UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

PROYECTO DE FIN DE CARRERA DE INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Realización de un Estudio Práctico del impacto producido por la aplicación del BDD a un diseño DDD siguiendo los principios SOLID y usando la plataforma de Microsoft .NET

Tomo I: Memoria

Autor: David García Chaves

Tutor: Alfonso Castro Martínez

A Coruña, septiembre de 2010

Resumen

El presente proyecto busca explicar las Prácticas de Diseño Agile asociadas al Behavior-Driven Development (BDD), partiendo de un Domain Model que sigue el Domain-Driven Design (DDD) y apoyándose en los principios de orientación a objetos S.O.L.I.D..

El trabajo se dividirá en dos partes diferenciadas. Por un lado, el software desarrollado, y por otro, el Estudio con el que documentaremos los hallazgos del mismo. Al mismo tiempo intentaremos sincronizar la evolución de ambos.

La idea que subyace aquí, es la de intentar reproducir todo el proceso de errores y aciertos que nos llevan a solucionar un problema a través de la Práctica de Diseño Agile escogida.

Para conseguir nuestros objetivos, nos valdremos del conocido mantra: "Red-Green-Refactor", que hunde sus raíces en el Test-Driven Development (TDD) del eXtreme Programming (XP), y gracias al mismo presentaremos los patrones básicos del DDD que nos permitirán estructurar las clases de una manera efectiva.

Todo ello en un ámbito en el que estableceremos una metodología basada en la micro-iteración y por tanto, el micro-diseño.

Palabras Clave

BDD, DDD, SOLID, TDD, Agile, Onion Architecture.

ÍNDICE

1 Contextualización y Objetivos

El presente proyecto tiene vocación pedagógico-divulgativa y busca explicar los principios, patrones y herramientas que deberían facilitarnos el desarrollo una aplicación siguiendo las Prácticas de Diseño Agile asociadas al BDD.

Para ello partiremos de un Domain Model conocido, el mismo que Eric Evans utilizó como ejemplo en su libro Domain-Driven Design: Tackling Complexity in the Heart of Software [1] y que utiliza los principios y patrones asociados al DDD. Este modelo, va camino de convertirse en el estándar del DDD.

A medida que vayamos avanzando en el desarrollo del proyecto, veremos como el modelo inicial que describe el sistema va siendo modificado por la práctica del BDD, llevándonos a tener que tomar decisiones, que ni por asomo, hubiésemos sospechado al comenzar.

Es precisamente por esta razón por la que se escogió el modelo de la Biblia Azul [1] de Evans, pues al ser el estándar del DDD, permitirá valorar de una forma mucho más clara el impacto que tiene la Práctica de Diseño Agile BDD, en él. Este impacto no sería el mismo si hubiésemos escogido cualquier otro modelo, por no mencionar que un dominio artificialmente creado para la ocasión, en nuestra opinión, desvirtuaría todo el Estudio.

Realizaremos varias refactorizaciones del código, apoyándonos en los principios S.O.L.I.D., enunciados por Robert C. Martin [2][3], al ser extremadamente valiosos para hacer aflorar los Design Smells del mismo.

Al mismo tiempo, se sincronizará todo el proceso de desarrollo del software con la documentación textual del proceso, de forma que podamos describir con total precisión los problemas que vayamos encontrando. No debemos

pasar por alto, el hecho de que, por mucho que intentemos separar la materialización textual de la relativa al código, la difuminación de dicha separación es algo buscado. Indirectamente, demostraríamos la simbiosis necesaria entre:

- BDD.
- Documentación.
- Código.

La idea que subyace aquí, es la de intentar reproducir todo el proceso de errores y aciertos que nos llevan a solucionar el problema a través de la Práctica Agile escogida (BDD). Se demostrará que el hecho de tomar caminos erróneos es inherente al desarrollo de software, algo a lo que no hay que temer, ya que la flexibilidad del BDD nos va a ayudar a subsanarlos.

Una segunda lectura de este proyecto, incluye la idea de que la documentación generada pueda servir de guía o tutorial para aquél que pretenda introducirse tanto al BDD como al DDD y los principios de diseño S.O.L.I.D., pero en ningún caso debe considerarse el objetivo principal.

Para poder establecer las distintas tareas necesarias para la correcta práctica del BDD, usaremos el conocido mantra: “Red-Green-Refactor” que hunde sus raíces en el TDD del XP.

Precisamente, integrado en este mantra, es como introduciremos los principios de diseño S.O.L.I.D., como antídoto ante algunos de los males comunes en el desarrollo de software.

De la misma forma, usaremos el mantra “Red-Green-Refactor” como detonante para presentar los patrones básicos del DDD, estructurando nuestras clases de una manera efectiva y dotando a nuestro software de un

significado y estructura, que nos libere a la hora de tener que diseñar.

Durante todo el desarrollo del presente proyecto se ha usado el Control de Versiones Distribuido Git. Por ello, todo el proceso asociado al presente proyecto puede ser consultado de forma pública a través del repositorio GitHub:

http://github.com/3j/dddsample

2 Fundamentos Teóricos

En este capítulo vamos a abordar las bases teóricas sobre las que sustentamos este Proyecto Fin de Carrera. Sin estos fundamentos, difícilmente se pueden comprender las intenciones del mismo.

2.1 Las Prácticas de Diseño Agile Behavior-Driven Development

2.1.1 Lean

Gran parte de las metodologías ágiles hunden sus raíces en los principios de fabricación Lean, desarrollados y perfeccionados por Toyota, y que han sido trasladados a la producción de software por unos cuantos gurús del software durante la última década del siglo pasado.

Toyota se dio cuenta de que eliminar el desperdicio (waste) era la clave para poder alcanzar sus objetivos, y que para poder lograr esto, debía implantar una rigurosa política de calidad que abarcaba tanto a los procesos como a los productos.

Por tanto, una aproximación Lean y Agile dependen directamente de la capacidad para inspeccionar y obtener feedback del producto, sea cual sea la etapa del proceso en la que se encuentre. Solo así es posible evitar el desperdicio (waste). [4]

2.1.2 El origen

XP y Scrum son dos de las metodologías ágiles más conocidas. Usadas en conjunto, aportan las dosis justas de organización y planificación junto a

prácticas de ingeniería ágiles bien definidas. Sin embargo XP y Scrum son dos metodologías diferentes. Existen agujeros entre ambas, que hacen que la transición entre la planificación de Scrum y las prácticas asociadas a la ingeniería de XP no sea todo lo confortable que debiera.

De esa fricción que se produce al aplicar XP y Scrum nace, precisamente, el Behavior-Driven Development.

El BDD acerca comunicación y feedback en la brecha abierta entre XP y Scrum, facilitándonos una nueva forma de aplicar principios antiguos.

Al interponer una capa de BDD en el binomio XP/Scrum, las prácticas Agile fluyen de forma más natural entre las diferentes actividades como análisis, diseño, desarrollo y prueba.

El BDD ayuda a armonizar las prácticas asociadas a las User Stories de Scrum con las prácticas de Test-Driven Development de XP. En esta situación, las User Stories representan análisis y especificación, mientras que el TDD representa el diseño, siempre dentro del ámbito de un proyecto Agile.

2.1.3 La conexión con el Domain-Driven Design

El BDD toma prestado del Domain-Driven Design la necesidad de establecer y evolucionar un lenguaje compartido usado por el equipo en:

- Requisitos.
- Abstracciones (Nombres de clases y variables, métodos, etc).
- Especificaciones.
- Documentación.

Este lenguaje compartido debería surgir de manera natural en las

conversaciones y comunicaciones existentes entre todos los actores implicados en el proyecto.

En DDD, este lenguaje compartido es conocido como DDD_Ubiquitous_Language.

2.1.4 Context / Specification

El BDD usa un estilo de test Context / Specification.

Usando esta práctica proveniente del TDD, podemos llegar a generar grupos de tests más pequeños, simples y enfocados que documenten el sistema.

Los tests Context / Specification se centran en el comportamiento del sistema, de sus módulos y unidades, en contextos de uso específicos, en vez de preocuparse de la implementación particular de clases y métodos.

2.1.5 La conexión con el Test-Driven Development

No hay nada en el BDD que modifique la mecánica ya impuesta por el Test-Driven Development. Todo lo que se hacía al practicar TDD se sigue haciendo de la misma manera. De hecho, pese a usar tests Context / Specification, seguiremos necesitando:

- Red-Green-Refactor.
- Arrange-Act-Assert.
- Test-First Programming.
- Minimización de la fricción.
- Diseño micro-incremental.
- Patrones.

Por lo tanto, pese a que el BDD cubre algo más de terreno que el TDD, el corazón que late en el BDD es de una clara naturaleza TDD.

Así, mientras el TDD trata, de forma especifica, el diseño de unidades de código y módulos, como puede ser una clase; el BDD, pese a preocuparse por el diseño de unidades, amplía su enfoque a otras cuestiones relativas al diseño:

- Diseño del lenguaje común.
- Diseño de los tests.
- Diseño de la experiencia de usuario.

2.1.6 Client-Driven Development y Assumption-Driven Development

Puesto que el TDD fomenta el Test-First Programming, una forma más adecuada de nombrar al Test-Driven Development es Client-Driven Development, en oposición a Assumption-Driven Development.

Al usar Client-Driven Development para desarrollar una API, debemos, antes de nada, escribir ejemplos que presentan la API en el contexto en el que va a ser usada. De esta forma estamos mostrando como nos gustaría que fuese si la hubiésemos diseñado nosotros mismos (de hecho la estamos diseñando). Podríamos decir, por tanto, que el cliente de la API es el que se encarga de su diseño, mediante estos ejemplos, que acabarán convirtiéndose en tests Context / Specification.

En el momento en el que el código de los ejemplos nos parezca correcto, procederemos con la implementación micro-incremental de la API, de tal forma que, al seguir ejecutando una y otra vez estos ejemplos, estaremos obteniendo un feedback continuo del diseño de la misma.

Por lo tanto, si diseñamos la API con TDD, la diseñamos desde la perspectiva del código que actúa como cliente de la API. El diseñador es obligado a ponerse en la piel de un usuario de la API y entonces, diseñar la experiencia de usuario de la misma, siendo este diseño un reflejo de sus necesidades e ideas sobre lo que considera una experiencia correcta y razonable.

Por el contrario, el uso de herramientas más gráficas como el diseñador de clases del Visual Studio o lenguajes extremadamente abstractos como UML, derivan inevitablemente, en un paradigma anti-Lean al generar mucho desperdicio (waste). Se produce un descenso de la precisión y un aumento de la necesidad de mejorar el código ya escrito. Este tipo de herramientas suelen ser un indicador del uso de Assumption-Driven Development.

2.1.7 La palabra Test en TDD

Uno de los errores más comunes a la hora de practicar TDD, es la creencia de que, por el simple hecho de usar tests, ya estamos practicando TDD. Nada más lejos de la realidad.

Este error viene derivado, probablemente, de que lleva la palabra Test (Test-Driven Development) en el acrónimo. Si los tests no son escritos antes del código (Test-First Programming), para que salga a la superficie y podamos eliminar la fricción presente en el diseño creando una experiencia de usuario positiva, no estamos practicando TDD.

De hecho, la simple presencia de Unit Tests en un proyecto, no es indicativo de que estemos ante un desarrollo que comparta prácticas Client-Driven, como el TDD. Por lo tanto, difícilmente favoreceremos una producción de software Lean.

La palabra Test en TDD es un efecto secundario de practicar Client-Driven

Development usando como herramienta un framework de Unit Testing.

2.1.8 Dan North y la palabra Behavior en BDD

El objetivo principal del TDD es diseñar y especificar los comportamientos (behaviors) del sistema según las expectativas del cliente y ser capaces, al mismo tiempo, de probar que esos comportamientos se mantienen fieles a esas expectativas durante la vida del producto.

En un esfuerzo por intentar aclarar los conceptos erróneos que provienen del uso y abuso de la palabra Test en TDD, así como para arrojar algo de luz sobre los descubrimientos que fueron surgiendo tras varios años de práctica del TDD, Dan North acuñó el término Behavior-Driven Development. De hecho, cualquier cosa que pudiese ser pensada gracias al BDD podría haber sido llevada a cabo vía TDD.

La idea que subyace, es que el comportamiento de los sistemas es lo importante, frente a la realidad de los datos que provocan ese comportamiento. Al fin y al cabo los Business Objects son objetos de comportamiento, pese a que estén asociados a unos datos concretos. Lo que los define es, precisamente, lo que hacen con esos datos, gracias a los comportamientos presentes en el sistema. Ese es el enfoque correcto.

Necesitamos que los comportamientos estén definidos correctamente, que puedan ser descritos de forma precisa, para que los otros desarrolladores que tengan que trabajar con el código puedan entenderlos sin ambigüedad.

Desde otro punto de vista. Como desarrolladores, a través del código, implementamos el sistema usando la especificación de los comportamientos; pero antes de que eso ocurra, el cliente especifica los comportamientos a través de las User Stories. Ahí tenemos el rol del BDD como puente entre

Scrum y XP.

2.1.9 Definición del comportamiento por parte del cliente

Las User Stories expresan los requisitos bajo los términos de:

- El Rol.
- El Objetivo.
- La Motivación.

Las User Stories se preocupan de la definición del Rol del usuario, por ejemplo: un comprador, un vendedor, ….

Una User Story describe el Objetivo del usuario a través de la necesidad de que el "negocio" haga algo por nosotros, en oposición al "sistema" haciendo algo por nosotros.

Finalmente, una User Story debe capturar la Motivación del usuario por la cual quiere conseguir el Objetivo.

Esta es la plantilla que nos permite especificar las User Stories:

> As a <**ROLE**> I want to <**GOAL**> so that <**MOTIVATION**>.

2.1.10 YAGNI

El TDD nos proporciona una gran reversibilidad en cuanto a las decisiones tomadas. Por lo tanto, no es necesario excederse con el código para requisitos futuros, ya que podremos ocuparnos de dicho código cuando llegue el

momento en el que ese requisito futuro se convierta en un requisito presente. En el lenguaje Agile, conocemos esta práctica como YAGNI (“You Ain't Gonna Need It”).

El principio YAGNI nos ayuda a evitar la codificación de puntos de extensión en “previsión de que ...” hasta que esa previsión se materializa en el presente (pasando a ser un requisito real), guiando nuestro código en esa dirección.

Si volvemos a desempolvar los conceptos asociados al Lean, el código que incumple YAGNI es una forma de desperdicio (waste) indeleblemente asociado al Design Smell conocido como Needless Complexity.

2.1.11 Acceptance Criteria y Test-First Programming

Cuando hablamos de Acceptance Criteria nos referimos a las especificaciones definidas en base a las User Stories, escritas en lenguaje natural. Éstas, se transforman en especificaciones de bajo nivel escritas en código, siendo ejecutadas a través de un framework especializado en tests o BDD_Specs.

Por tanto, podemos considerar las Acceptance Criteria como el punto inicial para poder practicar Test-First Development. Los desarrolladores pueden traducirlas en BDD_Specs (Specifications) para a continuación ser escritas en código e implementadas gracias al TDD.

Los miembros que forman el equipo agruparán estas observaciones (Observations) o tests en contextos (Context) según la relación de similitud de objetivos que se establezcan entre ellas. Estos Contexts, no serán arbitrarios, por tanto. Estaríamos ante representaciones de circunstancias cohesivas y consistentes, que nos ayudan a definir los distintos módulos del sistema.

Estos Context y Specifications estarán formados por Concerns, Contexts y

Observations.

Usando el ejemplo canónico del Banco para aclarar conceptos:

> **Concern**: Funds Transfer.
> **Context**: When trasfering between two accounts.
> **Observation**: The Amount is debited from the "from" account.
> **Observation**: The Amount is credited to the "to" account.

Es importante resaltar que con este proceso, no estamos intentando conseguir la correspondencia exacta y unívoca entre:

- User Stories y Acceptance Criteria
  con
- Requisitos.

Sino que buscamos conseguir la información suficiente, como para poder hacer que el trabajo y la codificación empiece.

De hecho nos encontraremos con casos en los que, gracias a la codificación de los requisitos, descubriremos aspectos que no estaban claros en las User Stories ni en las Acceptance Criteria, de forma que se producirá un feedback que ayudará a aclarar esos problemas.

Nunca debemos perder de vista que el TDD es una forma de ejecutar especificaciones. Los equipos usan TDD y Test-First Programming en BDD para poder convertir la User Story y la Acceptance Criteria en código ejemplo. Este código ejemplo especifica el código que necesita la aplicación para cumplir con los requisitos marcados por la User Story.

2.1.12 Duplicación de código en las Specifications

El código de las Specifications (BDD_Specs) tiene por objeto la documentación de los comportamientos del sistema.

Para conseguir este objetivo, muchas veces es necesario permitir la duplicación de código en las Specifications. Así conseguimos que la comprensión y el aprendizaje de la BDD_Spec, por un lado, y los conceptos y nociones que comunica, por otro, sea lo más sencillo posible.

El objetivo es crear la mejor experiencia de aprendizaje posible para cualquier desarrollador diferente al que haya escrito la BDD_Spec, y que se vea en la necesidad de adquirir conocimiento del sistema, en una primera aproximación al mismo.

Estas peculiaridades deben ser tenidas en cuenta a la hora de refactorizar el código de las BDD_Specs, en un intento de evitar la duplicidad del código de las mismas, ya que con facilidad podemos torpedear su fácil comprensión. Si no se tiene ésto en cuenta, corremos el riesgo de caer en un anti-pattern al menos tan grave como lo sería permitir esa duplicidad de código en la propia implementación de la aplicación.

2.1.13 La conveniencia del estilo Context / Specification

El estilo Context / Specification ha demostrado ser eficaz a la hora de escribir BDD_Specs sencillas y rápidas de entender.

Gracias a ellas, facilitamos la navegabilidad del código fuente y añadimos todavía más "agilidad" a la experiencia de la codificación, así como contribuimos a eliminar más deshecho (waste) en nuestro esfuerzo por

conseguir desarrollar, testar y mantener aplicaciones Lean. [4][5][6][7][8][9] [10][11][12]

2.2 Filosofía y Elementos del Domain-Driven Design

2.2.1 ¿Qué es el Domain-Driven Design?

