para internacionalización | Bajo |

Cuadro 7.1: Resumen de requisitos y estimaciónes de la segunda iteración

## 7.3. Iteración 2: Seguimiento

La primera iteración se terminó el 21 de enero con un esfuerzo de 69 horas × persona cumpliendo la planificación realizada al finalizar la primera iteración en términos de tiempo (y por tanto acumulando un desvío en tiempo de 21 días) y con un desvío en cuanto a esfuerzo de solo 1 hora × persona (pues se estimó en 68 horas × persona), lo cual supone solo un 1'47 % respecto a la planificación inicial (ver figura 7.3).

## 7.4. Iteración 2: Planificación de la iteración 3

Por motivos de tiempo debido a los problemas surgidos en la primera iteración y que conllevaron un desvío de 21 días frente a la planificación original, no ha sido posible desarrollar esta iteración.

| | N□mbre | Trabaj□ | Predeces□es |
|---|---|---|---|
| 1 | **Iteración 2 - Patrón Estrategia** | **68 h□ras** | |
| 2 | Iteración 2 - Análisis | 5 h□as | |
| 3 | **Subsistema 1 - Analyzer** | **28 h□ras** | **2** |
| 4 | Diseñ□ | 8 h□as | 2 |
| 5 | **Implementación** | **10 h□ras** | **4** |
| 6 | Etapa 3 - Parser | 5 h□as | 4 |
| 7 | Etapa 3 - Writer | 5 h□as | 9 |
| 8 | **Validación** | **10 h□ras** | **6** |
| 9 | Etapa 3 - Parser | 5 h□as | 6 |
| 10 | Etapa 3 - Writer | 5 h□as | 7 |
| 11 | **Subsistema 2 - TestMaker** | **20 h□ras** | **10** |
| 12 | Diseñ□ | 5 h□as | 10 |
| 13 | **Implementación** | **10 h□ras** | **12** |
| 14 | Builder desde XML | 5 h□as | 12 |
| 15 | PFEST-1 | 5 h□as | 17 |
| 16 | **Validación** | **5 h□ras** | **14** |
| 17 | Builder desde XML | 2 h□as | 14 |
| 18 | PFEST-1 | 3 h□as | 15 |
| 19 | Pruebas de integración y Validación | 5 h□as | 18 |
| 20 | Redacción de mem□ia | 10 h□as | 2SS;19FF |

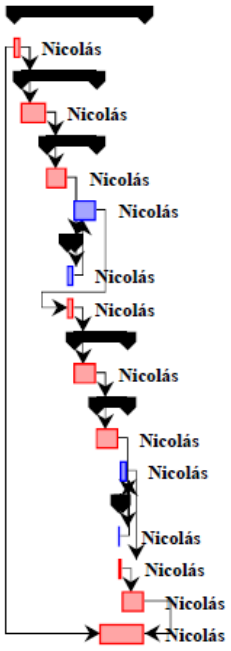


Figura 7.2: Planificación de la segunda iteración: Diagrama de Gantt

## 7.5. Estado final

El proyecto ha sufrido una gran desviación temporal como se observa en la figura 7.4, imposibilitando la entrega del sistema con el soporte para los tres patrones por ser un proyecto con fecha de entrega fijada.

En lo referente a la estimación del esfuerzo, se ha imputado un total de 303 horas × persona, contando con el desarrollo de las dos iteraciones junto con el esfuerzo de terminar la documentación (la presente memoria).

No se ha incluido un diagrama de Gantt general en la memoria por problemas de espacio, pero en los anexos están detallados todos los diagramas de Gantt, de cada iteración por separado y con las dos iteraciones juntas, tanto de línea base como de seguimiento.

Bajo esas consideraciones, se ha llegado a la conclusión de que hubo un error inicial de cálculo sobre el esfuerzo necesario para desarrollar el sistema completo

