UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA INFORMÁTICA

Evaluación de la calidad de código usando TDD en el desarrollo de la lógica de negocio de un sistema de información para un club deportivo

Autor

Francisco José Gómez-Caldito Viseas

Tutor

José Juan Hernández Cabrera
(Departamento de Informática y Sistemas)

Las Palmas de Gran Canaria, a 26 de noviembre de 2013

A mis padres, que son la razón principal por la que, después de tantos años, me embarcara de nuevo en la Universidad

A José Juan, mi tutor, que me convenció de que superara mis inseguridades y aceptara un Trabajo de Fin de Grado que de verdad me iba a enseñar a ser un mejor profesional

A mi esposa Jacqueline, por aguantarme todos esos días que entraba en 'ataque de pánico'

A los tipos que crearon Stack Overflow, y a todos aquellos que ofrecen sus aportaciones, sin las cuales me habría encontrado más de una vez en un callejón sin salida.

Y por último, aunque no menos importante, a la gente de NDepend, por esa licencia "Academic Sponsor" que me ha permitido realizar el estudio usando su impresionante herramienta, y a Roxanne, que tan amablemente me ha atendido en todos mis correos.

Muchas gracias a todos.

INDICE

Prólogo

No es frecuente que tu trabajo te permita estar al tanto de los continuos cambios en este mundo de la informática. Te centras en tu especialidad (la mía es la de Administración de Sistemas de Información) y en poco tiempo estas completamente obsoleto en cualquier otra rama. Así que, tras años de aislamiento, cada vez que volvía a pasar por la Universidad, quedaba conmocionado por el aluvión de nuevos descubrimientos que se agolpaban por entrar en mi cerebro.

Ocurrió por primera vez cuando realicé el Proyecto de Fin de Carrera de la Diplomatura. Fue un trabajo de Análisis y Diseño. En aquella época la Diplomatura no incluía asignaturas de Ingeniería de Software, y para mi resultó una verdadera revelación. Por fin encajaban todas las piezas. Resultaba absolutamente natural la secuencia Análisis-Diseño-Implementación, y aprendí un método formal que me permitía aplicar sus pasos de manera simple y diáfana, la Metodología en Cascada. Cada una de sus etapas: primero adquieres los requisitos, luego diseñas todo y lo representas mediante una documentación completa, y, por último, se pasan estos documentos al equipo de desarrollo, que lo implementa. Cada uno de los roles: el ingeniero, al que equiparaba con un arquitecto que hace los planos, el jefe de equipo, que era el aparejador que coordina, y los programadores, que eran los obreros. Todo se antojaba razonable, indiscutible. Me convertí en un adalid de la Métrica 3.

Y, tras muchos años, vuelvo a pisar la Universidad para la Adaptación al Grado. Llega el momento del TFG… y recibo el mazazo que hace tambalear mis convicciones. Se revela ante mí la existencia de una nueva radicalmente distinta de afrontar ciertos proyectos de desarrollo de software. El mismo tutor que me iluminó hacía tantos años en el camino de los métodos formales me va desgranando, en una tarde de principios de verano, una idea simple, la del Desarrollo Dirigido por Test. Es una idea muy atractiva, la de desarrollar 'sobre seguro'. Encaja muy bien en ese último paso, el de implementación. Pero *"No"*, me dice, "*No se trata de eso. TODO el desarrollo, de principio a fin, se hace con base en TDD"*. Me quedo desorientado. ¿Sin una análisis y diseño íntegro previo? ¿Sin un arquitecto? ¿Sin un aparejador? ¿Sin planos detallados? Mi edificio construido a lo largo de años se derrumba. ¿Cómo puede algo nacer de ese caos? *"Porque se basa en otros conceptos, en otras ideas, en otras herramientas"*, replica. Le muestro mi escepticismo, mi reticencia, mi inseguridad ante la idea de afrontar algo tan radicalmente distinto estando yo tan anclado, tan oxidado como me encontraba. El me explica, me argumenta, me persuade.

Me anima. *"Pruébalo"*. Le hice caso y lo he probado. ¿Y saben que?… ¡Me gusta!

Creo que a ustedes podría pasarles lo mismo.