El Domain-Driven Design [1] es una filosofía subterránea, que se ha venido gestando desde hace unos 15 años en el mundo de la Programación Orientada a Objetos. Detrás del DDD se esconde una doble premisa. Ésta establece que:

- La mayoría de los proyectos software, deberían centrar su enfoque en el dominio y la lógica que se encierra tras él.
- Los diseños de dominio más complejos deberían basarse en un modelo.

El DDD no es ni una tecnología ni una metodología. Es una manera de pensar y un conjunto de prioridades. Todo ello encaminado a acelerar los proyectos de software que tienen que hacer frente a dominios complejos.

Para lograr ese objetivo, los equipos necesitan un amplio conjunto de prácticas de diseño, técnicas y principios. Todas esas cuestiones fueron tratadas por Eric Evans en su Biblia Azul, el libro Domain-Driven Design: Tackling Complexity in the Heart of Software [1] y es a él a quien debemos el despegue definitivo del DDD.

2.2.2 Abordando la complejidad en el corazón del software

Partiendo de la base de que hay muchas cosas que pueden hacer descarrilar un proyecto (objetivos poco claros o falta de recursos, por nombrar algunos), debemos considerar la forma en que enfocamos el diseño como uno de los elementos clave a la hora de determinar, en gran medida, lo complejo que

puede llegar a convertirse el software.

Cuando la complejidad se nos escapa de las manos, el software deja de ser lo suficientemente claro como para poder ser modificado o ampliado, sin generar serios dolores de cabeza. Por el contrario, un buen diseño puede convertir la complejidad en una nueva oportunidad de mejora.

Algunos de los factores que condicionan el diseño son de naturaleza puramente tecnológica. Sin embargo, la complejidad no suele emanar del condicionamiento tecnológico, sino del dominio en sí. Es en el propio dominio donde reside la complejidad. Cuando evitamos abordar esta complejidad en el diseño, poco importa que la vertiente tecnológica de la infraestructura esté bien concebida.

Un diseño que aspire al éxito, debe ocuparse de la complejidad del dominio de forma sistemática.

2.2.3 Domain Patterns, Design Patterns, Architectural Patterns

Los Domain Patterns tienen un enfoque muy diferente a los Design Patterns o a los Architectural Patterns, puesto que su único propósito es hacer hincapié en como estructurar el Domain Model:

- Como encapsular el conocimiento del dominio en el modelo.
- Como aplicar el DDD_Ubiquitous_Language.
- Como evitar que la infraestructura nos distraiga de lo verdaderamente importante.

Puede haber, sin embargo, cierta superposición con los Design Patterns. Tomemos como ejemplo el GoF_Strategy [13]. Se trata de un Design Pattern que a su vez es también un Domain Pattern. Pero es una excepción, y hay que

buscar el sentido de esta particularidad en que los Design Patterns del estilo a GoF_Strategy, son grandes herramientas a la hora de estructurar el Domain Model.

Sin embargo debemos entender la siguiente sutileza en cuanto al enfoque:

- Como Design Pattern, hablamos desde un prisma técnico y generalista.
- Como Domain Pattern, lo único que nos interesa es el núcleo del Domain Model.

Por lo tanto, los Domain Patterns se encargan de hacer que el Domain Model sea:

- Claro.
- Conciso.
- Más expresivo.

Y además esté:

- Dotado de un propósito.

Facilitamos que el conocimiento adquirido sobre el Dominio, se evidencie en el Domain Model.

Un ejemplo de esta dicotomía entre los Domain Patterns y la dupla Design/Architectural Patterns sería la comparación entre:

- El Architectural Pattern Query Object [14].
- El Domain Pattern DDD_Specification.

El patrón Query Object es un patrón técnico, en el que un cliente puede definir

una consulta con una sintaxis basada en objetos, para encontrar cualquier objeto en el Domain Model.

El patrón DDD_Specification puede ser usado, también, para realizar consultas. Pero en vez de usar un "query object" genérico y establecer diferentes criterios de búsqueda, usa una especificación como el concepto que permite encapsular conocimiento sobre el dominio y que además comunica claramente su intención y/o propósito.

El ejemplo paradigmático de la dicotomía es:

- GoF_Factory [13].
- DDD_Factory.

Varía el enfoque:

- Los GoF_Factory se fijan en un nivel técnico.
- El DDD_Factory se fija en el nivel semántico del Domain Model.

Varía la intención y la implementación:

- El GoF_Factory_Method [13] trata de diferir la instanciación de la clase correcta a subclases.
- El GoF_Abstract_Factory [13] trata sobre la creación de familias de objetos dependientes.
- El DDD_Factory es sencillo y directo desde una perspectiva relativa a la implementación, ya que únicamente intenta capturar y encapsular el concepto de la creación para cierto tipo de clases.

2.2.4 Los elementos del DDD

A continuación describiremos algunos de los elementos y patrones que permiten articular el Domain-Driven Design.

2.2.5 DDD_Ubiquitous_Language

Si hubiese que salvar un solo elemento del DDD o destacar, con un solo término, la importancia del mismo, el DDD_Ubiquitous_Language, sería la solución.

El DDD_Ubiquitous_Language es un lenguaje estructurado en torno al Domain Model. Es utilizado por todos los miembros del equipo para conectar todas las actividades del mismo con el software, y sin que por ello deba ser el lenguaje que lo defina todo.

Nos permite eliminar, o al menos reducir, la fractura que se produce en el lenguaje que utilizan los expertos en el dominio por una parte, y los miembros técnicos del equipo de desarrollo por otra. De esta forma, evitamos la desconexión existente entre la terminología de las discusiones del día a día y la terminología utilizada en el código.

Para que esto sea posible debemos:

- Utilizar el modelo como vertebrador del lenguaje.
- Comprometer al equipo a que ejerciten, de forma inexorable, el DDD_Ubiquitous_Language, en todas las comunicaciones existentes entre ellos, así como en el propio código.
- Usar el mismo lenguaje en diagramas, escritos y presentaciones.
- Ante un problema, intentar modificar el lenguaje para enunciarlo y solucionarlo, de forma que si se consigue algún tipo de avance en la

dirección adecuada, se cambien todas las referencias al lenguaje anterior en el código y se incorporen esos nuevos términos al DDD_Ubiquitous_Language.
- Reconocer que un cambio en el DDD_Ubiquitous_Language supone, automáticamente, un cambio en el modelo.

Además:

- Los expertos en el dominio, deben mantenerse alerta ante términos o estructuras que no representen su experiencia con claridad.
- Los desarrolladores deben mantenerse alerta ante las ambigüedades e inconsistencias que puedan comprometer el diseño.

Gracias al DDD_Ubiquitous_Language, el modelo deja de ser un artefacto asociado al diseño y se convierte en algo omnipresente en todas las actividades que realizan conjuntamente los desarrolladores y expertos del dominio. El lenguaje pasa a ser el depositario del conocimiento de una forma dinámica. Todas las discusiones que se establecen en torno al DDD_Ubiquitous_Language contribuyen a revitalizar el significado que se esconde en los diagramas y en el código.

2.2.6 DDD_Entities

Los DDD_Entities son el primero de los elementos cuyo hábitat natural son los DDD_Aggregates.

Un DDD_Entity es un objeto que se define, principal y unívocamente, por su identidad.

Se trata de objetos, por tanto, que no se definen en base a sus atributos, sino que representan un hilo de identidad que se propaga a través del tiempo y a

veces, incluso, a través de distintas representaciones.

Podría darse el caso en el que este tipo de objetos:

- Tuviesen que ser emparejados aunque sus atributos difiriesen.
- Tuviesen que ser diferenciados pese a que sus atributos fuesen los mismos.

Por lo tanto una identidad errónea puede conducirnos a la corrupción de datos.

Hay que tener cierto cuidado con el concepto identidad en el ámbito del DDD, ya que los lenguajes de programación como Java o C# pueden confundirnos al implementar un operador de igualdad ("==") para todos los objetos que deseemos crear. Esta operación de igualdad determina si dos referencias apuntan al mismo objeto, comparando, por ejemplo, sus localizaciones en memoria. Y no es precisamente de eso de lo que hablamos cuando utilizamos la palabra identidad para referirnos a un DDD_Entity. Es algo mucho más sutil y que es necesario modelar con precaución, teniendo en cuenta que un mismo objeto en dos dominios diferentes puede tener identidad en el sentido DDD en uno de de ellos y carecer de la misma en el otro.

Por tanto, la identidad desde un punto de vista DDD, no es algo intrínseco a un concepto del mundo. Es algo superpuesto, porque nos resulta útil.

Cuando nos encontremos en el caso de un objeto que se distingue por su identidad en vez de por sus atributos, debemos:

- Resaltar este hecho a la hora de definir el objeto en el modelo.
- Favorecer una definición sencilla de la clase. Enfocada, claramente, hacia la continuidad del ciclo de vida y su identidad.
- Definir una manera de distinguir cada objeto a pesar de su forma o

historia.

- Mantenernos alerta ante requisitos que pidan emparejar objetos en base a sus atributos.
- Definir una operación que garantice un resultado único para cada objeto.
- Favorecer que el propio modelo sea el que defina qué es lo que significa ser igual, ser lo mismo.

Los DDD_Entities son los únicos nominables a ser un DDD_Aggregate_Root.

2.2.7 DDD_Value_Objects

Los DDD_Value_Objects son otro elemento que habitan los DDD_Aggregates.

Un DDD_Value_Object es un objeto que carece de identidad o al que la identidad no le aporta nada determinante, y que sin embargo, es definido en base a sus atributos.

La necesidad de esta distinción entre DDD_Entities y DDD_Value_Objects, más allá de sus implicaciones conceptuales y/o de modelado, debe ser vista de una forma mucho más práctica. Si tuviésemos que asignar identidades "artificiales" a todos los objetos del dominio nos encontraríamos con que:

- Podríamos dañar el rendimiento del sistema.
- Estaríamos añadiendo un trabajo analítico extra.
- Difuminaríamos la expresividad del modelo haciendo que todos los objetos pareciesen iguales (DDD_Entities).

Debemos tener en cuenta, como ya resaltamos con anterioridad, que el diseño de software es una batalla constante con la complejidad. Por lo tanto, es necesario hacer distinciones que favorezcan la comprensión de cuando

necesitamos un tratamiento especial, como mantener una identidad, y cuando no lo necesitamos.

Sería un error, en función de lo que hemos dicho, pensar que estamos ante una categoría de objetos que se definen por la ausencia de identidad. Si hiciésemos eso, estaríamos cayendo en una clara falta de pensamiento en negativo ("ausencia de …") y por tanto, poco o nada añadiríamos a nuestro vocabulario. Debemos, por el contrario, ser conscientes de la importancia de este tipo de objetos en nuestro modelo. Al tener características propias, los DDD_Value_Objects, son ideales para describir cosas.

Dicho de otra forma, los DDD_Value_Objects son instanciados para representar elementos del diseño que resultan clave por lo qué son y no por quién son.

Una vez aclarado este concepto, no podemos perder de vista, que los DDD_Value_Objects pueden albergar referencias a otros DDD_Value_Objects, pero también a otros DDD_Entities.

Es bastante común encontrarnos con que los DDD_Value_Objects son utilizados como parámetros en los mensajes entre objetos. Por lo tanto, estamos ante una categoría de objetos que, frecuentemente, son creados para una operación concreta y una vez concluye dicha operación, son desechados.

Cuando nos encontremos ante un objeto del que únicamente nos interesa el valor de los atributos como elemento del modelo, debemos:

- Clasificarlo como un DDD_Value_Object.
- Hacer que exprese el significado de los atributos que encapsula y asignarle una funcionalidad relativa a los mismos.
- Tratarlo como inmutable.

- No asignarle ningún tipo de identidad ni dotarlo de los complejos mecanismos necesarios para comprobar la identidad que sí tienen los DDD_Entities.

Recordar, finalmente, que es importante que los atributos que conforman un DDD_Value_Object, conceptualmente, formen un todo.

2.2.8 DDD_Service

Algunos de los conceptos/funcionalidades necesarios en el dominio, difícilmente pueden ser modelados, de forma natural, como si fuesen objetos. Si forzásemos ese concepto/funcionalidad, buscando que pudiese ser incluido, ya sea en un DDD_Entity o en un DDD_Value_Object; estaríamos, o bien distorsionando la definición de un objeto basado en el modelo, o bien añadiendo objetos artificiales sin ningún significado.

Un DDD_Service es una operación ofrecida a través de una interface independiente, en el modelo. A través del propio nombre (Service) se enfatiza la relación con otros objetos, siendo definido en relación a lo que ofrece a un cliente.

Es muy importante no utilizar los DDD_Services para despojar de comportamiento tanto a DDD_Entities como a DDD_Value_Objects. Únicamente deben ser usados cuando la operación que queremos modelar, representa, por si misma, un concepto importante dentro del dominio.

Un DDD_Service bien definido posee tres características:

- La operación que encapsula, está relacionada con un concepto del dominio que no tiene cabida de forma natural en un DDD_Entity o en un DDD_Value_Object.

- La interface a través de la cual se presenta, está definida en base a otros elementos del dominio y normalmente, el nombre de esta interface forma parte del DDD_Ubiquitous_Language.
- La operación encapsulada carece de estado (stateless). Es decir, cualquier cliente puede usar cualquier instancia del DDD_Service sin tener que preocuparse de la historia individual asociada a esa instancia concreta.

2.2.9 DDD_Domain_Event

Este es un patrón que no figura en la Biblia Azul, pero del que el propio Eric Evans ha estado hablando en sus ultimas apariciones públicas (QCon London 2009 y DDD-NYC SIG 2009 por poner un ejemplo).

Un DDD_Domain_Event captura una idea, un recuerdo de algo interesante, ocurrido en el pasado, y que afecta al dominio. Cuando éste se produce, es común que dispare un cambio de estado en algún elemento del dominio. Por lo tanto, estaríamos ante un patrón cuyo objetivo es proporcionar una solución para facilitarnos la tarea de crear dominios completamente encapsulados, entendiendo esa encapsulación, como la no-necesidad de depender de servicios o elementos externos para poder completar cualquier operación inherente al propio dominio.

Si atendemos al hecho de que estamos ante algo que ha ocurrido en el pasado, debemos asegurarnos de que los DDD_Domain_Events contengan verbos en pasado, ya que forman parte del DDD_Ubiquitous_Language.

Gracias a los DDD_Domain_Events conseguimos:

- Modelos más claros y expresivos.
- Flexibilidad arquitectónica, al poder:

- Representar el estado de las DDD_Entities.
- Desacoplar susbsistemas gracias a los Event Streams.
- Crear sistemas de alto rendimiento. [15][16][17][18][19][20][21]

2.2.10 DDD_Aggregate

Cuando nos encontramos ante un modelo que contiene asociaciones complejas, es muy difícil garantizar la consistencia al producirse cambios en los objetos. Las invariantes deben ser mantenidas y verificadas desde un punto de vista global, afectando a varios objetos a la vez, en oposición a cada objeto por separado. O, desde otro punto de vista: ¿Como podemos saber donde empieza y acaba un objeto que se compone a su vez de otros objetos?

Para solucionar estos problemas surge el DDD_Aggregate, como una abstracción que nos permite encapsular las referencias y asociaciones existentes dentro del modelo. Un DDD_Aggregate sería un cluster de objetos asociados que van a ser tratados como un todo, en cuanto a la modificación de datos se refiere.

Cada DDD_Aggregate consta de un DDD_Aggregate_Root y unos límites.

Los límites marcan el área de influencia del DDD_Aggregate. Qué es lo que se encuentra dentro del mismo.

El DDD_Aggregate_Root, es un DDD_Entity específico, que debe encontrarse dentro de los límites marcados por el DDD_Aggregate y se encarga de proporcionar la identidad al mismo.

Las reglas que se aplican a las transacciones que se producen dentro de un DDD_Aggregate son las siguientes:

- El DDD_Aggregate_Root tiene identidad global y es el responsable último de que las invariantes se comprueben.
- Los DDD_Entities que se encuentran dentro de los límites del DDD_Aggregate tienen una identidad local, interna al propio DDD_Aggregate.
- Nada que se encuentre fuera de los limites del DDD_Aggregate, puede contener una referencia a lo que se encuentra dentro, con la excepción del DDD_Aggregate_Root.
- Desde el punto de vista de un objeto exterior al DDD_Aggregate, se puede obtener una referencia a objetos internos del DDD_Aggregate a través del DDD_Aggregate_Root, pero únicamente para hacer un uso de la misma de forma transitoria, nunca para poder obtener una referencia permanente.
- Únicamente los DDD_Aggregate_Roots pueden ser obtenidos directamente con consultas a una Base de Datos. Los demás objetos que forman el DDD_Aggregate deben ser obtenidos navegando a través de las asociaciones establecidas dentro del mismo.
- Los objetos que se encuentran en los límites del DDD_Aggregate pueden mantener referencias a DDD_Aggregate_Roots externos.
- Una operación de borrado afecta por completo al DDD_Aggregate.
- Cuando se produce un cambio en algún objeto del DDD_Aggregate, deben satisfacerse las invariantes de todo el DDD_Aggregate.

Los elementos típicos que pueblan un DDD_Aggregate son:

- DDD_Entities.
- DDD_Value_Objects.
- DDD_Services.
- DDD_Domain_Events.

Y en menor medida, puede darse el caso de que contenga:

- DDD_Factories.

2.2.11 DDD_Factories

Así como una interface de un objeto debe servir para encapsular su implementación, de forma que permita que un cliente pueda usar su comportamiento sin tener que saber como está implementado; una DDD_Factory encapsula los detalles necesarios para crear un objeto complejo o un DDD_Aggregate.

Por lo tanto, debemos:

- Trasladar la responsabilidad de la creación de instancias de objetos complejos o DDD_Aggregates a un objeto aparte. Pudiera ser que éste no tenga una responsabilidad propia en el Domain Model, pero sin embargo debería aparecer en el diseño del dominio.
- Facilitar una interface que encapsule la complejidad del ensamblado del objeto o DDD_Aggregate, de forma que un cliente no necesite una referencia a las clases concretas implicadas en el proceso de creación.
- Crear DDD_Aggregates completos con sus invariantes.