| | N□mbre | Trabaj□ | Predeces□es |
|---|---|---|---|
| 1 | **Iteración 2 - Patrón Estrategia** | **69 h□ras** | |
| 2 | Iteración 2 - Análisis | 5 h□as | |
| 3 | **Subsistema 1 - Analyzer** | **26 h□ras** | **2** |
| 4 | Diseñ□ | 5 h□as | 2 |
| 5 | **Implementación** | **9 h□ras** | **4** |
| 6 | Etapa 3 - Parser | 5 h□as | 4 |
| 7 | Etapa 3 - Writer | 4 h□as | 9 |
| 8 | **Validación** | **12 h□ras** | **6** |
| 9 | Etapa 3 - Parser | 10 h□as | 6 |
| 10 | Etapa 3 - Writer | 2 h□as | 7 |
| 11 | **Subsistema 2 - TestMaker** | **17 h□ras** | **10** |
| 12 | Diseñ□ | 6 h□as | 10 |
| 13 | **Implementación** | **7 h□ras** | **12** |
| 14 | Builder desde XML | 2 h□as | 12 |
| 15 | PFEST-1 | 5 h□as | 17 |
| 16 | **Validación** | **4 h□ras** | **14** |
| 17 | Builder desde XML | 1 h□a | 14 |
| 18 | PFEST-1 | 3 h□as | 15 |
| 19 | Pruebas de integración y Validación | 6 h□as | 18 |
| 20 | Redacción de mem□ia | 15 h□as | 2SS;19FF |

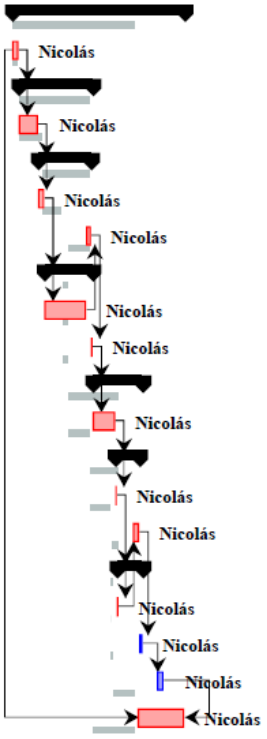


Figura 7.3: Seguimiento de la segunda iteración: Diagrama de Gantt

| | Nombre | Duración | Inicio | Fin anticipado |
|---|---|---|---|---|
| 1 | **TGR - SW** | **207 days** | **18/07/15 8:00** | **9/02/16 17:00** |
| 2 | Recogida de requisitos | 1 day | 18/07/15 8:00 | 18/07/15 17:00 |
| 3 | Iteración 1 - Patrón Estado | 162 days | 19/07/15 8:00 | 27/12/15 17:00 |
| 4 | Iteración 2 - Patrón Estrategia | 31 days | 28/12/15 8:00 | 27/01/16 17:00 |
| 5 | Memoria: Ajustes finales | 9 days | 1/02/16 8:00 | 9/02/16 17:00 |

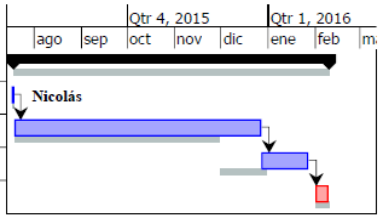


Figura 7.4: Seguimiento de la planificación general: Diagrama de Gantt

(infravalorándolo en 70 horas × persona si tomamos como referencia el esfuerzo invertido en la segunda iteración), pero no en cuanto a la estimación temporal necesaria para llevar a cabo el esfuerzo estimado, pese a la disponibilidad inconstante del recurso personal principal (el alumno).

### 7.5.1. Coste final

Al final el proyecto se desarrolló en 303 horas × persona, lo cual incurriría en un coste final real de 9090€, suponiendo un sobrecoste de 90€ respecto al coste inicial a ofertar calculado.

No obstante, y puesto que el sistema no alcanzó todos los objetivos planificados, se considera necesaria una reducción de dicho coste final a ofertar, por lo que puesto que al sistema se estima que le quedan 70 horas × persona para alcanzar dichos objetivos, y eso supone algo menos de 1/4 parte del esfuerzo realizado y algo menos de 1/5 parte respecto al esfuerzo total estimado en este punto, se considera adecuada una reducción de 1/5 (1800€) del precio final más una penalización de 500€, quedando 6700€ y suponiendo una diferencia final de 2300€ respecto al presupuesto inicial.

# Capítulo 8

# Conclusiones

## Índice general



En este apartado se presenta el cumplimiento de objetivos iniciales, junto con las conclusiones extraídas y las lecciones aprendidas de los distintos problemas surgidos durante el desarrollo del proyecto.