INTRODUCION

Quizá lo que más debería preocupar a un Ingeniero del Software, es la idea de la Calidad.

A veces, obsesionados por cumplir los plazos y facturar, dejamos este concepto un poco de lado, sin darnos cuenta, pues nos centramos tan solo en dos factores del mismo, los únicos que el cliente va a poder apreciar

- *Robustez:* El software no debe presentar fallos cuando se encuentre en producción
- *Conformidad:* Que cumpla con los requisitos que se han descrito contractualmente.

En las metodologías clásicas, de la que el Modelo en Cascada es quizá su mayor exponente, lo que normalmente nos certifica el departamento de Quality Asurance es que hemos seguido los procedimientos establecidos en la documentación del proyecto y que, por tanto, siguiendo la idea de la Calidad de Procesos, el producto final debería estar correcto. Unas pruebas finales, de parte de los compañeros de Quality Control, y 'sello al canto'.

Pero, últimamente, han surgido otro tipo de metodologías, llamadas *ágiles*, no tan centradas en la documentación, que proclaman conseguir un software de mayor 'calidad'. ¿Sin un soporte documental sobre el que basarnos para emitir veredicto?¿Cómo definir qué es el Quality Code entonces?¿Cómo medirlo?

Para los seguidores de las Metodologías Ágiles, la argumentación es simple. Su software resulta de mejor calidad por varias razones:

- *Robustez:* La manera de desarrollar en el 'agilismo' implica la continua comprobación del mismo. En las metodologías clásicas a veces el software resultante es difícilmente testable en su integridad, y los errores no surgen hasta que el uso continuado los hace patentes. Usando Test-Driven Developement, el software no es que sea fácilmente testable, es que está completamente testado, al 100%.

- *Conformidad:* Las metodologías ágiles no garantizan que el software cumpla con la documentación exhaustiva del contrato, sino que garantiza que cumple con lo que *el cliente realmente desea*. Es ciertos tipos de desarrollos es relativamente complejo que todos los requisitos de usuario emerjan desde el primer análisis. El cliente no tiene muy claro que necesita, o cómo el software puede ayudarle. Las metodologías ágiles facilitan al usuario final software operativo cada poco tiempo, que éste puede utilizar, y están especialmente diseñadas para aceptar sugerencias, cambios en los requisitos, y adatarse. Por ello se llaman 'agiles'. No quiere decir esto que no exista una fase inicial de Especificación de Requisitos, ni mucho menos, sino que saben adaptarse dinámicamente a la entrada de los nuevos.

Pero, además, existen otros factores que hacen que el software sea *intrínsecamente* de mayor calidad con el 'agilismo':

- *Legibilidad:* Un código no es más legible porque esté más documentado o comentado. Un código es más legible cuando, sencillamente, es más fácil de entender y seguir. Claridad, simplicidad, elegancia. Las metodologías ágiles, como veremos, por su propia naturaleza, necesitan producir código más legible.

- *Flexibilidad*: Nuevamente, la propia naturaleza del desarrollo ágil exige flexibilidad en el código. Y la flexibilidad emana de seguir de forma firme los principios del Diseño Orientado a Objetos, lo que garantiza la máxima *reusabilidad* y *portabilidad* del código. Código que es, *formalmente,* de mayor calidad.

En el presente Trabajo de Fin de Grado nos hemos planteado la cuestión de la validez de estas afirmaciones. Hemos realizado un proyecto siguiendo las ideas del agilismo, al menos en la medida de lo posible, pues, al ser un trabajo personal, no ha existido un equipo con el que comprobar el funcionamiento de, por ejemplo, el SCRUM. Pero sí nos hemos lanzado al desarrollo TDD, y hemos experimentado el BDD. Hemos probado nuevas herramientas de desarrollo, y hemos seguido, gracias a algunas métricas de código, la evolución y el resultado final.

Aquí presentamos el resultado de todos estos meses, en una memoria en la que podemos apreciar cuatro partes principales:

- Descripción inicial de los problemas de las metodologías clásicas

- Exposición de los conceptos del agilismo, y cómo dan solución a los problemas anteriores

- Presentación de las herramientas utilizadas durante el desarrollo.