Hay una serie de requisitos básicos a la hora de crear una buena DDD_Factory:

- Cada método que se encarga de la creación debe ser atómico y aplicar todas las invariantes asociadas al objeto o al DDD_Aggregate.
- Una DDD_Factory debe crear únicamente objetos en un estado consistente.
- En el caso de que tuviésemos que permitir la creación de un objeto en un estado incorrecto, debemos disponer de algún mecanismo que así lo

resalte (excepciones, por ejemplo).

- La DDD_Factory deberá ser abstracta.

2.2.12 DDD_Specification

Muchas veces nos vamos a encontrar con el caso en el que las Business Rules no encuentran acomodo en los DDD_Entities y DDD_Value_Objects del sistema. Además, la variedad y combinación de las mismas, estresarían el significado básico de cualquier objeto de dominio. Sin embargo, esas reglas pertenecen al dominio, por lo que no es factible sacarlas fuera del mismo. Si operásemos de esa forma, correríamos el riesgo de que el código del dominio dejara de expresar el modelo, conduciéndonos, inexorablemente, por el camino que lleva a los Dominios Anémicos (anti-pattern).

La programación lógica nos proporciona el concepto de los predicados, que no son más que una serie de objetos que representan reglas de forma atómica y combinable.

Aquí es donde entra en juego el patrón DDD_Specification.

Una DDD_Specification fija una restricción sobre el estado de un objeto, de forma que pueda ser usada para comprobar que cualquier objeto de un tipo determinado, satisface el criterio especificado.

Aplicando el Domain-Driven Design a este razonamiento, podemos crear una serie de DDD_Value_Objects que definan predicados de forma explícita para propósitos especializados. Por lo tanto una DDD_Specification es un predicado que determina si un objeto satisface o no una condición/criterio.

Hay, al menos, tres razones que justifican la necesidad de especificar el estado de un objeto del dominio:

1. Validar un objeto para saber si está listo para un propósito determinado.
2. Seleccionar un objeto concreto dentro de una colección.
3. Concretar la creación de un nuevo objeto que cumpla con alguna necesidad.

Por lo tanto, con las DDD_Specifications, cubriremos tres usos concretos:

1. Validación.
2. Selección.
3. Construcción. [1][22]

2.3 El Concepto de Diseño Agile

Para entender el concepto de Diseño Agile, necesitamos situar el contexto en un Equipo Agile.

En un Equipo Agile, la visión global evoluciona ligada íntimamente al desarrollo del software. Con cada iteración, el equipo mejora el diseño del sistema.

En un momento dado cualquiera, el diseño del software alcanza su madurez a través del diseño que tenemos en ese momento. El equipo no va a malgastar su tiempo intentando adelantarse a los acontecimientos en busca de futuros requisitos y/o necesidades. Por lo tanto, no va a cometer el error de dotar al software de una infraestructura que facilite el desarrollo de esos futuros requisitos que quizás necesitemos dentro de un tiempo. El esfuerzo y energía del equipo se centrará en la estructura actual del sistema, para que sea todo lo buena que humanamente consideremos posible.

No debemos caer, sin embargo, en la trampa de creer que ésto supone un

abandono tanto de la Arquitectura como del Diseño. Más bien al contrario. Estaríamos ante una forma de desarrollar, en base a pequeños incrementos, la Arquitectura y el Diseño más apropiados para el sistema. De esta forma ambos se ajustan a medida que el sistema crece y evoluciona.

Podemos concluir, por lo tanto, que el Diseño Agile es un proceso continuo que busca adaptar y mejorar el sistema a medida que crece, o como diría Kent Beck: "Embrace Change" (Abraza el Cambio) [23].

2.3.1 Diseño Agile vs Big Desing Up Front (BDUF)

El acrónimo BDUF (Big Design Up Front) se aplica a todas aquellas metodologías en las que se realiza un diseño previo (costoso en tiempo y recursos) a la codificación y prueba del software. Este diseño previo es de capital importancia y marcará el rumbo de la posterior codificación y prueba del software.

Suele asociarse a metodologías de tipo Waterfall y su principal inconveniente, además de lo caro que resulta, es la dificultad que opone al cambio de las especificaciones y requisitos.

El contraste con el Diseño Agile, y por añadidura con las Metodologías Agile, es total, si tenemos en cuenta que en el caso Agile el proceso de diseño es algo constante y que se puede plantear en cualquier etapa del desarrollo del software. Esto suele derivar tanto en costes más moderados, como en facilidad a la hora de resolver los cambios en las especificaciones y requisitos.

Por el contrario, uno de los argumentos más extendidos a favor del BDUF, es que el hecho de que muchos desarrolladores sean buenos a la hora de desarrollar software, no implica que sean buenos a la hora de diseñarlo. Por lo tanto, esta visión no apoya la idea de que el diseño es algo que puede surgir

en cualquier momento y a cualquiera de los desarrolladores.

El argumento que más controversia levanta en este tipo de comparaciones es que muchos detractores del Diseño Agile, confunden no practicar BDUF con practicar NDUF (No Design Up Front). Huelga decir, que no es el caso. [24] [25][26][27]

2.4 Design Smells, síntomas de un diseño pobre

¿Cómo sabemos si el diseño de un sistema que sigue los preceptos Agile es bueno?

Para responder a esta pregunta tenemos los siguientes epígrafes en los que describiremos y pondremos nombre a los síntomas de un diseño pobre. Estos síntomas son los que conocemos como Design Smells y es habitual que campen a sus anchas por toda la estructura del programa.

Estos Design Smells serían el equivalente, pero a un nivel de abstracción más alto, a los Code Smells. Podríamos considerarlos como Smells que en lugar de afectar a una pequeña porción de código, afectan a partes completas de la estructura del programa.

La mayor parte de las veces, el "hedor" (traducción directa del término Smell), viene causado por la violación de alguno de los Principios de Diseño que trataremos más adelante agrupados bajo el acrónimo S.O.L.I.D.. [2[3]

2.4.1 Rigidez (Rigidity)

Es la tendencia del software a dificultar el cambio, incluso aunque éste sea aparentemente sencillo.

Cuando un simple cambio causa una miríada de cambios encadenados en módulos dependientes, nos encontramos ante un diseño que adolece de Rigidez.

Cuantos más módulos se vean afectados, más rígido será nuestro diseño.

2.4.2 Fragilidad (Fragility)

Es la tendencia del software a romperse por múltiples lugares como reacción ante un cambio. Y lo que es más grave, una gran parte de esos lugares en los que el código rompe (no compila), no tienen relación conceptual con el lugar en el que hemos realizado el cambio.

Cuando estamos ante este caso, es frecuente encontrarse con que al intentar arreglar todos esos problemas surgidos ante el cambio, desencadenamos, a su vez, una nueva serie de errores en lugares que, nuevamente, no tienen relación conceptual con ese nuevo cambio.

Por lo tanto, podemos afirmar que los módulos que sufren de Fragilidad empeoran más, cada vez que intentamos arreglarlos.

2.4.3 Inmovilidad (Inmobility)

Podemos considerar que un diseño adolece de Inmovilidad, cuando pese a contener partes que podrían ser útiles en otros sistemas, su coste de reutilización se convierte prácticamente en prohibitivo, debido al esfuerzo y riesgo necesarios para poder desligar esa parte del original.

En otras palabras, si nuestro diseño sufre de Inmovilidad, la posibilidad de reutilización del mismo es nula (o tan costosa que deja de merecer la pena).

2.4.4 Viscosidad (Viscosity)

Un proyecto Viscoso es aquel en el que resulta muy difícil preservar y mejorar el diseño.

Viscosidad del software

A la hora de acometer un cambio en el programa, generalmente, se nos ofrecen varias posibles soluciones. Algunas de éstas ayudan a preservar el diseño, o al menos, no lo emponzoñan. Otras, sin embargo, oscurecen y empeoran el diseño (comúnmente conocidos como "Hacks").

Dentro de este escenario debemos entender la Viscosidad como una medida. La Viscosidad del software será alta cuando resulte más sencillo acometer los cambios a través de soluciones que empeoran el diseño. Se produciría, por tanto, una situación en la que "es fácil hacer lo incorrecto y difícil hacer lo correcto."

Viscosidad del entorno

La Viscosidad del entorno se produce cuando el entorno de desarrollo es lento e ineficiente.

Aquellos entornos, en los que los tiempos de compilación son tan largos, que puedan provocar que los desarrolladores se vean tentados a realizar cambios que no desemboquen en la recompilación de toda la aplicación, aunque por ello empeoren el diseño, sería un ejemplo típico.

Otro ejemplo serían los Sistemas de Control de Versiones (source code control systems) ineficientes. Éstos, pueden tardar horas en actualizar unos pocos ficheros. El desarrollador podría verse obligado a tomar la decisión de realizar cambios que afecten al menor numero posible de ficheros, aunque así

empeore el diseño.

2.4.5 Complejidad Innecesaria (Needless Complexity)

Si nuestro diseño contiene elementos que no son útiles en este preciso momento, estaremos cayendo en Complejidad Innecesaria.

La razón más común para que se produzca este Design Smell, es anticipar cambios en los requisitos y desarrollar infraestructura para facilitar esos cambios que todavía no se han producido (y muchas veces, no se producirán jamás).

Lo más curioso de este problema es que la intención original es buena. Es decir, puesto que queremos que nuestro software sea flexible y acepte cambios más adelante, desarrollamos todo tipo de infraestructura que facilite esa futura tarea. El problema es que conseguimos el efecto inverso. Al haber desarrollado tantas clases que por ahora no vamos a usar, enfangamos el diseño.

De todos los males asociados al desarrollo de software, éste es uno de los más comunes y requiere de mucha disciplina y experiencia para no caer repetidamente en él.

2.4.6 Repetición Innecesaria (Needless Repetition)

Cuando el mismo código aparece una y otra vez, con variaciones mínimas, claramente, los desarrolladores están pasando por alto una abstracción.

El problema se manifiesta en toda su dimensión cuando necesitamos cambiar ligeramente ese código, ya que nos obliga a revisar todo el código fuente a la caza y captura de todas esas repeticiones. Y aun por encima, como existen

variaciones mínimas entre las distintas manifestaciones del mismo, no siempre es trivial el cambio.

Imposible mejorar la explicación de Robert C. Martin [3]:

> "Cut and paste may be useful text-editing operations, but they can be disastrous code-editing operations."

2.4.7 Opacidad (Opacity)

Es la tendencia de un módulo a ocultar su verdadero cometido.

Con el tiempo, los cambios producidos en un módulo, dificultan más la comprensión del mismo. Y este sería el mejor de los casos, ya que estamos presuponiendo que el módulo se diseñó correctamente y en algún punto del pasado no adolecía de Opacidad, ya que, muchas veces esa Opacidad es característica indisoluble desde su temprana concepción.

2.5 Los Principios del Diseño Agile S.O.L.I.D.

> "The principles are there to help us eliminate bad smells. They are not a perfume to be liberally scattered all over the system."
>
> Robert C. Martin

El Diseño Agile es un proceso.

Los desarrolladores que siguen los preceptos del Agile, no destilan el diseño una vez cada pocas semanas. Si fuese así, el Diseño Agile sería un evento y no un proceso.

Por contra, el Diseño Agile es el resultado de la aplicación continua de una

serie de principios, patrones y prácticas con el objeto de mejorar la estructura y legibilidad del software.

Conocemos esos principios bajo el acrónimo S.O.L.I.D.. [2][3]

2.5.1 SRP (Single-Responsability Principle)

Podemos rastrear los orígenes de este principio en los trabajos de Tom DeMarco [28] y Meilir Page-Jones [29] bajo el término "cohesión", entendiendo el mismo, como una medida de la relación funcional existente entre los elementos que conforman un módulo.

Sin embargo, Robert C. Martin [2] lleva ésto a una nueva dimensión, al tamizar el concepto original con el matiz que suponen las razones que tiene un módulo para cambiar.

El SRP afirma que una clase debería tener una única razón para cambiar.

Partiendo de esta nueva lectura efectuada por "Uncle" Bob, podríamos incluso aplicar el concepto a nivel método, con lo que obtendríamos que un método debería tener una única razón para cambiar.

Otro concepto imprescindible para poder comprender las consecuencias del SRP es el de "Responsabilidad".

En el contexto del SRP, definimos una Responsabilidad como una razón para el cambio. Es decir, si podemos enumerar más de un motivo para cambiar una clase (o un método), esa clase (o método) tiene más de una Responsabilidad y por lo tanto incumpliría el SRP.

Sin embargo no siempre es fácil determinar este aspecto, ya que estamos

acostumbrados a agrupar las responsabilidades.

Un ejemplo clásico que ilustra esta perspectiva es el de un módem. Si consideramos las siguientes cuatro operaciones:

- Llamar.
- Enviar.
- Recibir.
- Colgar.

Probablemente podamos concluir que son propias de un módem y que, por lo tanto, pueden ser agrupadas en una única interfaz, ya que constituyen una única Responsabilidad (Ver figura 1).

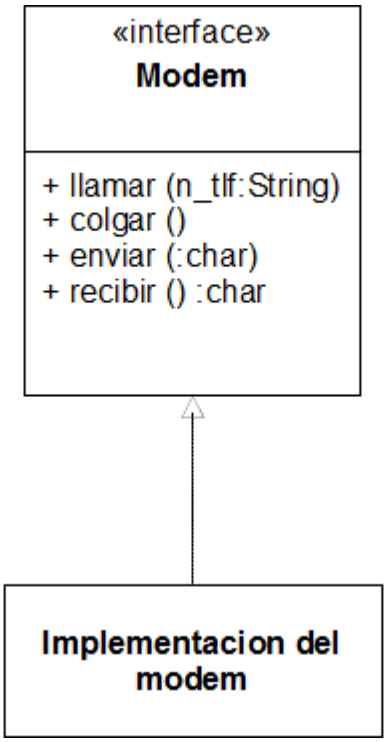

Figura 1: Implementación del modem con una sola interface

Sin embargo, estamos ante dos Responsabilidades distintas.

Por un lado nos encontramos con la Responsabilidad "Gestión de la Conexión" definida por las operaciones:

- Llamar.
- Colgar.

Mientras que por otro estaríamos ante la Responsabilidad "Canal de Datos (Comunicación)":

- Enviar.
- Recibir.

Por lo tanto, podríamos considerar también este otro diseño, que respeta mejor el SRP (Ver figura 2).

La cuestión es la siguiente, ¿debemos separar estas dos responsabilidades?

La respuesta la vamos a encontrar en el contexto, ya que, de forma abstracta, no se puede contestar correctamente a la pregunta.

Si nos encontramos en un contexto en el que, por poner un ejemplo, la aplicación puede cambiar afectando la signatura de alguna de las operaciones relativas a la "Gestión de la Conexión" (Colgar/Llamar), debemos aplicar el SRP (y por tanto favorecer la solución de la figura 2), ya que nos encontraríamos ante un Design Smell de Rigidez, puesto que las clases que llaman a las operaciones Enviar/Recibir serían recompiladas con más frecuencia de la necesaria.

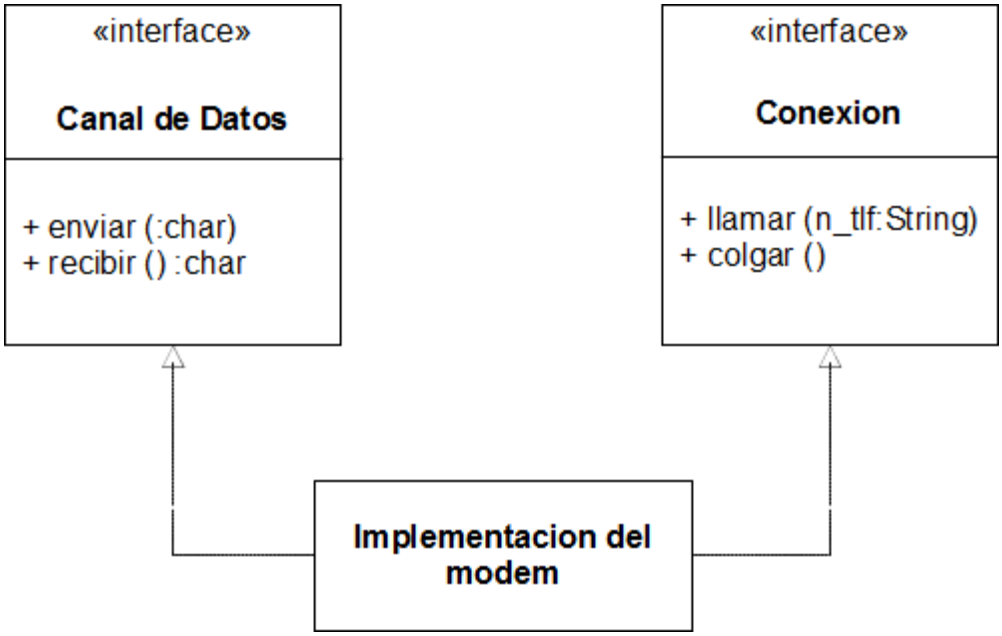

Figura 2: Implementación del modem con dos interfaces

Sin embargo, si el contexto en el que nos encontramos no produce un cambio en una sola de las Responsabilidades y ambas, cuando cambian, lo hacen a la vez, no sería necesario aplicar el SRP, ya que esa separación sugeriría un Design Smell de Complejidad Innecesaria.

Por lo tanto, deberíamos asumir el siguiente corolario: "Un eje de cambio es un eje de cambio únicamente en el caso de que los cambios se produzcan".

2.5.2 OCP (Open/Closed Principle)

Debemos este principio a Bertrand Meyer [30] que en cierta forma contesta a la pregunta formulada por Ivar Jacobson [31] unos años antes: ¿Cómo podemos crear diseños que se comporten de forma estable ante la posibilidad de cambio y que por tanto estén preparados para sobrevivir más allá de la primera versión?.

La respuesta la encontramos en el OCP.

Las entidades software (módulos, clases, métodos, …) deben estar abiertas a la extensión pero cerradas a la modificación.