## 8.1. Consecución de objetivos

En general, se considera que el resultado del proyecto es satisfactorio, ya que ha dado lugar a un producto que permite generar pruebas relevantes y necesarias de una manera sencilla y efectiva con un mínimo esfuerzo por parte del usuario, ya que para las pruebas básicas de los patrones soportados solo ha de poner las anotaciones en su código y para las pruebas más avanzadas la intervención adicional requerida es mínima (decidir una secuencia de acciones en PFE-2) o inevitable (definir una forma de comparar los resultados de las estrategias en PFEST-1).

Por ello se cree que el uso del producto conseguido ayudará a fomentar el uso consciente e intencionado de patrones al añadir a los beneficios tradicionales sobre proceso de diseño, los beneficios adicionales de la obtención automática de implementación de pruebas unitarias significativas.

Más en concreto, al final del plazo del proyecto se han cumplido la mayor parte de los objetivos marcados, incluyendo los más fundamentales, consistentes en el desarrollo del sistema base, la prueba de su facilidad de ampliación para soportar nuevos patrones y la capacidad de detección de errores de las pruebas diseñadas. No obstante, por problemas con la planificación (esfuerzo subestimado), no se ha conseguido completar la última iteración, consistente en extender la funcionalidad para dar soporte al patrón Observador.

Dando una visión detallada objetivo por objetivo, podemos decir:

**Objetivo-1:** *Que el sistema sea escalable, de forma que añadir el soporte a nuevos patrones suponga el menor esfuerzo posible.*

Se considera cumplido este objetivo al haberse implementado soporte para un segundo patrón, comprobando que dicha extensión no implicó modificaciones significativas en el sistema base ya implementado.

Una prueba empírica de esta afirmación se obtiene al comparar el esfuerzo de desarrollo de la primera iteración con la segunda, en la cual, gracias a estar basada en la primera, se necesito un esfuerzo muchísimo menor.

**Objetivo-2:** *Analizar el patrón Estado y diseñar un conjunto de pruebas genéricas, independientes y de generación lo más automatizada posible.*

Este objetivo se considera cumplido al ejecutarse las pruebas de aceptación sobre los ejemplos preparados, de manera que éstas detectan los errores introducidos intencionalmente a este propósito y no arrojan falsos positivos.

**Objetivo-3:** *Implementar el soporte del sistema para el patrón Estado.*

Se da por satisfecho este objetivo con el código implementado que da soporte al patrón Estado, en el cual no se han conseguido detectar fallos con las pruebas unitarias realizadas pese al esfuerzo dedicado en ellas.

**Objetivo-4:** *Analizar el patrón Estrategia y diseñar un conjunto de pruebas genéricas, independientes y de generación lo más automatizada posible.*

Al igual que en el **Objetivo-2**, este objetivo se considera cumplido con las pruebas de aceptación.

**Objetivo-5:** *Implementar el soporte del sistema para el patrón Estrategia.*

De la misma manera que con el *Objetivo-3*, al no ser capaces de detectar fallos con las pruebas unitarias realizadas, se considera cumplido el objetivo.

**Objetivo-6:** *Analizar el patrón Observador y diseñar un conjunto de pruebas genéricas, independientes y de generación lo más automatizada posible.*

Por falta de tiempo y debido a los desvíos sufridos durante la primera iteración, este objetivo no ha podido cumplirse.

**Objetivo-7:** *Implementar el soporte del sistema para el patrón Observador.*

Sin el análisis pertinente, la implementación del soporte para el patrón Observador no ha sido llevada a cabo.

En resumen, se han alcanzado todos los objetivos a excepción de los referentes al patrón Observador (tercera iteración). Aun así, puesto que el objetivo principal de obtener un sistema robusto y escalable ha sido alcanzado, se ha conseguido implementar el soporte para dos de tres patrones y el esfuerzo total ha sido el esperado para el proyecto completo, se considera que el error está en la estimación y planificación de la primera iteración, habiendo sido demasiado optimista.

## 8.2. Lecciones aprendidas

Una vez diseñado el sistema de representación intermedia por XML, al volver la vista atrás se ha llegado a la conclusión de que no fue la mejor decisión existiendo la alternativa de JSON, pues al final la estructura interna de representación de los patrones en los dos módulos ha resultado ser prácticamente idéntica, por lo que una librería de JSON podría haber serializado y deserializado hacia y desde JSON directamente. Cuando se llegó a esta conclusión, la primera iteración ya

estaba terminada y el coste de modificar la primera iteración y hacer la segunda ya con JSON (unas 30 horas × persona) habría sido más grande que continuar con el sistema actual e implementar la segunda iteración con XML usando la estructura de la primera como guía (unas 10 horas × persona), pues si bien no habría que diseñar el lenguaje XML, sí habría que documentar y especificar minuciosamente las clases a serializar, obligando a terceros que quisieran ampliar el sistema a adaptarse a dicha estructura de clases y a hacer otra con la que trabajar si quieren añadirle campos diferentes a los que se han de serializar o que te ofrece la deserialización. Por lo tanto, ya que la desviación en tiempo y esfuerzo ya era importante y no suponía realmente un problema a la hora de continuar el proyecto, se continuó con el sistema existente.