- Resultados

Esperamos, con todo ello, poder responder a la cuestión que enunciamos en el título de nuestro Trabajo de Fin de Grado: *¿Cómo es la calidad del código resultante en un desarrollo TDD?*

1. OBJETIVOS

1.1. Objetivos del proyecto

El objetivo principal del actual Trabajo de Fin de Grado es realizar un desarrollo parcial de una aplicación, en particular algunas bibliotecas de clases relacionadas con facturación en la gestión de un club deportivo, siguiendo las propuestas del *Agilismo* y las técnicas TDD (Test Driven Developement), evaluando la calidad del código resultante desde los puntos de vista:

- *Cualitativo:* El desarrollo es eminentemente empírico. La mejor manera de evaluar un método es probarlo, y hacer una exposición en la memoria de todo el proceso, argumentando problemas hallados y las ventajas encontradas.

- *Cuantitativo:* Toda la evolución del proyecto estará monitorizada a través de métricas objetivas de código. Dichas métricas se expondrán y comentarán en los resultados.

Al mismo tiempo, como en cualquier proyecto de fin de carrera, podemos enfocarlo a través de dos objetivos específicos:

- **Formar:**

    - **Es un proceso de aprendizaje:** Todo proyecto de Fin de Carrera, como reflejan las competencias, está pensado para formarte como un mejor profesional. Desde el principio mi tutor me dejó claro que este era el reto esencial del proyecto. En el más simple de los casos, un proyecto es un entrenamiento, una gimnasia, un adiestramiento que te prepara para la vida laboral. La mayoría de las veces es además un proceso de descubrimiento, donde exploras nuevos campos guiado por tu tutor. Y, si hay suerte, será un proyecto de investigación, donde descubres cosas realmente nuevas que compartir con otros.

    - **Es un estudio didáctico:** Todo TFG queda finalmente publicado para que otros aprendan a través de tu trabajo. En este caso el proyecto es principalmente didáctico, con algunos toques de investigación a través de estudio con métricas. Es una guía tutorial sobre cómo empezar a aplicar el agilismo, donde los lectores podrán ver ejemplos prácticos de uso de estas metodologías y herramientas para que puedan afrontar sus propios proyectos.

- **Implementar:** A veces el resultado de un TFG es un producto final que puede ser utilizado a nivel de producción. En nuestro caso el desarrollo no fue más que un instrumento para nuestro estudio. Sin embargo, se han implementado algunos productos finales:

- Se ha desarrollado todo el procedimiento de envío de adeudos directos SEPA según ISO20022 XML. Esta normativa entra en vigor en Febrero de 2014, y todas las empresas que emitan recibos han de adaptarse a las mismas, por lo que esta implementación en C# puede resultarles tremendamente útil.

- Finalmente, y aunque no fuera parte inicial del proyecto, para el estudio de las métricas he desarrollado una aplicación completa que, haciendo uso de la API de el software NDepend, presenta cifras, estadísticas y gráficas de las mismas, extrayéndolas 'in situ' del código fuente y del histórico de análisis previos realizados.

2. COMPETENCIAS CUBIERTAS

Las competencias cubiertas en el presente proyecto han sido:

Competencias comunes

CII01: *Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente*

Este TFG es, precisamente, un estudio sobre evaluación de calidad en un desarrollo de software que nosotros mismos hemos implementado, centrándonos en elementos tales como la fiabilidad del código. Durante el desarrollo del mismo se ha tenido en cuenta la normativa vigente en cuestiones referentes a la privacidad, no habiéndose usado para la carga del sistema ningún dato personal real

CII02: *Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.*

Para la correcta realización de este Trabajo, el alumno debió realizar una correcta planificación, dirección y ejecución del proyecto, apoyado por su tutor. El proyecto es, además, una guía para ayudar a otros profesionales para planificar, concebir, desplegar y dirigir proyectos con 'metodologías ágiles'.