Este principio es muy útil a la hora de solucionar el Design Smell de la Rigidez, ya que, cuando un único cambio a un programa desemboca en la necesidad de cambiar también varios de los módulos dependientes (la definición de Rigidez), es hora de aplicar una Refactorización [32], de forma que al volver a encontrarnos con una situación similar, podamos resolverla añadiendo código nuevo en vez de tener que editar código escrito con anterioridad.

Cuando un módulo satisface el OCP tiene las siguientes características:

1. Está abierto a la extensión. Por lo tanto responde correctamente ante la posibilidad de cambio, ya que al poder extender el comportamiento del módulo, se puede cambiar dicho comportamiento.
2. Está cerrado a la modificación. Al extender el comportamiento del módulo no modificamos el código fuente ya existente, sino que añadimos código nuevo que es el que se encarga de codificar ese nuevo comportamiento.

Para conseguir este objetivo, tenemos que echar mano del viejo concepto de la Abstracción.

Un ejemplo que nos puede ayudar a ilustrar la aplicación del OCP y la Abstracción, sería el de un Cliente/Servidor.

Primeramente suponemos un contexto en el que violen el OCP (Ver figura 3).

Figura 3: Cliente no cumple OCP

En este caso tenemos un Cliente que usa al Servidor directamente. Ambas clases son concretas.

El problema se plantea a la hora de hacer que el Cliente utilice un objeto Servidor distinto, ya que es entonces cuando el Cliente se muestra Abierto a la Modificación (lo contrario de lo que defiende el OCP), ya que nos veríamos en la obligación de cambiar al Cliente para poder instanciar al nuevo Servidor. Esto es precisamente lo que intenta evitar el OCP.

Podríamos plantear un contexto radicalmente diferente en el que se cumpliese el SRP (Ver figura 4), aplicando el patrón GoF_Strategy [13] a este problema.

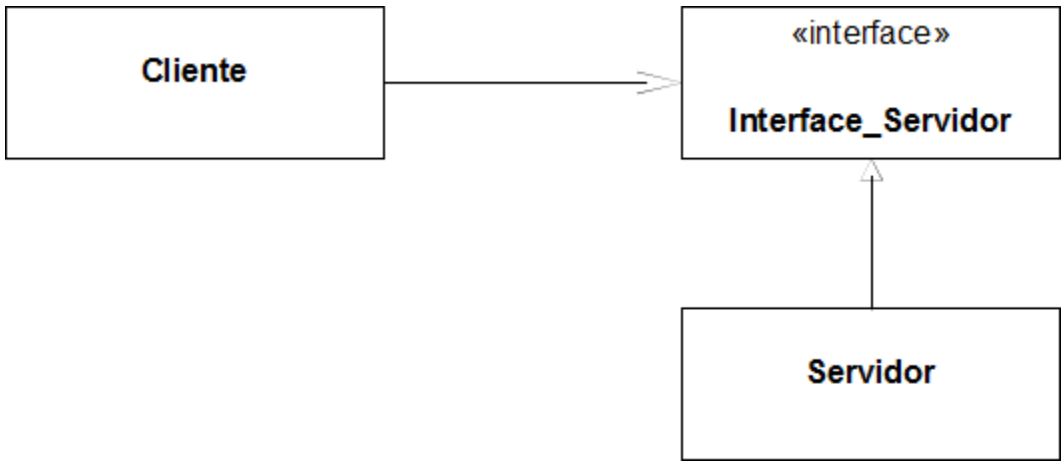

Figura 4: Cliente cumple OCP (GoF Strategy)

En este caso disponemos de una interface llamada Interface_Servidor, que es la encargada de declarar cuales son los términos que debe cumplir cualquier clase Servidor que pretenda permitir la comunicación con la clase Cliente. La clase Cliente, por tanto, utiliza esta Abstracción (Interface_Servidor) en lugar de utilizar la clase concreta Servidor, de forma que si necesitásemos un nuevo Servidor, con tal de que éste implementase la interface Interface_Servidor, podría ser usada por el Cliente sin ningún cambio.

Si seguimos este camino, podríamos volver a evaluar la situación de la siguiente forma:

- El Cliente tiene una Dependencia en la Abstracción Interface_Servidor, a través de la cual declara que necesita que le ayuden a la hora de realizar una serie de tareas.
- Puesto que cualquier clase Servidor que implemente Interface_Servidor podrá hacerlo de la manera que decida, estamos abriendo una puerta a la extensión del comportamiento de Cliente y a su vez cerrando otra puerta en cuanto a la modificación del mismo.
- Por lo tanto cumpliríamos con el OCP.

Si evaluamos ambas situaciones, nos damos cuenta de que la aplicación del OCP viene con un coste añadido. Más allá del tiempo y esfuerzo necesarios para crear la Abstracción Interface_Servidor, este tipo de abstracciones incrementan la complejidad del diseño. Esto es algo que debe ser tenido en cuenta a la hora de aplicar el OCP.

Una feliz consecuencia de la aplicación del OCP, y que no es aparente, se manifiesta a través de la forma de evitar ese coste añadido que supone crear las abstracciones. Podríamos seguir una metodología de tipo Test-First Programming:

- TDD (ya sea en su versión más clásica o de estado, o en su versión "mockista" o de interacción).
- El más moderno BDD (el que seguimos y defendemos en este Proyecto Fin de Carrera).

Al vernos en la obligación de probar el sistema antes de codificarlo, crearemos todas las abstracciones necesarias para que este sistema se adapte a las pruebas, con lo que indirectamente estaremos protegiéndonos, con estas abstracciones, de los efectos que puedan producir los cambios que surjan más adelante.

Pero una herramienta tan poderosa como esta, no viene exenta de una advertencia, y es que debemos resistir el impulso a abstraer demasiados conceptos prematuramente y solo aplicar este concepto a las partes del programa que tienen tendencia a cambiar con frecuencia. Esto es algo tan importante como la abstracción en si misma.

2.5.3 LSP (Liskov Substitution Principle)

Debemos el LSP a Barbara Liskov [33] quién afirmó que cualquier subtipo deberá poder ser substituido por su tipo base.

Por lo tanto podemos afirmar, que mientras el OCP ocultaba los mecanismos de la Abstracción y el Polimorfismo, el LSP se encarga de regular la Herencia.

Este es un Principio que resulta más fácil de abordar si le damos la vuelta. Es decir, vamos a considerar qué es lo que ocurriría si no cumpliésemos el LSP.

Supongamos un método Prueba que recibe como único argumento un objeto de tipo SoyUnaClaseBase. Supongamos a continuación que un objeto que hereda de SoyUnaClaseBase, por ejemplo HeredoDeLaClaseBase, es pasado

como argumento a nuestro método Prueba, ya que, al heredar de SoyUnaClaseBase es un escenario perfectamente factible.

Si por culpa de esto, nuestro método Prueba se comporta de forma no deseada, estaríamos ante una violación del LSP por parte de HeredoDeLaClaseBase.

Como consecuencia de una violación del LSP, nos podemos encontrar con una violación de OCP, si el método que recibe la clase que viola el LSP (en nuestro ejemplo anterior sería el método Prueba), añade código para manejar ese caso en concreto, ya que, ahora, dicho método ya no está cerrado con respecto a todos los posibles objetos que hereden del tipo base (de nuevo en el ejemplo anterior, SoyUnaClaseBase).

Podemos sacar una conclusión muy interesante del LSP. Un modelo visto de forma aislada (aislado de sus posibles contextos), difícilmente va a ser validado. Por lo tanto, solo podremos validar un modelo desde el punto de vista de los clientes que lo vayan a utilizar/consumir.

Ésta es, claramente, otra razón de peso para utilizar una metodología Test-First Programming, ya que de esta forma, al menos tenemos una suite de tests / pruebas / especificaciones que comprueban desde un punto de vista propio de un cliente la clase / método / comportamiento que vamos a desarrollar.

Otra posible solución que ayude a mitigar este tipo de problemas sería el Diseño por Contrato expuesto por Bertrand Meyer [30], ya que al explicitar el "contrato" al que está sometido una clase, disminuimos la posibilidad de que algún cliente la utilice para algo que se aleje de su cometido original.

Aún así no podemos dejar de señalar (de nuevo) que una buena suite de

pruebas puede definir claramente el comportamiento que podemos esperar de una clase cualquiera, de forma que, un desarrollador que se enfrenta a una clase programada por nosotros, deba consultar previamente los test escritos, como si de documentación se tratase (de hecho ES documentación, pero esa es otra historia completamente distinta).

Algo que debe hacer saltar todas nuestras alarmas, ya que es una posible indicación de que podría haber una violación del LSP, es cuando nos encontramos con el caso en el que una clase derivada de otra clase base, sobrescribe (override en C#) un método para el que ya había un comportamiento programado en la clase base. Al sobrescribir ese comportamiento predefinido por la clase base se corre el riesgo de que la clase derivada elimine parte de la funcionalidad que viene definida en la clase base, de forma que, muy probablemente, acabe produciendo una violación del LSP por parte de la clase derivada.

Desde un punto de vista más personal, resaltar que si se practica BDD, este último caso es realmente raro, ya que, por norma general se favorecerá el uso de interfaces frente al uso de clases base.

Por lo tanto, el LSP es el guardaespaldas del OCP, ya que la capacidad de intercambiar subtipos permite que un modulo que depende de la clase base, pueda ser extendida sin necesidad de ser modificada.

2.5.4 ISP (Interface Segregation Principle)

El objeto de este principio que debemos a Robert C. Martin [2] (responsable también de la enumeración de los principios S.O.L.I.D.), es la descomposición de las interfaces que son responsables de demasiado comportamiento, en varias interfaces más especializadas y por lo tanto, con mejores posibilidades de cumplir el SRP (más en concreto, el ser más cohesivas).

Pese a que podamos encontrarnos con el caso en el que una clase determinada necesite de una interface muy poco cohesiva, también sugiere que los clientes no deberían verlas como una clase única, sino como un conjunto de interfaces que si cumplan con la cohesión.

Otro concepto indisoluble al ISP es el de Polución de las Interfaces (Interface Pollution).

Este concepto se refiere a la situación en la que una interface que cumple escrupulosamente con el ISP (y por lo tanto con la cohesión y SRP), se ve polucionada, con al menos un método que en realidad no debería estar ahí. Generalmente, encontramos que la razón por la cual ese método ha sido añadido, es la de favorecer a una sola de las múltiples implementaciones de la misma. El problema se manifiesta en que cualquier clase que decida implementar la interface se ve en la obligación de implementar dicho método que como ya dijimos anteriormente, le es ajeno. Ante esta situación, nos encontramos con una potencial violación del LSP, ya que, las clases a las que ese nuevo método les resulta ajeno, van a proveer al mismo con una implementación degenerada (o por defecto), dejando, por tanto, abierta la puerta a la violación del LSP.

Éste es uno de los escenarios típicos por las que nuestras interfaces crecen y crecen sin control. La causa hay que buscarla en que cada vez que añadimos un método a alguna de las múltiples implementaciones de una interface, promovemos dicho método a la interface. Este es el comportamiento que debemos evitar, y en contraste, intentar cumplir con la máxima de: clientes diferentes implican interfaces diferentes.

Por lo tanto una forma más o menos precisa de definir el ISP es la de que los clientes no deberían verse forzados a depender de métodos que no usan.

La razón es tan simple como que, si fuerzas a los clientes a depender de esos métodos, los clientes son vulnerables a cualquier cambio en dichos métodos, y esa es una dependencia que debemos evitar siempre que sea posible.

Evaluar el ISP, requiere un pequeño cambio de mentalidad a la hora de "entender" el diseño de interfaces, ya que, en vez de fijarnos exclusivamente en la relación interface / implementador, debemos empezar a prestar atención a la relación que se establece entre las interfaces (e implementaciones) que representan a clientes distintos y sus acoples (coupling). En cuanto conseguimos ese cambio de enfoque, la tendencia es a que nuestras interfaces tiendan a adelgazar (y a multiplicarse), respetando de forma más fidedigna el SRP (cuantos menos métodos tiene una interface, más posibilidades existen de que esa interface defina un única responsabilidad).

2.5.5 DIP (Dependency Inversion Principle)

El último principio de los principios S.O.L.I.D. es también obra de Robert C. Martin [2].

Podemos enunciarlo en dos apartados:

- Los módulos de alto nivel (high-level modules) no deben depender de los módulos de bajo nivel (low-level modules). Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de los detalles, sino que son los detalles los que deben depender de las abstracciones.

La *inversión* proviene del hecho de que, la estructura de dependencias que se da en un programa típico de POO, debería *estar invertida* con respecto a la estructura de dependencias que se establecen en la programación más clásica

no orientada a objetos.

Debemos buscar la razón en que, si seguimos la cadena de dependencias desde arriba a abajo (de alto nivel a bajo nivel), nos vamos a encontrar con que lo más importante de nuestro diseño que suele encontrarse en los niveles más altos, van a "heredar" una dependencia a los conceptos más mecánicos que son propios de los niveles más bajos.

Esto limita claramente la reutilización, pero también es una posible fuente de problemas de mantenimiento, si tenemos en cuenta que cualquier tipo de cambio que se produzca en nuestros niveles más bajo, nos puede obligar a cambiar los niveles más altos, que es donde habitan los conceptos más cercanos al dominio.

De hecho, si lo pensamos por un instante, debería ser exactamente a la inversa. Los módulos de más alto nivel deberían tomar el mando y ser ellos los que influyesen en los módulos de más bajo nivel.

Cuando aplicamos ésto a la Arquitectura de Capas (Layer Architecture), no podemos pasar por alto la afirmación de Grady Booch [34]:

> "Toda arquitectura orientada a objetos bien estructurada debe tener capas claramente definidas, de forma que cada una de las capas expongan un conjunto coherente de servicios a través de una interface controlada y bien definida."

Por desgracia, la manera más tradicional y común de interpretar y aplicar esta afirmación consiste en implementaciones con dependencias que van de arriba a abajo (Ver figura 5).

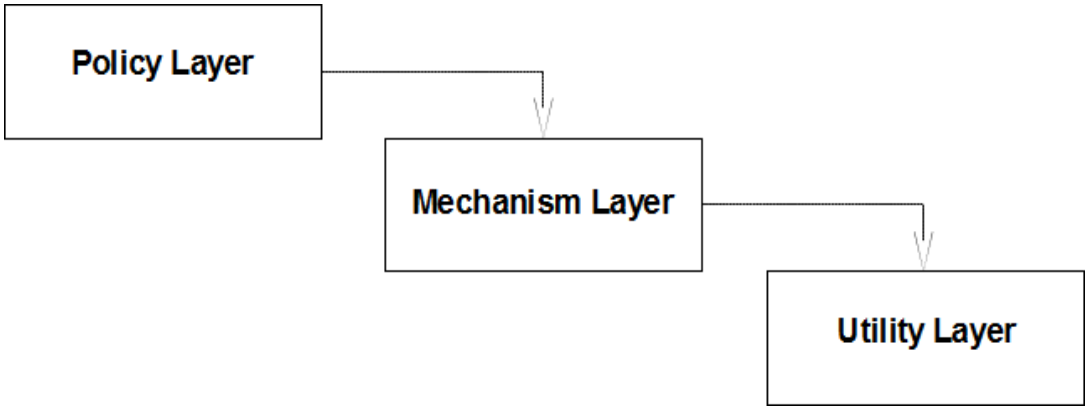

Figura 5: Arquitectura de Capas que incumple DIP

Sin embargo con este tipo de implementaciones estamos provocando lo que, unas lineas antes, intentábamos evitar, obteniendo como resultado que nuestra Policy Layer (la más importante desde un punto de vista del dominio) acabe dependiendo (y siendo susceptible a los cambios) de la Utility Layer (la menos importante desde ese mismo punto de vista del dominio).

Una forma de arreglar este desaguisado es la aplicación del conocido Hollywood Principle:

> "No nos llames que ya te llamaremos nosotros."

Aplicado al caso que tenemos entre manos, el Hollywood Principle, no solo promueve el DIP, sino que también promueve el Owner Inversion. Con esto nos referimos a que mientras anteriormente lo común era que la capa donde residían las implementaciones de las interfaces eran las dueñas de las interfaces, ahora serán los clientes y la capa en la que estos residen, los que se adueñan de dichas interfaces. Es decir, los módulos de más bajo nivel, son los que implementan el comportamiento definido por las interfaces declaradas en el entorno de, y llamadas por, los módulos de más alto nivel.

Siguiendo este tipo de razonamiento, el anterior esquema (Ver figura 5) se podría convertir en algo más digno de las sabias palabras de Grady Booch [34] (Ver figura 6). De forma que el concepto de posesión (ownership) por parte de los clientes hacia la interface de la que dependen, se traduciría en que las interfaces serían distribuidas en el mismo paquete que los clientes que las usan, en vez de en los paquetes de los servidores que las implementan.

Si llevamos este concepto al extremo y además lo sazonamos con los principios que rigen el Domain-Driven Design llegaríamos a la Onion Architecture, obra de Jeffrey Palermo [35] y deudora a su vez de la Hexagonal Architecture de Alistair Cockburn [36] y que precisamente usamos en este Proyecto Fin de Carrera.

El DIP incluye otro concepto igualmente importante y que podríamos resumir en la famosa frase:

> "Depende de Abstracciones."

En clara oposición a:

> "Depende de Implementaciones."

Si seguimos el hilo de esta afirmación, podríamos concluir que todas las relaciones que se establecen en un programa, deberían terminar en una interface:

- Ninguna variable debería contener una referencia a una clase concreta.
- Ninguna clase debería derivar de una clase concreta.
- Ningún método debería sobrescribir un método ya implementado en su clase base.