Un error grave fue el cometido a la hora de estimar el esfuerzo y tiempo totales del proyecto durante la planificación inicial, habiendo dado lugar a una mala estimación, como se ha hecho patente al no haberse podido desarrollar la tercera iteración por el desvío temporal, y aun y así haber llegado a la cantidad de esfuerzo estimada inicialmente para el proyecto completo, con una desviación acumulada en las dos iteración implementadas de 22 horas × persona. Incluso aunque no hubiera habido el desvío temporal, en base al esfuerzo realizado en la segunda iteración, de haber desarrollado la tercera iteración habría conllevado otras 60 horas × persona por lo menos, suponiendo un desvío en esfuerzo de 63 horas × persona, lo que supone un 20 % de la planificación inicial.

## 8.3. Trabajo futuro

Como trabajo futuro se propone, en primera instancia, la ampliación del sistema para soportar más patrones de un lenguaje ya soportado usando lo ya implementado como guía. Se estima, fijándose en el coste de esfuerzo de la segunda iteración (69 horas × persona), que ampliar para un nuevo patrón pueda llevar entre 50 y 90 horas × persona en función de la familiaridad del desarrollador con el proyecto y la dificultad técnica de análisis del patrón y diseño de sus pruebas.

En segunda instancia se propone la ampliación para soportar diferentes lenguajes. Esto repercutiría en la necesidad de ampliar la parte común de forma que sepa procesar diferentes lenguajes, para los cuales los marcadores pueden o no identificarse de la misma forma ( dentro de comentarios de bloque actualmente), pero no supondría la necesidad, una vez procesados los marcadores a la representación interna, de modificar la forma de trabajo de los parsers concretos, siendo posible incluso la reutilización completa o casi completa con poco esfuerzo pues no trabajan con el fichero de código sino con los marcadores ya procesados, la información de los cuales debiera ser igual o muy similar (por ejemplo, puede variar cómo se pasan los parámetros a un método, pero al trabar con patrones para la orientación a objetos, no es habitual). Por ello, en función del grado de reutilización posible y de la familiaridad del desarrollador con el proyecto, se estima que añadir un nuevo lenguaje al sistema para un primer patrón costaría un esfuerzo de entre 40 y 120 horas × persona, correspondientes respectivamente a las combinaciones de "alta reutilización y familiaridad" y de "nula reutilizacion y baja familiaridad".

Como tercera mejora, se podría extender el sistema dando soporte a la actualización del código de pruebas generado en lugar de ser completamente recreado cada vez que se produzcan cambios en la implementación, con lo que en lugar de regenerar completamente, se analizarían los cambios necesarios poder para mantener los posibles cambios manuales que el usuario hubiera podido realizar en dichas pruebas. Esto solucionaría situaciones como por ejemplo, cuando el usuario ha completado la prueba de PFE-2 y vuelve a ejecutar el sistema sobre las mismas clases con la misma ruta de salida, tendría que volver a completarlo.

Además, en consonancia con lo dicho en el apartado anterior de lecciones aprendidas, una modificación recomendable –pero significativamente costosa– de cara a facilitar el trabajo futuro es la modificación del sistema existente sustituyendo la tecnología XML por JSON. Con esto, al ampliar el sistema con soporte para nuevos patrones se reduciría el esfuerzo de generar la estructura intermedia de unas 10 horas × persona (que es en lo que se estima con XML) a unas 5 horas × persona, a coste de una pequeña reducción de independencia entre los dos módulos.

# Apéndices

# Apéndice A

# Manual de usuario

El sistema de generación de código puede ser ejecutado en su totalidad, o cada módulo por separado. Esto además puede hacerse desde línea de comandos o importándolo como librería en el código.