CII03: *Capacidad para comprender la importancia de la negociación, los hábitos de trabajo efectivos, el liderazgo y las habilidades de comunicación en todos los entornos de desarrollo de software.*

Uno de los principales principios del Agilismo es "valorar a los individuos y su interacción", por lo que la adquisición de esta competencia es vital en grupos de desarrollo que sigan esta práctica. Durante todo el TFG se ha hecho hincapié en este valor, y en la idea de 'Software Craftmanship'.

CII04: *Capacidad para elaborar el pliego de condiciones técnicas de una instalación informática que cumpla los estándares y normativas vigentes.*

Al no tratarse este TFG del desarrollo de un producto con destino a producción, sino de un estudio de metodologías, esta capacidad tan solo ha sido tangencialmente cubierta como resultado de los análisis comparativos.

CII16: *Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería de software.*

Al tratarse de un estudio sobre los resultados de la aplicación de una nueva metodología, hasta cierto punto emergente, podemos afirmar que esta ha sido la competencia principal cubierta por el proyecto

Competencias de la Ingeniería del software

IS01: *Capacidad para desarrollar, mantener y evaluar servicios y sistemas software que satisfagan todos los requisitos del usuario y se comporten de forma fiable y eficiente, sean asequibles de desarrollar y mantener y cumplan normas de calidad, aplicando las teorías, principios, métodos y prácticas de la ingeniería del software.*

El presente proyecto es un estudio de Ingeniería de Software donde se evalúa, mediante un desarrollo parcial, las capacidades de una metodología para crear software cuya característica principal es, precisamente, su perfecto ajuste a los requisitos de usuario y su fiabilidad, por lo que, junto a la competencia CII16, conforman el núcleo del aprendizaje.

Competencias del Trabajo de Fin de Grado

TFG01: *Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.*

Una vez presentado el presente Trabajo de Fin de Grado, el alumno realizará la debida presentación y defensa del mismo ante el tribunal, con lo que se cumplirá la primera parte de este requisito. La segunda parte viene garantizada por la aprobación en su momento por la comisión del documento TFG01, donde se exponía el contenido del TFG.

3. APORTACIONES

El presente proyecto es un estudio, y como tal, su objetivo es aportar conocimientos, tanto al propio estudiante como a la comunidad de profesionales de la ingeniería, sobre el Agilismo, sus principios, cómo aplicarlos, y los resultados que de su aplicación se obtienen.

Por tanto, presenta aportaciones al entorno socio-económico, técnico y científico a varios niveles:

- **Como proyecto de investigación:** Presentamos resultados tanto cualitativos (en forma de discusión argumentada) como cuantitativos (en forma de resultados medibles mediante métricas de código) sobre el uso del Agilismo.

- **Como proyecto formativo:** El proyecto supone una guía de aprendizaje en varios campos:

    - *Aprendizaje de metodologías ágiles:* Se exponen los conceptos en los que se basan, se muestra cómo construir un proyecto siguiendo dicho modelo, y se enseña como integrar y utilizar las herramientas necesarias en un grupo de desarrollo que desee empezar a programar siguiendo este paradigma.

    - *Aprendizaje de la API de NDepend:* NDepend es una potentísima herramienta de análisis de código, como ya veremos. Pero una de las características que le definen es que ofrece una magnífica API a través de la cual sus funciones se pueden integrar completamente en cualquier sistema de desarrollo. Como parte del proyecto se ha implementado una herramienta que facilita la adquisición de métricas y gráficas de resultados..

    - *Aprendizaje de la normativa SEPA para el envío de adeudos domiciliados:* En febrero de 2014 dejarán de estar en vigor las normativas del Consejo Superior Bancario que regían el envío de pagos domiciliados en España según el Cuaderno 19. A partir de esta fecha solo se podrá usar las especificaciones indicadas a nivel europeo por el SEPA, que incluyen una compleja normativa para adaptar el envío de recibos domiciliados al estándar ISO20022 XML. Para el presente proyecto se ha implementado una librería en C# que realiza dicha función, incluyendo esta memoria, en la parte de normativa, una descripción, resumen y referencia de enlaces a las diferentes fuentes.