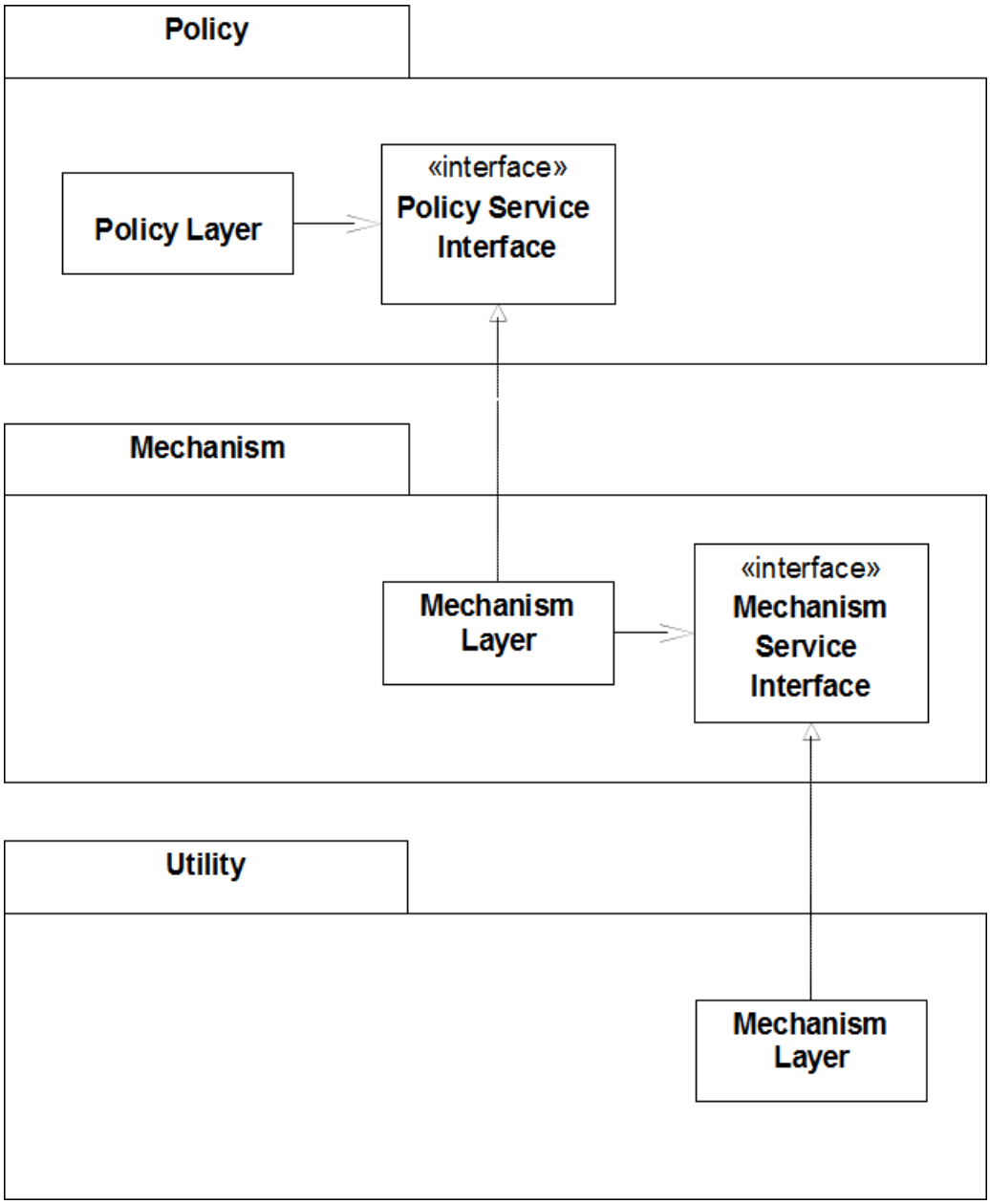

Figura 6: Arquitectura de Capas Invertidas que cumple DIP

Como conclusión al DIP, podríamos decir que si en un programa las dependencias están invertidas estaremos ante un Diseño Orientado a Objetos,

mientras que si no están invertidas estaremos ante un Diseño Procedimental.

No podemos finalizar el DIP sin hacer referencia a toda la fiebre actual de Frameworks DI/IoC. Debemos aclarar que ese tipo de Frameworks se basan en, y surgen gracias a, el DIP.

2.6 Onion Architecture

Es una propuesta de Jeffrey Palermo [35] en base a la Hexagonal Architecture de Alistair Cockburn [36], donde tanto el DIP del SOLID como el DDD juegan un papel clave.

Ambas arquitecturas tienen como premisa principal evitar la dependencia de la infraestructura, de forma que ésta es separada del núcleo de la aplicación.

Para que esta idea funcione es imprescindible escribir código específico de adaptación entre infraestructura y el resto de la aplicación.

De esta forma evitamos el acople excesivo (tightly copled) entre ambas.

2.6.1 Principios de la Onion Architecture

Estos son los principios por los que se rige la Onion Architecture:

- Construcción de la aplicación en torno a un modelo de objetos independiente.
- Las capas interiores definen interfaces.
- Las capas exteriores implementan las interfaces.
- El vector que representa el acople (coupling) va en dirección al centro.
- El código del núcleo de la aplicación puede ser compilado y ejecutado sin depender de la infraestructura.

La figura 7 debería ayudar a comprender mejor el sentido de la Onion Architecture.

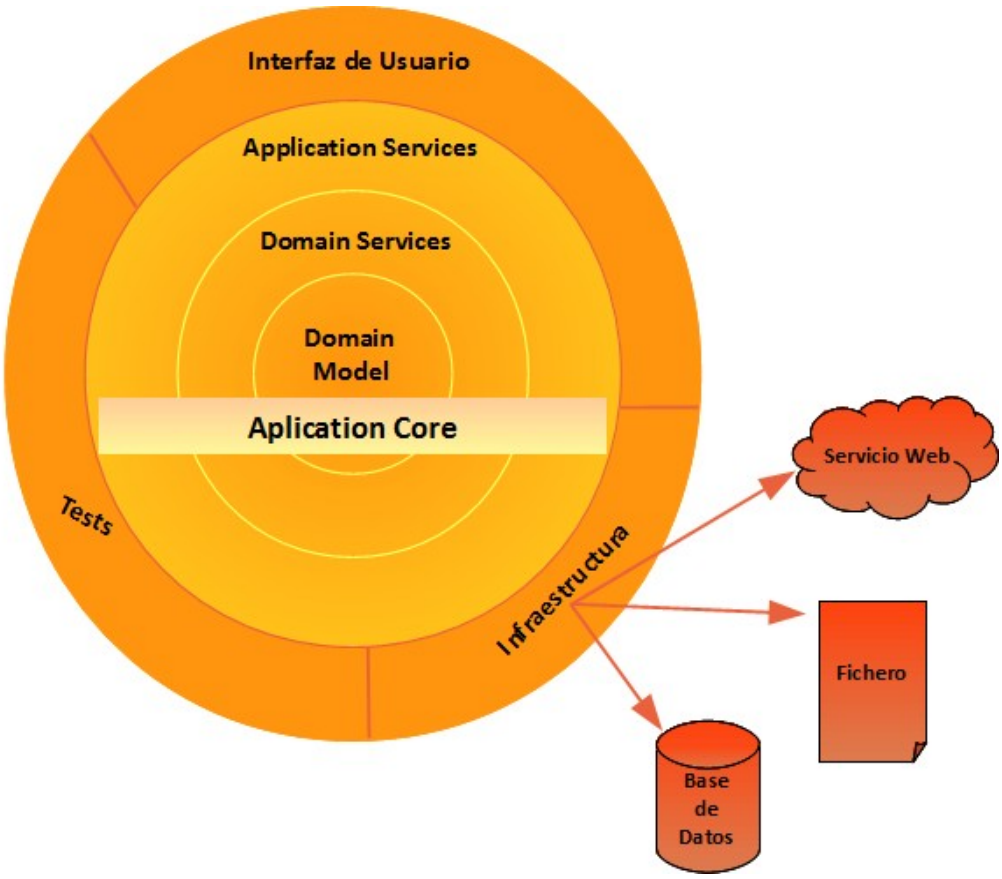

Figura 7: Representación de la Onion Architecture

Atendiendo a los principios antes enumerados, es importante señalar que:

- El Domain Model se sitúa en el centro.
- Una interface definida, por ejemplo, en la capa Domain Services o en la capa Application Services, podría ser implementada tanto en la capa Infraestructura como en la capa Interfaz de Usuario.

- Las dependencias van hacia el centro. Es decir, cualquier capa exterior puede depender de todas las capas interiores que necesite (no solo con la que tiene una frontera directa). Por lo tanto, la única capa que no tiene dependencias es precisamente la del Domain Model.

2.6.2 Onion Architecture y Layered Architecture

Si comparamos la Onion Architecture frente a la más tradicional Layered Architecture podremos comprender mejor el sentido de la primera.

Basándonos en la asunción de una Layered Architecture arquetípica (Ver figura 8), queda claro que se establece una dependencia entre las capas superiores con respecto a las inferiores.

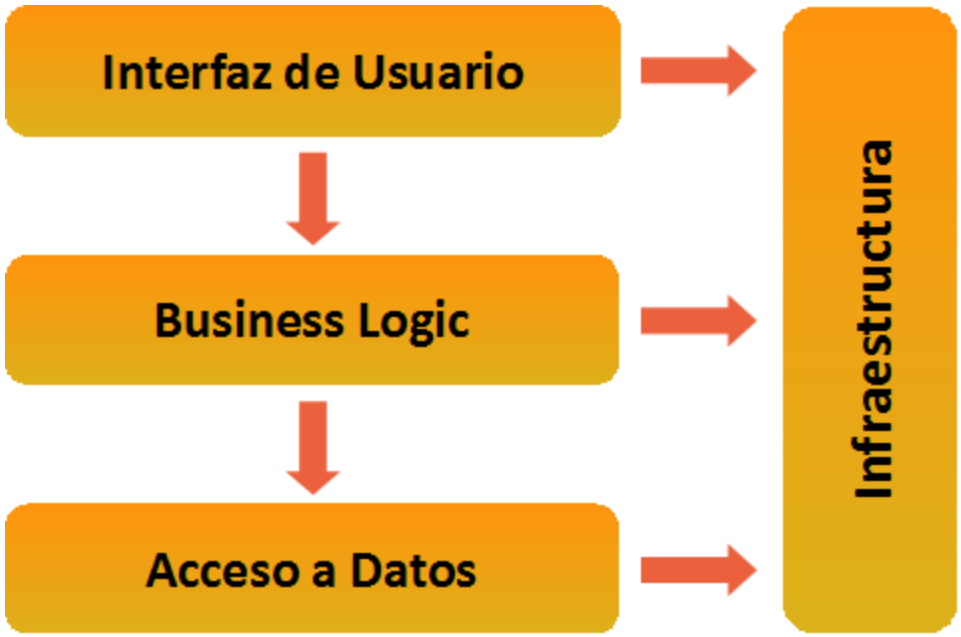

Figura 8: Representación de una Layered Architecture arquetípica

Por poner un ejemplo, la capa Business Logic depende de la capa Acceso a

Datos.

Sin embargo, si consideramos como un tipo de dependencia la dependencia transitiva, nos toparíamos entonces con que la capa Interfaz de Usuario depende de la capa Acceso a Datos.

A mayores, todas ellas dependen de la capa Infraestructura.

El mayor problema es que tanto la capa Interfaz de Usuario como la capa Business Logic dependen de la capa Acceso a Datos.

Visto de otra forma, la capa Interfaz de Usuario no puede funcionar si la capa Business Logic no se encuentra ahí, de la misma forma que la capa Business Logic, a su vez, tampoco puede operar si falta la capa Acceso a Datos.

Podemos argumentar que se pueden establecer varias variantes a este esquema arquitectónico que ayuden a mitigar este problema, pero la dependencia transitiva sigue haciendo estragos.

La conclusión es que esta Layered Architecture crea acople innecesario (unnecesary coupling).

Frente a esto, la Onion Architecture intenta controlar este tipo de acople innecesario.

Al hacer que las dependencias vayan hacia el centro, se aísla el Domain Model en la cadena de dependencias. La única dependencia que puede tener es consigo misma. Por esta razón la Onion Architecture funciona tan bien con el DDD.

2.6.3 Descripción de las capas de la Onion Architecture

El elemento central es el Domain Model, que viene a representar tanto el estado como el comportamiento que se encarga de modelar el problema que queremos resolver.

Los Domain Services suelen encargarse, entre otras tareas, de definir las interfaces con comportamiento asociado a la salvaguarda y recuperación de datos y comportamiento. En un entorno DDD, estaríamos hablando de los DDD_Repositories. Aun así conviene resaltar que son las interfaces y no sus implementaciones, las que se encuentran en esta capa. Es decir, tenemos el contrato pero no el comportamiento. Si queremos ir al comportamiento / implementación deberemos dirigirnos a la capa de Infraestructura.

Los Application Services suelen encargarse de la definición de cualquier interface que sea necesaria para poder hacer que la aplicación funcione, pero que sin embargo, no está ligada íntimamente con lo que oferta el Domain Model. Por poner un ejemplo, este sería el lugar adecuado para definir interfaces que luego vayan a ser implementadas por Controladores (la C de MVC) cuya implementación pertenecería a la capa Interfaz de Usuario.

Estas tres capas, conformarían el Application Core.

Por otro lado, se sitúan los elementos con más tendencia a cambiar, a saber:

- Interfaz de Usuario.
- Infraestructura.
- Tests.

No es casual que se encuentren en la capa exterior, ya que, de esta forma, nada de lo que hay en el interior puede depender de ellos.

En la capa Interfaz de Usuario nos encontraríamos con el código relativo a la VC de la tríada MVC. Nos referimos a sus implementaciones, ya que las interfaces que definen su comportamiento probablemente se sitúen en la capa Application Services.

En la capa Tests tendríamos los Tests del TDD o las BDD_Specs (Context / Specification) del BDD. Pero la Application Core no dependería de la capa Tests, sino que, por el contrario, son los Tests los que dependen y por tanto están acoplados al Application Core.

En la capa Infraestructura, se sitúa el código que trata directamente con la tecnología necesaria para llevar a cabo el proyecto, pero que no tiene sentido desde el punto de vista del cliente. Es decir:

- Acceso a datos.
- Entrada / Salida.
- Servicios Web.

Fuera de la propia estructura en forma de "cebolla", encontramos a los elementos ajenos a la POO, intencionadamente aislados del Application Core, comunicándose con el sistema a través de la Infraestructura.

Gracias a este esquema, logramos expulsar a las Bases de Datos del reino de los objetos, produciendo un desacople total entre el Application Core y la Base de Datos. El Application Core puede haber definido interfaces de algunos DDD_Repositories, pero su implementación se produce en la capa Infraestructura.

Por lo tanto, podemos decir que la Onion Architecture tiende descaradamente a la POO, ya que asigna un rol capital a los objetos.

2.6.4 La Onion Architecture y el DIP del S.O.L.I.D.

Para poder conseguir que todo esto funcione correctamente, la Onion Architecture descansa sobre el DIP.

El Application Core necesita de las implementaciones de las interfaces definidas tanto en Domain Services como en Application Services. Pero estas implementaciones se hallan en la frontera exterior de la aplicación. Por lo tanto es necesario algún mecanismo de inyección de dependencias que mantenga aglutinada toda esta estructura. Por eso se hace necesario el DIP.

3 Estado del Arte

> "I don't know what it was I professed to doing before I had added Domain-Driven Design and Test-Driven Development to my toolkit, but from my present perspective, I'm reticent to call it anything but chaotic hacking."
>
> Scott Bellware "Praise for *Applying Domain-Driven Design and Patterns*" (2004)

Si tomamos el presente proyecto como un todo, resulta difícil encontrar algo equivalente, o al menos alternativo, que podamos describir en esta sección. Tengamos en cuenta que la combinación de BDD, DDD y S.O.L.I.D. en el mundo .NET, desde la perspectiva que aquí abordamos, es suficientemente novedoso para no encontrar apenas documentación. No hablemos ya de un libro de referencia que trate en su conjunto todas las piezas de este puzzle.

Si atendemos al enfoque que hemos adoptado, creemos imprescindible nombrar el excelente libro de Jimmy Nilsson [22] al que le dedicamos la sección 3.2 unas lineas más adelante. También existe una cierta similitud entre sus planteamientos y los nuestros.

Si por el contrario atendemos a los elementos que conforman este proyecto, nos encontramos con que la única alternativa viable que podemos tratar es la del TDD, en lugar del BDD.

Ni Scrum, ni eXtreme Programming, pueden ser considerados "estado del arte" en cuanto a este proyecto se refiere, ya que, sus objetivos apenas se solapan (más bien se complementan) con lo aquí tratado, tal y como ya hemos explicado en los fundamentos teóricos cuando hablamos del BDD. La excepción habría que buscarla, precisamente, en el componente TDD del eXtreme Programming, que sí que sería reemplazado (o ampliado)

completamente por el BDD.

Por lo tanto, limitaremos el estado del arte a esas dos referencias.

3.1 Test-Driven Development

El Test-Driven Development es una Práctica de Diseño Agile, piedra angular de la Metodología Agile eXtreme Programming junto a las User Stories, los Ciclos Cortos de Desarrollo, los Acceptance Tests, el Pair Programming y el Refactoring, entre otras muchas prácticas.

Originalmente el TDD seguía las siguientes tres reglas:

1. No escribir una sola linea de código de la aplicación hasta que no hayamos escrito un Unit Test que falle.
2. No escribir más de un Unit Test que no compile o que falle.
3. Escribir únicamente el mínimo código necesario para que el test pase.

Atendiendo a estas 3 reglas tan simples, englobaríamos el TDD dentro del grupo de prácticas conocido como Test-First Programming, pues antes de codificar la funcionalidad en la aplicación, codificamos el Test que va a comprobar que dicha funcionalidad es correcta.

La mezcla que se produjo entre estas 3 reglas con el Refactoring (otro de los puntos cardinales del eXtreme Programming), dan lugar a una versión más moderna del TDD, que se rige por el mantra “RED-GREEN-REFACTOR”, que explicamos a continuación.