## A.1. Ejecución completa

Para ejecutar el sistema en su totalidad por línea de comando, se deberá ejecutar el jar de la aplicación pasándole como parámetros la ruta donde empezar a buscar recursivamente el código, la ruta donde se desea se depositen las pruebas generadas y bajo qué espacio de nombres han de generarse.

En la alternativa desde código, tras importar la clase `PatternTestGenerator` del espacio de nombres "`tfg.sw.patternTestGenerator.generator`", se deberá ejecutar el método estático "`+execute (sourceCodeOriginPath: String, sourceCodeResultPath: String, resultPackage: String): void`", el cual recibe los mismos parámetros en el mismo orden.

## A.2. Ejecución por separado

Para el sistema de análisis de código y generación de la representación intermedia, los parámetros son la ruta donde buscar el código a analizar y el nombre completo (ruta + nombre de fichero) que se le quiere dar al fichero XML con dicha representación. Se ejecuta bien ejecutando el jar, bien importando la clase `Analyzer` del espacio de nombres "`tfg.sw.analyzer.analyzer''` y ejecutando el método estático "`+execute (root: String, destiny: String): void`".

Para el sistema de generación de pruebas, los parámetros son el nombre completo (ruta + nombre de fichero) del fichero XML con la representación intermedia, la ruta donde depositar las pruebas generadas y el espacio de nombres bajo el que han de generarse, y se ejecuta bien ejecutando el jar, bien importando la clase `TestMaker` del espacio de nombres "`tfg.sw.maker.maker`" y ejecutando el método estático "`+execute (definitionFile: String, destinyPath: String, destinyPackage: String): void`".

# Bibliografía


[1] 8879:1986, I. (N.D.).
Standard generalized markup language.
http://www.iso.org/iso/catalogue_detail?csnumber=16387.

[2] Alexander, C. (1979).
*The Timeless Way of Building.*
Oxford University Press.

[3] Clements, P. (2010).
*Documenting Software Architectures: Views and Beyond.*
Addison-Wesley, 2 edition.

[4] Company, S. (2008).
*Maven: the definitive guide.*
O'Reilly, cop.

[5] Cunningham, W. and Beck, K. (1987).
*Using Pattern Languages for OO Programs.*
Addison-Wesley.

[6] Dice Holdings, Inc (N.D.).
Repositorio source forge.
http://sourceforge.net/home.html.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (2005).
*Design Patterns: Elements of Reusable Object-Oriented Software.*
Addison-Wesley.

[8] Farrell-Vinay, P. (2008).
*Manage software testing.*
Auerbach.

[9] Foundation, A. S. (N.D.).
Apache software foundation.
https://www.apache.org/.

[10] Free Software Foundation (N.D.a).
Compilador gnu para java.
gcc.gnu.org/java/.

[11] Free Software Foundation (N.D.b).
Gnu classpath.
http://www.gnu.org/software/classpath/.

[12] Free Software Foundation (N.D.c).
Projecto gnu.
https://www.gnu.org/.

[13] JUnit (N.D.).
Framework de tests unitarios para java.
https://github.com/junit-team/junit.

[14] Oracle (N.D.a).
Estadísticas de uso de java.
https://www.java.com/es/about/.

[15] Oracle (N.D.b).
Java.
https://www.java.com/es/.

[16] Oracle (N.D.c).
Javadoc.
http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html.

[17] RAE (2016).
Real academia española - definición de patrón.
http://dle.rae.es/?id=SBler1T.

[18] Software, P. (N.D.).
Spring framework.
https://projects.spring.io/spring-framework/.

[19] Software Freedom Conservancy (N.D.).
Git.
https://git-scm.com/.

[20] Sommerville, I. (2011).
*Ingeniería del Software*.
Addison-Wesley.

[21] The Eclipse Foundation (N.D.).
Ide eclipse.
https://eclipse.org/downloads/.

[22] W3C (N.D.).
Extensible markup language.
https://www.w3.org/TR/2006/REC-xml11-20060816/.

[23] Wikipedia (N.D.).
Wikipedia - patrones de diseño.
https://es.wikipedia.org/wiki/Patrón_de_diseño.

[24] XMLUnit (N.D.).
Framework de tests unitarios sobre xml para java y .net.
https://github.com/xmlunit/xmlunit/releases/tag/v2.0.0-alpha-03/.

[25] XUnit (N.D.).
Diseño general para frameworks de pruebas unitarias.
https://github.com/junit-team/junit.