Por otra parte, aunque no fue su objeto principal, el TFG aporta algunos productos terminados que pueden usarse como base para desarrollos propios:

- Una aplicación para monitorizar en tiempo real las métricas de código y hacer representaciones gráficas de las mismas usando la API de NDepend. Si bien lo

primero (las métricas) se podían seguir desde el NDepend, esta iba a ser una gran aportación, ya que el NDepend 4.5 no incluía funciones de representación de gráficas de evolución. La nueva versión 5 de NDepend, lanzada en octubre de esta año 2013, ha añadido una poderosa funcionalidad de representación de gráficas de todo tipo

- Una librería para la implementación de envío SEPA en C#. Por mucho que busqué en su momento, no encontré ninguna librería pública que realizara este trabajo. Espero con sinceridad que mi aportación sea de utilidad a la comunidad.

4. NORMATIVA Y LEGISLACION

Seguidamente referimos tanto la legislación general que pudiera afectar al desarrollo del proyecto y la propia memoria presentada, como normativa específica que se ha debido seguir como requerimiento del software implementado.

4.1. Legislación general

4.1.1. LOPD

Incluir la legislación vigente sobre proyectos informáticos que afecten al TFG (ley de protección de datos, leyes sobre seguridad,…)

La *Ley Orgánica 15/1999 de 13 de diciembre de Protección de Datos de Carácter Personal (LOPD)*[1], tiene por objeto garantizar y proteger, en lo que concierne al tratamiento de los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente de su honor e intimidad familiar.

El tratamiento de datos de carácter personal se rige por esta ley de acuerdo al artículo 2.1, que establece:

> *La presente Ley Orgánica será de aplicación a los datos de carácter personal registrados en soporte físico, que los haga susceptibles de tratamiento, y a toda modalidad de uso posterior de estos datos por los sectores público y privado.*
>
> Se regirá por la presente Ley Orgánica todo tratamiento de datos de carácter personal:
>
> a) Cuando el tratamiento sea efectuado en territorio español en el marco de las actividades de un establecimiento del responsable del tratamiento.
>
> b) Cuando el responsable del tratamiento no esté establecido en territorio español, pero le sea de aplicación la legislación española en cumplimiento de normas de Derecho Internacional público.
>
> c) Cuando el responsable del tratamiento no esté establecido en territorio de la Unión Europea y utilice en el tratamiento de datos medios situados en territorio español, salvo que tales medios se utilicen únicamente con fines de tránsito.

En el artículo 3 de la LOPD, se realiza la siguiente definición de "datos de carácter personal":

> *Datos de carácter personal: cualquier información concerniente a personas físicas identificadas o identificables.*

En el presente trabajo no vamos a implementar ningún software que deba asegurarse o garantizar el cumplimiento de estas medidas, pero tanto para los test como en los ficheros de prueba XML se necesita cargar el sistema con ejemplos. Dichos ejemplos han sido especialmente escogidos para no incluir datos reales de personas físicas. En algunos casos, incluye algunos datos públicos de personas jurídicas, que no están recogidas en esta ley.

Por tanto, podemos concluir que el presente desarrollo se ajusta a la LOPD.

4.1.2. Derecho de Cita

A lo largo de este estudio se han ido incorporando fragmentos de obras de otros autores con el objetivo de enriquecer sustancialmente nuestro conocimiento sobre la materia tratada.

El "**derecho de cita"** es un concepto legal que limita los derechos de un creador intelectual respecto al uso de parte de su obra para fines docentes o de investigación.

En España, este derecho está regulado en el artículo 32 del Texto Refundido de la Ley de Propiedad Intelectual (TRLPI) según el Decreto Legislativo 1/1996, de 12 de abril. Este artículo posibilita la inclusión, en una creación propia fragmentos de obras de otros autores, de naturaleza escrita, sonora o audiovisual, siempre que la obra reproducida ya haya sido divulgada y su inclusión se realice para su análisis, comentario o juicio crítico. Para poder acogerse a esta norma es necesario que se emplee con fines docentes o de investigación.

Desde el punto de vista de la normativa internacional, El " derecho de cita" con fines docentes está regulada por el artículo 10 del Convenio de Berna para la Protección de las Obras Literarias y Artísticas[2], *"a condición de que se hagan conforme a los usos honrados y en la medida justificada por el fin que se persiga."*

Así mismo, destacar que se reserva a las legislaciones nacionales la facultad de autorizarlo.