RED representa el fallo del test:

- Su función consistiría en cerciorarse de que efectivamente no se ha escrito una sola linea de funcionalidad de la aplicación, previamente a la definición / codificación del Test.
- Por tanto, atendiendo a las tres reglas listadas unas lineas más arriba, se correspondería, al menos, con la primera de ellas.
- Abarcaría el periodo en el que traducimos una parte del requisito capturado a través de una User Story en un Unit Test.
- El sentido de RED (color rojo) viene de que si no hemos programado absolutamente nada de la aplicación de forma previa a la codificación del Unit Test, éste, al ejecutarse, fallará. Se encontrará por tanto en una situación "errónea".
- La metáfora con un semáforo en un paso de cebra es clara:
  - Rojo = NO PASAR.
  - Rojo = NO SEGUIR CON EL SIGUIENTE TEST.
- La metáfora del semáforo nos indica que también sería garante de la segunda de las tres reglas originarias del TDD.

GREEN representa el test pasando:

- Estaríamos ante el escenario en el que ya se ha programado la funcionalidad que hace que se cumpla lo establecido en el Unit Test de la fase RED.
- Implícitamente se corresponde con la tercera y última de las tres reglas originarias del TDD, ya que se supone que para programar la funcionalidad requerida por el Unit Test, hemos debido hacer aquello de "escribir únicamente el mínimo código necesario para que el test pase".
- La metáfora del semáforo sigue siendo valida. Ahora:
  - Verde = PASAR.
  - Verde = PODEMOS SEGUIR CON EL SIGUIENTE TEST.
- La única salvedad a la metáfora del semáforo debemos buscarla en que antes de pasar al siguiente test, deberíamos evaluar si es necesario

realizar alguna refactorización, pues al "escribir el mínimo código necesario para que el test pase", podemos no estar escribiendo el código más adecuado para resolver el problema.

REFACTOR representa la fase en la que se revisa el código escrito en las dos fases anteriores (RED-GREEN):

- La actividad de "refactorizar" toma el nombre, intención y contenido del libro de Martin Fowler "Refactoring. Improving the Design of Existing Code" [32].
- Se considera que una refactorización es una forma de mejorar tanto el código como el diseño de un programa sin alterar ni un ápice la funcionalidad ya existente. Si se añadiese o eliminase alguna funcionalidad en el proceso de refactorización sería indicativo de que hay algo que no se está haciendo correctamente.
- Es importante entender que el proceso de refactorización afecta tanto al código de la aplicación como al código del Unit Test.
- La implicación más interesante de la incorporación de la refactorización al núcleo del TDD es que nos permite evaluar constantemente la calidad del código, dejando abierta la posibilidad de la modificación del mismo siempre que se considere necesario.

El cisma en el seno del TDD se dispara cuando se pone de manifiesto que hay dos enfoques muy diferentes a la hora de crear los Unit Tests:

- Podemos asertar sobre valores concretos. Por ejemplo: "el resultado de esta operación es 324".
- Podemos asertar sobre interacciones entre objetos. Por ejemplo: "para poder llevar a cabo esta operación necesitamos que colaboren estos dos objetos de una forma concreta".

Nos referimos a la primera como TDD Tradicional o Clásico (más formalmente State-Based Testing TDD), mientras que a la segunda la conocemos como TDD Mockista (más formalmente Interaction-Based Testing TDD).

El Interaction-Based Testing TDD es el germen del BDD y viene propiciado, desde un punto de vista técnico (que no filosófico), por la utilización de Frameworks de Mock Objects que nos permiten establecer condiciones en los objetos colaboradores tales como:

- "Se ha producido una llamada al método X de la clase Y".
- "Cuando se produce una llamada al método X de la Clase Y devuelve un valor Z."

Si dichas condiciones no se cumplen, el Unit Test falla (no se consigue completar la etapa GREEN).

El State-Based Testing TDD, por el contrario, establece condiciones directas sobre el resultado de una funcionalidad sin llegar a importarle como se llega a ese resultado desde el punto de vista de los objetos colaboradores. A saber:

- "El resultado del método X de la clase Y es el numero 53."
- "El resultado del método HolaMundo de la clase MiPrimeraClaseConCSharp es la cadena *Hola Mundo desde C#*."

Llegados a este punto, debemos aclarar que una gran parte de lo que se aborda en este proyecto, podría haberse conseguido de la misma forma con el TDD en lugar del BDD, siempre que se favoreciese el Interaction-Based Testing TDD, cuando fuese posible.

Sin embargo, decidimos utilizar el BDD en lugar del TDD, ya que, tal y como se explica en los fundamentos teóricos, el BDD no es más que una evolución

natural del TDD. Hay incluso quién dice que el BDD es el TDD tal y como se debería haber planteado desde el principio.

Desde un punto de vista de la tecnología, en cuanto al TDD se refiere, no hay ninguna alternativa tan potente y flexible como Machine.Specifications al BDD. NUnit, MbUnit, xUnit y demás suites de Unit Tests, semánticamente, no son comparables con Machine.Specifications y cuando se acercan, es precisamente porque incorporan nuevas extensiones más al estilo del BDD.

3.2 Applying Domain-Driven Design and Patterns

Publicado en el año 2006 y prologado por gente tan ilustre como Martin Fowler o Eric Evans, el libro de Jimmy Nilsson "Applying Domain-Driven Design and Patterns With Examples in C# and .NET" posee un valor intrínseco enorme, por su inteligente mezcla de texto y código, como si fuesen uno solo.

Cuando muestra código, no lo hace a través del ya consabido "Véase listado x", sino que utiliza los dos puntos (:) y a continuación muestra el código, como si se tratase de texto que es imprescindible leer y no como algo opcional (que es precisamente lo que sugiere la forma "Véase listado x").

Este tipo de integración, claramente ha inspirado nuestro estudio, ya que, en el fondo, lo que intenta transmitir es que el código es tan importante (o más) que el texto que lo acompaña. Algo así como romper una lanza a favor del código como muestra última de documentación.

En cuanto a la temática, Nilsson, invierte los términos con respecto a lo que se trata en el presente proyecto.

Tal y como indica su título, el enfoque del autor sueco, se basa en el DDD.

Para conseguir poner en juego todas las prácticas asociadas al DDD que considera interesantes, se vale del TDD en su versión State-Based Testing.

Desde luego en este punto encontramos una gran diferencia en cuanto al objetivo del proyecto de Nilsson (proyecto y no solo libro, en cuanto desarrolla una aplicación descargable y que además trasciende el mero ámbito del "ejemplo de libro") y el objetivo del presente proyecto. Nuestro enfoque se basa en el BDD y en como su práctica puede influenciar en un diseño DDD.

Pese a las diferencias (que las hay y son realmente importantes), no podemos dejar de reconocer que de una forma inconsciente, "Applying Domain-Driven Design and Patterns" ha sido un referente importante, no tanto por el contenido, como por el continente y sus formas. [22]

4 Fundamentos Tecnológicos

Una vez conocidos los objetivos y fundamentos teóricos necesarios para comprender y llevar a cabo el desarrollo del presente proyecto, nos centraremos ahora, en las herramientas utilizadas.

4.1 Entorno de Desarrollo: Visual Studio 2010

Es el IDE (Integrated Development Environment) por excelencia para desarrollar aplicaciones para la plataforma .NET.

Hay cuatro ediciones diferentes, de la más básica a la más compleja:

- 2010 Professional.
- 2010 Premium.
- 2010 Ultimate.
- Test Profesional 2010.

En nuestro caso concreto nos hemos decantado por la versión más básica (2010 Professional) ya que permite añadir diferentes extensiones que nos ayudan a personalizar el IDE, según nuestras necesidades.

Ésta es una forma de contrarrestar uno de los puntos débiles de los IDEs en general, a saber: el exceso de herramientas disponibles, puede contribuir a saturar la Interfaz de Usuario con una cantidad ingente de ruido. Y lo que es peor, contribuir a ocultar las pocas herramientas que sí son útiles para llevar a cabo nuestra tarea, de forma que nuestra productividad se vea afectada e incluso nuestras prácticas y procesos alterados. Llevando esto al extremo, tal y como se plantea en los Fundamentos Teóricos, correríamos el riesgo de acabar practicando Assumption-Driven Development en vez de Client-Driven

Development.

No es ésta una cuestión menor, por mucha capacidad de configuración que nos ofrezcan. De hecho, desde hace un par de años, existe una corriente muy importante en el mundo que rodea a .NET (y más en concreto, a ALT.NET), que aboga por volver a los orígenes en cuanto a las herramientas de desarrollo se refiere. Son los autoproclamados "Keyboard Jedis", que buscan convertir el desarrollo de aplicaciones en una experiencia Mouseless ("Sin ratón"). Esta corriente se está materializando en una vuelta al uso de editores de texto tan antiguos como Vi y Vim, en versiones especiales e incluso de pago como ViEmu.

Desde un punto de vista pragmático y pese a considerar como opción la experiencia Mouseless, por comodidad, hemos decidido decantarnos por la versión más sencilla de Visual Studio que permitiese la instalación de extensiones, puesto que, consideramos, de esta forma, retenemos lo mejor de dos mundos.

4.2 Lenguaje de Programación: C# 3.5

Por la temática tratada en el presente proyecto (BDD y DDD), la decisión era clara, C#. Si bien existe otra alternativa como Visual Basic, la experiencia previa con C# decantó la balanza.

La única decisión que había que tomar era la referente a qué versión de C# utilizar.

Teníamos una serie de requisitos previos que se materializaban en la necesidad de disponer de las siguientes características:

- Expresiones Lambda.
- Extension Methods.
- Tipos Anónimos.

Esto nos obligaba a usar, al menos, C# 3. Y esa es la versión escogida (si bien a lo largo del desarrollo del proyecto se utilizó también C# 4, demostrando que en lo que concerniente al proyecto, son perfectamente intercambiables).

Pero entonces, ¿por qué reza el epígrafe de este apartado como C# 3.5?.

Encontramos la respuesta en un malentendido tan extendido, que se ha convertido casi en norma.

La versión del Framework .NET que implementa el lenguaje C# 3 es la 3.5. De ahí viene que muchas veces se asimile el numero de la versión del Framework .NET con la del lenguaje C#, dando como resultado C# 3.5. En realidad no deja de ser una forma compacta de indicar que se está usando la versión C# 3 bajo el Framework .NET 3.5.

4.3 Librerias

4.3.1 Machine.Specifications

Machine.Specifications es un Framework Context/Specification (BDD, por tanto) Open Source, orientado a la eliminación del ruido provocado por el lenguaje y a la simplificación de los tests. La única condición que impone es la aceptación del operador Lambda "()=>" como algo esencial en la codificación de las BDD_Specs.

Su primera encarnación se produce en 2008 y se la debemos a Aaron Jensen. En cierta manera, podríamos considerarlo deudor de su anterior AutoMocking

Container, una combinación (o quizás más técnicamente, un miniDSL) de Rhino.Mocks y el Contenedor IoC/DI Windsor. Sin embargo, el propio autor, reconoce la deuda contraída con el Framework Spec.Unit.NET del gurú del BDD canadiense Scott Bellware, así como del Framework BDD para Ruby RSpec.

Desde un punto de vista más técnico, hace un uso intensivo de los Delegados y las Funciones Anónimas, en aras de la legibilidad de las BDD_Specs escritas.

Su código fuente se puede encontrar en github:

http://github.com/machine/machine.specifications

En la actualidad, ha dejado de ser obra exclusiva de Jensen, y cada vez más colaboradores hacen aportaciones al mismo. [37][38][39]

4.3.2 Machine.Specifications.DevelopWithPassion

DevelopWithPassion son las extensiones que Jean-Paul Boodhoo ha incorporado al proyecto original Machine.Specifications. A través de ellas adapta todas las características de su librería previa DevelopWithPassion.BDD.

No existe apenas documentación, ni información, acerca de estas extensiones, si exceptuamos el repositorio github del propio proyecto, localizado en:

http://github.com/developwithpassion/machine.specifications

donde podemos consultar directamente el código fuente de las extensiones.

A mayores, existe un screencast del autor donde muestra un uso básico de las mismas [41].

La mejor manera de informarse de la utilidad de las extensiones DevelopWithPassion es leer el Estudio del presente proyecto.

Un resumen de su funcionalidad y objetivos sería:

- Proveer de un mecanismo sencillo que permita la creación y el registro de instancias "mockadas" de las dependencias del sistema que pretendemos probar (vía Rhino.Mocks).
- Proveer de un mecanismo sencillo que permita la creación de mock objects para poder ser usados en cualquier punto de las BDD_Specs (vía Rhino.Mocks).
- Crear y ofrecer una instancia del SUT (System Under Test), con todas sus dependencias registradas e inyectadas, en base a la interface que defina el tipo o al tipo concreto. [40][41][42]

4.3.3 Rhino.Mocks

Es un Framework de Mock Objects dinámicos para la plataforma .NET, que debemos al israelí Oren Eini (más conocido como Ayende Rahien), una de las mentes detrás de proyectos tan conocidos como NHibernate o la suite Castle.

Su propósito básico es ofrecer, bajo licencia BSD, una herramienta que facilite el proceso de prueba del software, permitiendo al desarrollador la creación de implementaciones Mock para cualquier objeto, así como la verificación de las interacciones de los mismos usando Unit Testing.

Las características principales de Rhino.Mocks son:

- Implementación de la sintaxis Arrange-Act-Assert (AAA).
- Ofrecer la posibilidad de trabajar con Mocks fuertemente tipados.
- Permitir la definición de expectativas basadas en diferentes tipos de

criterios.

- Facilitar la definición de acciones sobre métodos, obteniendo como resultado:
  - La devolución de un valor concreto.
  - Una excepción. [43][44]

4.4 Utilidades

4.4.1 Git y GitHub

Git es un Sistema de Control de Versiones Distribuido con licencia Open Source.

Se lo debemos a Linus Torvalds, quien en 2005, creó Git con el objetivo de mejorar los sistemas de versiones de entonces, para poder administrar las versiones del Kernel de Linux de una forma más eficiente.

Originalmente se pretendía crear un Sistema de Control de Versiones con las siguientes características:

- Velocidad.
- Sencillez en cuanto al diseño.
- Serio soporte para desarrollo no lineal (cientos de ramas paralelas).
- Completamente distribuido.
- Capaz de manejar proyectos de grandes dimensiones.

Git ha evolucionado mucho desde su concepción inicial, pero ha conseguido mantener intactos sus puntos fuertes, como un sistema de creación/fundido de ramas sencillo y completamente revolucionario, mientras que ha limado algunas de sus asperezas, pues por ejemplo, ahora es algo más sencillo de usar.

La gran alternativa a Git dentro del mundo distribuido es Mercurial (hg). Si buscamos algo en el mundo no distribuido, Subversion (svn) es la elección lógica.

Atendiendo a las apreciaciones de Martin Fowler [46], Mercurial se sitúa ligeramente por encima de Git, en cuanto a sistema recomendable se refiere (Ver figura 9).

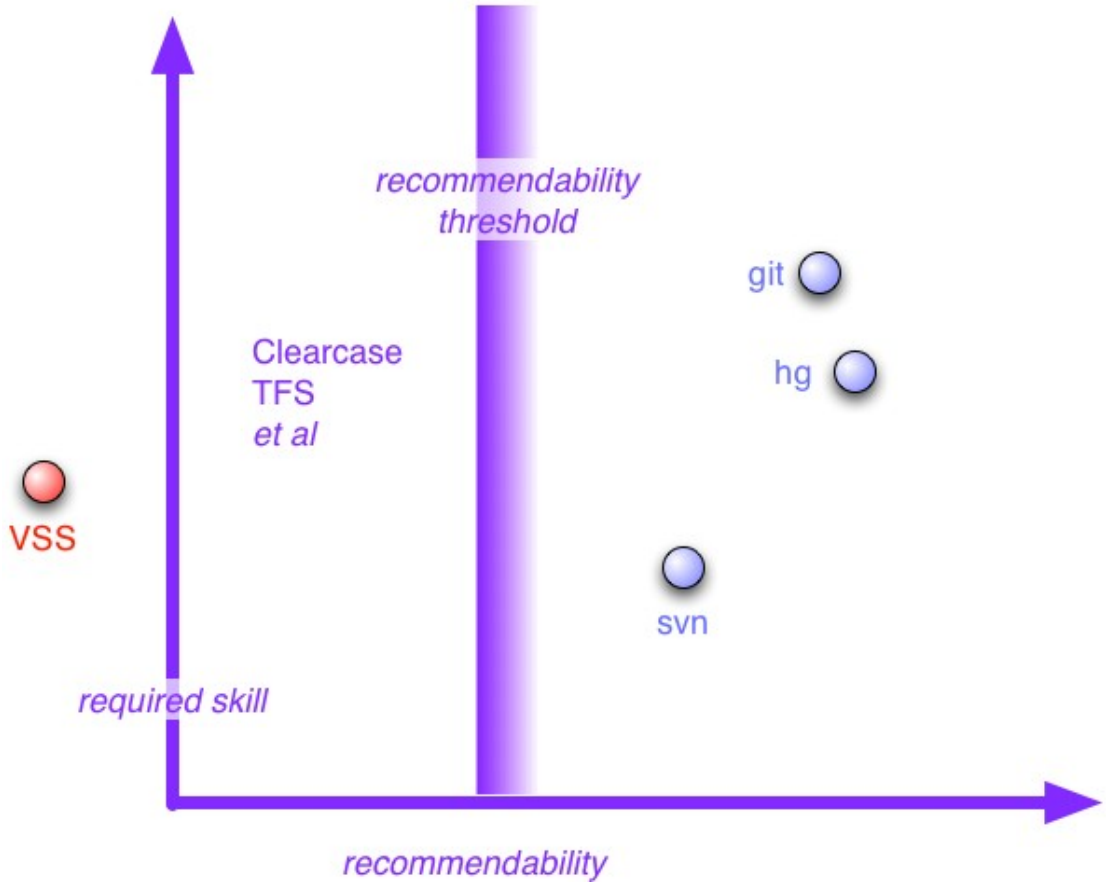

Figura 9: Comparativa de Sistemas de Control de Versiones según Fowler

En nuestro caso, el peso de GitHub (repositorio online de Git) frente a BitBucket (repositorio online de Mercurial), no solo equilibra la balanza, sino que la desequilibra en favor de Git. De ahí la razón de que usemos Git y no Mercurial en el presente proyecto. [45][46][47]

4.4.2 ReSharper

JetBrains ReSharper es una herramienta que extiende la funcionalidad de las diferentes versiones de Visual Studio existentes, con la refactorización y la productividad en mente.

Es tan importante dentro del mundo TDD y BDD, que sería inimaginable trabajar sin ella.

Algunas prácticas asociadas a estas técnicas de diseño; como el nombrado de Tests/Specs o el propio nombrado de clases, interfaces y demás elementos de la programación; se ha visto alterado gracias a la existencia de una herramienta como ReSharper, que nos permite navegar a velocidades espectaculares entre todos estos elementos.

Otra actividad asociada al TDD y BDD que se ha visto influenciada por las estupendas características del ReSharper es la conocida como Ping-Pong. Ésta consiste en alternar la codificación de la BDD_Spec y la codificación de las clases/interfaces que hacen que funcione. Gracias a ReSharper, podemos realizar esa operación de alternancia en décimas de segundo, integrando estas dos actividades tan diferentes, en un solo proceso.

La única herramienta que parece hacerle algo de sombra es DevExpress CodeRush. Pero si se ha probado JetBrains ReSharper, es realmente difícil prescindir de ella y valorar objetivamente la competencia.

4.4.3 TestDriven.NET

TestDriven.NET es un complemento para Visual Studio. Tiene por objeto facilitar la tarea relativa a la ejecución de una suite de Tests o BDD_Specs. Esta tarea se convierte en algo rápido y sencillo, y no añade ningún tipo de fricción al proceso. Estaríamos, por tanto, ante una herramienta totalmente enfocada a aquellos desarrolladores que utilicen las Prácticas de Diseño Agile TDD o BDD.

Entre sus principales características, podemos resaltar que nos permite:

- Ejecutar cualquier Unit Test situado en un método, clase, namespace, proyecto o solución de forma directa (con un solo click de ratón o un atajo de teclado).
- Depurar cualquier Unit Test a través de las herramientas que incluye Visual Studio al respecto, como si fuese código de la propia aplicación en desarrollo.
- Ejecutar la suite de Unit Tests para ser evaluada por NCover, que nos facilita el análisis de la cobertura conseguida por nuestros Unit Tests en nuestro código (Code Coverage).

En la actualidad da un soporte limitado para BDD y Machine.Specifications, pero aún así, podemos sustituir Unit Test por BDD_Spec, en el listado de características anterior, sin demasiado problema.

Las alternativas a TestDriven.NET para BDD son escasas. La única seria sería tomar el camino del "modo texto". Es decir, crear una serie de ficheros de configuración con una herramienta como NAnt, que nos permitiesen ejecutar en una ventana de comandos distintas tareas como:

- Correr los tests.

- Compilar el código.
- Ejecutar el código.

Se trata de una posibilidad muy interesante y no-excluyente con respecto a la que hemos utilizado aquí. De hecho, aparecía como opción en el Anteproyecto, pero usando la combinación Ruby / Rake / Albacore en vez de NAnt. De esa forma nos mantendríamos alejados de los ficheros de configuración XML propios de NAnt, y a cambio dispondríamos de un lenguaje de programación orientado a objetos, tan potente como Ruby, y tareas ya predefinidas a través de Rake y Albacore.

Para evitar una mayor complejidad en un proyecto suficientemente complejo de por sí, se decidió no tomar el "Camino Ruby", al no aportar nada interesante al objeto del mismo.

5 Viabilidad

En la actualidad, no existe ningún tipo de Estudio o Guía que satisfaga los objetivos del presente proyecto.

Si la documentación al respecto es escasa en inglés, no digamos ya en castellano, donde las referencias tanto a la Práctica de Diseño Agile BDD como a los Principios de Diseño S.O.L.I.D., son prácticamente nulas; mientras que las referencias a las Prácticas de Diseño DDD son marginales, cuando no directamente cuestionables o erróneas.

Es comprensible que no exista una literatura en castellano asociada al BDD, puesto que, estamos hablando de algo muy puntero y todavía en fase de evolución. De hecho, tampoco existe esa literatura en inglés.

Sin embargo, la razón por la cual no hay una extensa bibliografía en castellano asociada tanto al DDD como a los Principios S.O.L.I.D., escapa a nuestro entendimiento. Pues, estamos hablando de temas más asentados y que han provocado ríos de tinta al otro lado del Atlántico, por poner un ejemplo.

En cierta forma, hemos tratado esto mismo en el capitulo 3, cuando establecíamos las dificultades con que nos habíamos topado al necesitar establecer el Estado del Arte.

Por todo ello, entendemos que existe una necesidad imperiosa de poder establecer un punto de referencia en castellano para las últimas tendencias en la Comunidad Agile. Es aquí donde creemos que este proyecto muestra su fuerza e importancia, ante la perspectiva de situarse como dicho punto de referencia sobre el que, posteriormente, poder establecer un necesario debate que glose las virtudes y defectos de los planteamientos aquí mostrados.

Asimismo, consideramos que podría erigirse como la plataforma natural para poder comenzar el aprendizaje de las Técnicas y Prácticas desarrolladas.

Por último, debemos destacar que no estamos ante un Estudio Teórico al uso. Muy al contrario, tal y como demuestra el software asociado, estamos ante una realidad demostrada de forma empírica.

6 Diseño, Metodología, Implementación y Contenido

> "The process is messy, as all human processes are. The result: Well, the order that came out of such a messy process is amazing."
>
> Robert C. Martin "Agile Principles. Patterns, and Practices in C#"

6.1 Estableciendo el Diseño asociado al Estudio

A la hora de establecer el Diseño asociado al Estudio, así como el tono en el que debía ser desarrollado, se tuvieron en cuenta las siguientes fuentes como posibles modelos a seguir:

6.1.1 Jimmy Nilsson: "Applying Domain-Driven Design and Patterns With Examples in C# and .NET"

La obra de Nilsson [22] nos parece fabulosa. Es innegable que existe una cierta similitud entre sus planteamientos y los nuestros. Por esa razón es incluido en el Apartado 3.2 del Estado del Arte.

Sin embargo, en cuanto al diseño del Estudio, teníamos claro que no coincidíamos.

Donde Nilsson ofrece todas las alternativas posibles para resolver un problema, glosando virtudes y defectos de las mismas, en nuestro caso buscábamos una aproximación mucho más directa y práctica a la solución de un problema concreto, para finalmente, en base a ello, sacar algunas conclusiones.

Por lo tanto, la aproximación Nilsson no es válida, en nuestro caso.

Ante esto, pasamos a consultar otra literatura cuya temática no tuviese tanto que ver con la nuestra, pero cuyo enfoque hacia la resolución del problema sí que pudiese aportar ideas en la dirección correcta. En concreto, se consideraron tanto la serie "Head First" de O'Reilly como la serie con el sufijo "In Action" de Manning.

6.1.2 La serie "Head First" de O'Reilly

Inicialmente, la serie "Head First" de O'Reilly demostró ser una opción viable. Llena de ilustraciones y con un lenguaje llano, pero sin llegar a tratar de forma condescendiente al lector. Hubiese sido un serio contendiente si el objetivo principal del presente proyecto fuese escribir un tutorial más o menos artificial sobre, pongamos por caso, la práctica del BDD. Sin embargo, sin pretender descartar de inicio la posibilidad de que el presente proyecto cumpla una función de guía, teníamos claro que nuestros objetivos eran mucho más amplios.

Por lo tanto nos vimos obligados a seguir buscando. [50][51]

6.1.3 La serie "In Action" de Manning

A favor de la serie "In Action" de Manning jugaba el hecho de que gracias a ella, habíamos aprendido diferentes tecnologías, prácticas y conceptos. Así que nuestra familiaridad con esa forma concreta de atacar un tema nos proporcionaba cierta ventaja.

Sin embargo, precisamente gracias a ese conocimiento tan cercano, éramos conscientes de sus limitaciones. Si bien había grandes lecciones que aprender en cuanto al enfoque utilizado por la Serie de Manning, dicho enfoque todavía distaba algo del que teníamos en mente.

La serie "In Action" es eminentemente práctica, pero suele introducir primeramente el concepto teórico para a continuación mostrarlo de forma práctica. Nuestra idea inicial, era precisamente la opuesta. Demostrar de forma práctica los conceptos y más adelante dejar caer el concepto teórico si fuese pertinente o intentar estimular la curiosidad del lector lo suficiente como para que decidiese acudir a las fuentes originales, en las que sí queda clara la motivación y base teórica de los mismos.

A mayores, había otra cuestión que no habíamos solucionado todavía, y es que pretendíamos que el presente Estudio documentase de forma fidedigna la metodología seguida en el desarrollo del software. Expresándolo en Agile, buscábamos compartir una sesión de Pair Programming de la forma más realista posible, lo que nos lleva a nuestra siguiente referencia, el Capítulo 6 "A Programming Episode" de "Agile Principles, Patterns and Practices in C#" de Robert C. Martin ("Uncle Bob") y su hijo Micah Martin. [48][49]

6.1.4 Sesión de Pair Programming de Robert C. Martin y Ross Koss en "Agile Principles, Patterns and Practices in C#"

En este capítulo Ross Koss y Bob Martin comparten una sesión de Pair Programming donde desarrollan un programa que calcula la puntuación de un partido de bolos, a través de la recreación de la conversación mantenida por ambos en una habitación de hotel en el año 2000.

Esta aproximación nos parecía muy interesante. Sin embargo, el estilo conversación llegaba a resultar tedioso hacia el final del capítulo. Teniendo en cuenta que en la edición impresa del libro estamos ante unas 50 páginas escasas, no parecía una buena idea mantener ese estilo para un Estudio que ha acabado teniendo del orden de 650 páginas (depuradas de más de 800).

Pese a los inconvenientes, sí que hay una serie de cuestiones que se acercaban a lo que buscábamos. A saber:

- Se documenta un Proceso Agile con todo detalle (pese a ser, únicamente, una recreación más o menos fidedigna de la conversación).
- Se mezcla texto/conversación con código de una forma tal, que hace que el código sea tan protagonista como el propio texto. La consecuencia más interesante es que nos vemos en la obligación de leer y comprender el código justo en el momento en el que aparece, ya que, de lo contrario, perdemos el hilo de la conversación.
- Se documentan todos los errores cometidos de forma realista (tanto de código, como de diseño, como de lógica o asociados a una interpretación defectuosa de los requisitos). Y todo ello sin que se nos advierta de antemano de que nos encontramos ante un camino erróneo.
- Se admite la naturaleza confusa asociada al diseño y desarrollo de software.

Se pueden encontrar trazas de estos cuatro puntos en el Estudio. [3]

6.1.5 Destilando el Diseño del Estudio

Tras la evaluación de las diferentes fuentes podemos establecer que el diseño, tono y estilo utilizado en el Estudio asociado al presente proyecto, es una mezcla de las distintas aproximaciones tratadas anteriormente:

- De Nilsson [22], sacamos la integración texto/código, así como utilizamos parte de su concepción temática como prueba de la validez de nuestro planteamiento.
- De la serie "Head First" [50][51] adoptamos la necesidad de conseguir que la apariencia de lo que se presenta sea lo más amena y llamativa posible sin por ello resultar cargante. La traducción directa de ésto,

posiblemente haya que buscarla, en el tratamiento de color dado al código, así como, los distintos formatos utilizados según lo que se quisiese mostrar (por ejemplo los logs resultantes de la ejecución de la suite de BDD_Specs).

- De la serie "In Action" [48][49], sacamos la inspiración necesaria para ordenar el material que queríamos tratar de una forma coherente, incluyendo el uso de Apéndices para temas necesarios pero no directamente relacionados con el proceso documentado en el grueso del Estudio.
- De la sesión Pair Programming entre Bob Martin y Ross Koss [3], sacamos los medios necesarios para documentar de forma fidedigna el proceso que nos lleva a solucionar un problema a través del software. Admitiendo el error como algo inherentemente humano en el proceso de aprendizaje y solución de problemas.

6.2 Metodología del Software asociado

La metodología seguida en la creación del software asociado al presente proyecto, estaba clara ya desde la concepción del Anteproyecto.

Puesto que buscábamos reproducir de forma fidedigna un entorno en el que aplicar la Práctica de Diseño Agile BDD (principalmente), la metodología no debía entorpecer dicho objetivo.

Contextualizando el uso que se puede dar al BDD, nos veíamos en la obligación de evitar tomar partido por cualquiera de las Metodologías Agile, de forma que, por poner un ejemplo, si alguien pretendiese introducir todo lo que se muestra en este proyecto en un Entorno Agile dominado por la combinación de las Metodologías Scrum y eXtreme Programming, las prácticas mostradas no obligasen al equipo a modificar dicho Entorno.

Con el objetivo de respetar esos parámetros, se establecieron micro-iteraciones (de 2 a 15 minutos, aproximadamente) a nivel de BDD_Spec. Es decir, por cada BDD_Spec (Context / Specification) realizaremos un ciclo completo del proceso.

Este ciclo debería ser compatible con cualquier Metodología Agile. Es más, el establecimiento de micro-iteraciones debería favorecer una estimación de requisitos mucho más precisa, ya que como máximo, cada 15 minutos obtendríamos feedback real (queremos recalcar aquí el término "real" frente a "estimado") que nos permitiría ajustar las predicciones.

Por lo tanto, la metodología utilizada en el desarrollo del software orbita alrededor del concepto de la micro-iteración. Dicha metodología consta de las siguientes fases o etapas:

1. **Consecución de la última versión del software**. Nos situamos en la rama (branch) correcta y descargamos la última versión del código del repositorio albergado en GitHub. Gracias a este punto, sabemos que disponemos siempre de la versión más reciente del código con todos los cambios efectuados hasta el momento. Hay que tener en cuenta que este aspecto es de capital importancia en el entorno de un equipo, pues nosotros no vamos a ser los únicos que tengamos la capacidad de modificar el código.
2. **Definición formal de la BDD_Spec**. Definimos la BDD_Spec (Context / Specification) formalmente a través de lenguaje natural. Con esto pretendemos establecer de forma clara qué parte de los requisitos de la aplicación vamos a abordar en la presente micro-iteración.
3. **Codificación del Contexto de la BDD_Spec**. Nombramos el Contexto a través de la clase que alberga la BDD_Spec tal cual aparece definido en el punto anterior. Éste sería el primer paso en el que escribimos

código, a través del nombrado de la clase como el Contexto de la BDD_Spec siguiendo la plantilla "When ...".

4. **Codificación de las Especificaciones de la BDD_Spec**. Escribimos las Especificaciones, respetando, al igual que en el punto anterior, lo establecido en la BDD_Spec formal, palabra a palabra. En este caso estaríamos definiendo las Especificaciones en forma de Aserciones a través de los bloques IT proporcionados por Machine.Specifications siguiendo la plantilla: "It should ...". En este punto, por tanto, ejercitamos la primera parte del estilo de testing conocido como AAA (Arrange-Act-Assert), pero en sentido inverso (empezamos escribiendo el Assert).
5. **Codificación de la ejercitación del SUT**. Ejercitamos el SUT (System Under Test) a través del bloque BECAUSE proporcionado por Machine.Specifications siguiendo la plantilla: "Because of ... sut ..." y cumpliendo la parte ACT del AAA. En esta etapa, nos ponemos del lado del código cliente, pues al obligarnos a hacer la llamada al método que va a proporcionar el comportamiento deseado cuando ni siquiera existe el nombre del mismo, estamos diseñándolo. De esta forma, buscamos la frase que mejor exprese nuestra intención, así como sus dependencias en forma de signatura.
6. **Codificación de las condiciones bajo las que se ejecuta la BDD_Spec**. Establecemos las condiciones en las cuales va a tener sentido la ejecución del SUT y por tanto de la BDD_Spec. Para ello nos valemos del bloque ESTABLISH proporcionado por Machine.Specifications. Seguimos la plantilla "Establish context ..." y cumplimos con el ARRANGE del AAA.
7. **Compilación del código de la BDD_Spec**. Procedemos con la compilación del código de la BDD_Spec, haciendo todo lo necesario para conseguirlo, sin tener por ello que codificar funcionalidades nuevas del sistema. Este paso lo hemos ido resolviendo en parte, en las etapas anteriores.

8. **Fallo significativo de la BDD_Spec**. Buscamos el fallo de la BDD_Spec, pero no cualquier fallo. Buscamos el fallo esperado. De esta forma comprobamos, entre otras cosas, que la BDD_Spec no se satisface por un error nuestro, evitando escenarios potencialmente dañinos. Un ejemplo sería el escenario en el que la BDD_Specs se cumpliese siempre (no fallase nunca), independientemente de la codificación que vamos a hacer de la funcionalidad definida por la misma. Esta fase corresponde con el RED del RED-GREEN-REFACTOR.
9. **Implementación de la funcionalidad definida por la BDD_Spec**. Escribimos el código que implementa la funcionalidad y que permite que la BDD_Spec se cumpla. Hay que tener en cuenta que en el proceso de codificación debemos intentar mantenernos lo más cerca posible de la máxima del TDD "Escribe el mínimo código necesario". Esta fase marca el inicio del GREEN en el RED-GREEN-REFACTOR.
10. **Comprobación de que la BDD_Spec se cumple**. Comprobamos que el código hace que la BDD_Spec se cumpla. Esta fase corresponde con la finalización del GREEN del RED-GREEN-REFACTOR.
11. **Refactorización de todo el código escrito**. Modificamos el código tanto como sea necesario sin añadir ni quitar funcionalidades. Para ello, atendemos a los casos y prácticas definidos por Fowler [32] para el código de la aplicación y a los casos y prácticas definido por Meszaros [52] para el código de las BDD_Specs. Ambas referencias son la base de las refactorizaciones, pero existen más cambios posibles, como los que buscan que nuestro código cumpla con los Principios de Orientación a Objetos S.O.L.I.D. [3], o aquellos que únicamente pretende mejorar la legibilidad del código. Muchas veces nos encontraremos en la situación de refactorizar hacia patrones clásicos del Gang of Four [13] u otros autores tal y como detalla Kerievsky [53]. Una última aproximación a la refactorización puede venir provocada por las Practicas de Diseño DDD. Pero en este caso, suelen ir más allá de una refactorización, haciendo necesario una reevaluación de los requisitos, que desembocarían en una

nueva micro-iteración. Esta fase corresponde con el REFACTOR del RED-GREEN-REFACTOR.

12. **Check-In Dance para salvar el código escrito**. Representa el final de la micro-iteración, y por tanto el momento en el que gracias al Check-In Dance, estamos listos para compartir el código con el equipo. Representa el punto en el que la funcionalidad definida por la BDD_Spec está programada y revisada.

Ésta es, por tanto, la metodología utilizada para desarrollar el código del presente proyecto. Sin embargo debemos señalar que sobre esta metodología de la micro-iteración íntimamente ligada a la Práctica de Diseño Agile BDD, establecemos unas modificaciones que nos permiten adaptarla a un escenario de Legacy Code como el definido por Feathers [54] o Brownfield Development como el definido por Baley y Belcham [55], para el que inicialmente no estaba pensada. Dichas modificaciones se materializan en el Capitulo 3 del Estudio Adjunto (Value Object Itinerary – Aplicando BDD a código preexistente).

6.3 Adaptando el proceso de creación del Estudio a la Metodología del Software.

Una vez estaban claros los puntos referentes al diseño del estudio y la metodología del software, debíamos conseguir que el proceso de documentación del mismo, del cual el Estudio es su principal depositario, fuese lo más fidedigno posible.

Para ello, decidimos que la aproximación más interesante era la de documentar el proceso no solo al nivel de la micro-iteración (antes y después de la misma), sino al nivel de cada una de las etapas de la micro-iteración. Por tanto, antes y después de cada fase de la metodología de la micro-iteración procedíamos con la documentación de lo observado, así como, hacíamos todos las consideraciones, razonamientos y expectativas que considerábamos

oportunas.

6.4 Contenido del Estudio

El Estudio se divide en dos partes claramente diferenciadas:

- Cuerpo del Estudio.
- Apéndices del Estudio.

El Cuerpo del Estudio detalla el procedimiento seguido para conseguir implementar una parte representativa del Domain Model escogido. Está dividido en 7 capítulos, en los que se trata el desarrollo asociado a un elemento DDD del Domain Model:

- El DDD_Aggregate_Root Cargo (Capítulos 1 y 6).
- El DDD_Value_Object RouteSpecification (Capítulo 2).
- El DDD_Value_Object Itinerary (Capítulo 3).
- El DDD_Value_Object RouteStatus (Capítulo 4).
- Refactorización del DDD_Factory RouteSpecificationFactory (Capítulo 5).
- El DDD_Aggregate_Root Location (Capítulo 7).

En los primeros capítulos, el nivel de detalle en cuanto a la descripción de las micro-iteraciones es alto. A medida que avanzamos y hemos ido practicando las bases del BDD, vamos introduciendo nuevas discusiones que se alejan más de la parte mecánica para adentrarse en cuestiones relativas al diseño, como pueden ser:

- La aplicación de los Principios de Diseño S.O.L.I.D. y sus consecuencias.
- La introducción de los patrones del DDD.
- La importancia de la Interface-Based Programming.
- El impacto de Interaction-Based Testing TDD y el State-Based Testing

TDD en el BDD.

Al llegar al Capítulo 7 culminamos el estudio con la colisión BDD-DDD.

Mención aparte merece el capítulo dedicado al DDD_Value_Object Itinerary (Capítulo 3), pues sirve como ejercicio simulado de Legacy Code [54] o Brownfield Development [55]. Este capítulo lleva la Metodología BDD de la micro-iteración al límite y demuestra como incluso ante las circunstancias más adversas, sale bien parada.

Los Apéndices del Estudio reúnen el material más heterodoxo y que, pese a ser necesario, no tenía acomodo claro en el Cuerpo del Estudio:

- Introducción al BDD con Machine.Specifications (Apéndice I).
- Las Interfaces DDD (Apéndice II).
- Implementando el Patrón DDD_Specification (Apéndice III).
- Primeros paso con Git (Apéndice IV).

Los Apéndice I y IV pueden ser utilizados a modo de introducción antes de atacar el Cuerpo del Estudio. Con un nivel de detalle alto, explican de forma práctica, la mecánica detrás de las micro-iteraciones (sin llegar a nombrarlas como tales).

El Apéndice II sienta las bases de los Patrones principales del DDD que vamos a utilizar. Su lectura es optativa, si bien ayudan al neófito del DDD a entender los conceptos básicos.

El Apéndice III fue lo primero que se escribió del Estudio y también lo primero que se escribió del software. Su complejidad es considerable y a no ser que uno sea un especialista en DDD, debe ser consultado una vez concluido el Cuerpo del Estudio. En caso de tener unos conocimientos avanzados de DDD,

éste es la introducción perfecta al BDD, ya que se detalla concienzudamente todo el proceso seguido.

7 Conclusiones

Consideramos que hemos cubierto los siguientes objetivos:

- Hemos llenado el vacío existente relativo a la documentación y desarrollo de un Domain Model desde un prisma Agile-BDD.
- Se ha establecido un punto de referencia en castellano sobre el que mostrar las últimas tendencias de diseño de la Comunidad Agile.
- Se ha demostrado empíricamente la viabilidad de una aproximación al diseño de software BDD-DDD híbrida.
- Se ha proporcionado un software de referencia que puede y debe ser ampliado.
- Se ha demostrado la viabilidad del BDD como sucesor del Interaction-Based Testing TDD.
- Se ha documentado con abundantes ejemplos el framework BDD Machine.Specifications así como sus extensiones DevelopWithPassion.
- Se ha demostrado la capacidad de adaptación del BDD como Práctica de Diseño Agile ante escenarios adversos de Código Heredado (Legacy Code o Brownfield Development).
- Se ha demostrado el impacto en cuanto al diseño se refiere de los procesos Test-First Development.
- Se ha demostrado a través de escenarios prácticos el camino para eliminar el desperdicio (waste) desde una filosofía Lean.
- Se ha demostrado que es factible desarrollar un Domain Model sin dependencias ni con otras capas, ni con infraestructura, en un contexto de la Onion Architecture.
- Se provee de un Estudio Práctico de gran valor pedagógico.

8 Lineas Futuras

Uno de los objetivos del presente proyecto era el establecimiento de un punto de referencia en castellano para las últimas tendencias en la Comunidad Agile. Por lo tanto, desde su concepción, el presente proyecto ha tenido en cuenta la posibilidad de la extensión del mismo. En base a esto, establecemos cinco posibles lineas futuras no excluyentes entre si:

- Implementación del Patrón DDD_Repository en combinación con NHibernate (y/o Entity Framework), siguiendo la Práctica de Diseño Agile BDD.
- Implementación de una interfaz gráfica que siga el Patrón MVC bajo ASP.NET MVC, siguiendo la Práctica de Diseño Agile BDD.
- Finalización de todos los elementos que participan en el Domain Model utilizado.
- Incorporación de un Framework IoC/DI como Ninject! para automatizar la resolución del DIP del S.O.L.I.D..
- Implementación de las capas restantes que entran en juego en la Onion Architecture.

Cualquiera de estas cinco lineas futuras deberían ir acompañadas de su Estudio Práctico, donde quedase expuesto el proceso por el cual se llega a esa solución concreta.

9 Glosario de Acrónimos

AAA: Arrange-Act-Assert.

BDD: Behavior-Driven Development o Behaviour-Driven Development.

BDD_Spec: Context / Specification.[1]

BDUF: Big Design Up Front..

DDD: Domain-Driven Design.

DI/IoC: Dependency Injection / Inverse of Control.

GoF: Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides).

IDE: Integrated Development Environment.

MVC: Model-View-Controller.

NDUF: No Design Up Front.

POO: Programación Orientada a Objetos.

S.O.L.I.D.: 5 principios listados a continuación

- **SRP**: Single-Responsability Principle.
- **OCP**: Open/Closed Principle.
- **LSP**: Liskov Substitution Principle.
- **ISP**: Interface Segregation Principle.
- **DIP**: Dependency Inversion Principle.

SUT: System Under Test.

TDD: Test-Driven Development.

XP: eXtreme Programming.

YAGNI: You Ain't Gonna Need It.

1 Acrónimo creado por nosotros.

10 Bibliografía

[1] Eric Evans.
«*Domain-Driven Design: Tackling Complexity in the Heart of Software*».
Addison-Wesley, 2004.

[2] Robert C. Martin.
«*Agile Software Development: Principles, Patterns and Practices*».
Prentice Hall, 2002.

[3] Robert C. Martin and Micah Martin.
«*Agile Principles, Patterns and Practices in C#, 2nd Printing*».
Prentice Hall, 2007.

[4] Wikipedia.
«*Lean Software Development*».
http://es.wikipedia.org/wiki/Lean_software_development
Acceso Agosto 2010.

[5] BDDWiki:
«*Behaviour-Driven Development Introduction*».
http://behaviour-driven.org/Introduction
Acceso Agosto 2010.

[6] BDDWiki:
«*Getting the words right*».
http://behaviour-driven.org/GettingTheWordsRight
Acceso Agosto 2010.

[7] BDDWiki:

«*Enough is Enough*».
http://behaviour-driven.org/EnoughIsEnough
Acceso Agosto 2010.

[8] BDDWiki:
«*Where's the Business Value*».
http://behaviour-driven.org/WheresTheBusinessValue
Acceso Agosto 2010.

[9] BDDWiki:
«*It's all Behaviour*».
http://behaviour-driven.org/ItsAllBehaviour
Acceso Agosto 2010.

[10] Dan North.
«*Introducing BDD*».
http://blog.dannorth.net/introducing-bdd/
Acceso Agosto 2010.

[11] Wikipedia.
«*Behavior-Driven Development*».
http://en.wikipedia.org/wiki/Behavior_Driven_Development
Acceso Agosto 2010.

[12] Scott Bellware.
«*Behavior-Driven Development*».
http://www.code-magazine.com/article.aspx?quickid=0805061
CoDe Magazine, May/June 2008.

[13] Eric Gamma, Richard Helms, Ralph Johnson and John Vlissides
«*Design Patterns: Elements of Reusable Object-Oriented Software*».

Addison-Wesley, 1994.

[14] Martin Fowler and others.
«*Patterns of Enterprise Application Architecture*».
Addison-Wesley, 2003.

[15] Martin Fowler.
«*Domain Event*».
http://martinfowler.com/eaaDev/DomainEvent.html
Acceso Agosto 2010.

[16] Udi Dahan.
«*Domain Events - Salvation*».
http://www.udidahan.com/2009/06/14/domain-events-salvation/
Acceso Agosto 2010.

[17] Greg Young.
«*What is a Domain Event?*».
http://codebetter.com/blogs/gregyoung/archive/2010/04/11/what-is-a-domain-event.aspx
Acceso Agosto 2010.

[18] Paul Rainer.
«*Domain-Driven Design Immersion Part#2*».
http://www.virtual-genius.com/blog/post/Domain-Driven-Design-Immersion-e28093-Part-2.aspx
Acceso Agosto 2010.

[19] Ritesh Rao.
«*Domain Events Pattern in NCommon*».

http://www.codeinsanity.com/2009/06/domain-events-pattern-in-ncommon.html
Acceso Agosto 2010.

[20] Gojko Adzik.
«*QCon London 2009: Eric Evans – What I've learned about DDD since the book*».
http://gojko.net/2009/03/12/qcon-london-2009-eric-evans-what-ive-learned-about-ddd-since-the-book/
Acceso Agosto 2010.

[21] Domain-Driven Design Community.
«*Eric Evans: What I've learned about DDD since the book*».
http://domaindrivendesign.org/library/evans_2009_1
Acceso Agosto 2010.

[22] Jimmy Nilsson.
«*Applying Domain-Driven Design and Patterns With Exmaples in C# and .NET*».
Addison-Wesley, 2006.

[23] Kent Beck.
«*eXtremme Programming eXplained. Embrace Change*».
Addison-Wesley, 1999.

[24] Wikipedia.
«*Big Design Up Front*».
http://en.wikipedia.org/wiki/Big_Design_Up_Front
Acceso Agosto 2010.

[25] Portland Pattern Repository's Wiki.

«*Big Design Up Front*».
http://c2.com/xp/BigDesignUpFront.html
Acceso Agosto 2010.

[26] Tom McCracken.
«*BDUF Waterfall vs Lightweight Agile vs the Cowboy Way*».
http://www.leveltendesign.com/blog/tom/bduf-waterfall-vs-lightweight-agile-vs-cowboy-way
Acceso Agosto 2010.

[27] Justin Etheredge.
«*It is BDUF, not DUF*».
http://www.codethinked.com/post/2008/09/13/It-is-BDUF-not-DUF.aspx
Acceso Agosto 2010.

[28] Tom DeMarco.
«*Structured Analysis and System Specification*».
Yourdon Press Computing Series, 1979.

[29] Meiler Page-Jones.
«*The Practical Guide to Structured Systems Design*».
Yourdon Press Computing Series, 1988.

[30] Bertrand Meyer.
«*Object Oriented Software Construction, 2nd Edition*».
Prentice Hall, 1997.

[31] Ivar Jacobson and others.
«*Object-Oriented Software Engineering: A Case Driven Approach*».
Addison-Wesley, 1992.

[32] Martin Fowler and others.
«*Refactoring. Improving the Design of Existing Code*».
Addison-Wesley, 1999.

[33] Barbara Liskov.
«*Data Abstraction and Hierarchy*».
SIGPLAN Notices, 23(5), May 1988.

[34] Grady Booch.
«*Object Solutions: Managing the Object-Oriented Project*».
Addison-Wesley, 1996.

[35] Jeffrey Palermo.
«*The Onion Architecture 1, 2 and 3*».
http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
http://jeffreypalermo.com/blog/the-onion-architecture-part-2/
http://jeffreypalermo.com/blog/the-onion-architecture-part-3/
Acceso Agosto 2010.

[36] Alistair Cockburn.
«*Hexagonal Architecture*».
http://alistair.cockburn.us/Hexagonal+architecture
Acceso Agosto 2010.

[37] Aaron Jensen.
«*Machine.Specifications Documentation*».
http://github.com/machine/machine.specifications
Acceso Agosto 2010.

[38] Aaron Jensen.
«*Introducing Machine.Specifications (or MSpec for short)*».

http://codebetter.com/blogs/aaron.jensen/archive/2008/05/08/introducing-machine-specifications-or-mspec-for-short.aspx
Acceso Agosto 2010.

[39] Aaron Jensen and others.
«*Test + AutoMocking + IoC Container= ?*».
http://blog.eleutian.com/CommentView,guid,762249da-e25a-4503-8f20-c6d59b1a69bc.aspx
Acceso Agosto 2010.

[40] Jean-Paul Boodhoo.
«*Machine.Specifications.DevelopWithPassion Source Code Home Page*».
http://github.com/developwithpassion/machine.specifications
Acceso Agosto 2010.

[41] Jean-Paul Boodhoo.
«*Screencast – Getting Started With Machine.Specifications*».
http://blog.jpboodhoo.com/ScreencastNdashGettingStartedWithMachineSpecifications.aspx
Acceso Agosto 2010.

[42] James Broome.
«*Introducing Machine.Specifications.AutoMocking*».
http://jamesbroo.me/introducing-machinespecificationsautomocking/
Acceso Agosto 2010.

[43] Ayende Rahien.
«*Rhino.Mocks*».
http://www.ayende.com/projects/rhino-mocks.aspx
Acceso Agosto 2010.

[44] Ayende Rahien.
«*Rhino Mocks Documentation*».
http://www.ayende.com/wiki/Rhino+Mocks+Documentation.ashx
Acceso Agosto 2010.

[45] Anónimo.
«*Git-scm*».
http://git-scm.com/
Acceso Agosto 2010.

[46] Martin Fowler.
«*Version Control Tools*».
http://martinfowler.com/bliki/VersionControlTools.html
Acceso Agosto 2010.

[47] Scott Chacon.
«*Pro Git*».
Apress, 2009.

[48] Jeffrey Palermo, Ben Scheirman and Jimmy Bogard.
«*ASP .NET MVC In Action*».
Manning, 2010.

[49] Fabrice Maguerie, Steve Eichert and Jim Wooley.
«*LINQ In Action*».
Manning, 2008.

[50] Dan Pilone and Russ Miles.
«*Head First Software Development*».
O'Reilly, 2008.

[51] Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates.
«*Head First Design Patterns*».
O'Reilly, 2004.

[52] Gerard Meszaros.
«*xUnit Test Patterns. Refactoring Test Code*».
Addison-Wesley, 2007.

[53] Joshua Kerievsky.
«*Refactoring To Patterns*».
Addison-Wesley, 2004.

[54] Michael C. Feathers.
«*Working Effectively with Legacy Code*».
Prentice Hall, 2005.

[55] Kyle Baley and Donald Belcham.
«*Brownfiled Application Development in .NET*».
Manning, 2010.

10.1 Específica para el Estudio

[56] John Skeet.
«*C# In Depth 1st Edition*».
Manning, 2008.

[57] Juval Löwy.
«*Programming .NET Components 2nd Edition*».
O'Reilly, 2005.

[58] Greg Pearman and James Goodwill.
«*Pro .NET 2.0 eXtreme Programming*».
Apress, 2006.

[59] Andrew Troelsen.
«*Pro C# 2008 and the .NET 3.5 Platform*».
Apress, 2006.

[60] Jeffrey Richter.
«*CLR via C#*».
Microsoft Press, 2006.

[61] Steve McConnell
«*Code Complete 2*».
Microsoft Press, 2004.

[62] Dhanji R. Prasanna
«*Dependency Injection*».
Manning, 2009.

[63] John Skeet.
«*C# In Depth 2nd Edition*».
Manning, 2011?.[2]

2 La referencia bibliográfica [63] todavía no ha sido publicada. La única forma de acceder a ella es a través del EAP (Early Access Program) de Manning. Este programa, en caso de haber comprado el libro por adelantado, te permite acceder a la creación del mismo mientras el Autor lo va escribiendo. A mayores te proporciona un canal directo de comunicación con el Autor, en caso de que quieras sugerir algún cambio o aclaración. Por esa razón el año de publicación aparece como 2011?.