En el ámbito de la Unión Europea, la Directiva 2001/29/CE[3], relativa a la armonización de determinados aspectos de los derechos de autor y derechos afines a los derechos de autor en la sociedad de la información, reconoce el derecho de cita con fines docentes en su artículo 5.3:

"a) cuando el uso tenga únicamente por objeto la ilustración con ***fines educativos o de investigación científica****, siempre que, salvo en los casos en que resulte imposible, se indique la fuente, con inclusión del nombre del autor, y en la medida en que esté justificado por la finalidad no comercial perseguida"*

4.2. Normativas aplicadas al proyecto

4.2.1. Single Euro Payments Area (SEPA)

La creación de la Unión Económica y Monetaria y la introducción de los billetes y monedas en euros han sido hitos decisivos para la existencia de un mercado único en la Unión Europea. Desde su introducción, en enero de 2002, en todos los países de la eurozona es posible realizar pagos en efectivo en la misma moneda con la comodidad y sencillez con la que se efectuaban anteriormente los pagos en las respectivas monedas nacionales. No obstante, en el ámbito de los pagos que no se hacen en efectivo, permanece una situación de fragmentación que, en última instancia, dificulta la culminación de ese objetivo.

Para contribuir a paliar esta situación nace la **Zona Única de Pagos en Euros**[4], conocida bajo el acrónimo SEPA (de la terminología inglesa Single Euro Payments Area).

¿Qué es la SEPA?

La SEPA es la zona en la que ciudadanos, empresas y otros agentes económicos pueden hacer y recibir pagos en euros, con las mismas condiciones básicas, derechos y obligaciones, y ello con independencia de su ubicación y de que esos pagos impliquen o no procesos transfronterizos.

La SEPA supondrá un nuevo escenario caracterizado por una armonización en la forma de hacer pagos en euros principalmente mediante el empleo de tres grandes tipos de instrumentos: las transferencias, los adeudos domiciliados y las tarjetas de pago.

Calendario de entrada en vigor del SEPA

La Comisión Europea[5] estableció los fundamentos legales del nuevo marco a través de la Directiva 2007/64/CE[6] sobre servicios de pago en el mercado interior.

Esta directiva ha sido traspuesta a la legislación española en 2009, mediante la Ley de Servicios de Pago[7].

Aunque la migración hacia los nuevos estándares SEPA ya ha comenzado, y actualmente coexisten con los esquemas nacionales, la reciente adopción del Reglamento CE 260/2012, establece el **1 de febrero de 2014** como fecha límite para que las transferencias y adeudos nacionales sean reemplazados por los nuevos instrumentos SEPA.

Donde encontrar información sobre el SEPA

Además de la página principal del European Payments Council (EPC)[8], podemos encontrar información exhaustiva en el Banco Central Europeo[9] . Al mismo tiempo, el Banco de España ha creado una página[10] donde reúne toda la información existente en español, orientada tanto a empresas que necesiten realizar sus transacciones según este estándar, como para las entidades de crédito.

Por otra parte casi todos loa bancos y cajas ofrecen a sus clientes, a través de sus servicios on-line, acceso a información relativa al SEPA.

Instrumentos del SEPA

A través del se han definido los instrumentos SEPA para transferencias y adeudos directos, junto con un conjunto de normas y estándares que deben ser cumplidos por los proveedores de servicios de pago.

Se detallan a continuación cada uno de estos instrumentos de pago:

- **Transferencias:** La transferencia SEPA es un instrumento de pago básico para efectuar abonos en euros, sin límite de importe, entre cuentas bancarias de clientes en el ámbito de la SEPA, de forma totalmente electrónica y automatizada.

- **Tarjetas:** La Zona Única de Pagos en Euros (SEPA) establece un marco general en el que los titulares de tarjetas pueden hacer pagos y retirar efectivo en euros dentro de la SEPA, con la misma facilidad y comodidad que en sus países de origen.

- **Adeudos Directos básicos:** El adeudo directo es un servicio de pago destinado a efectuar un cargo en la cuenta del deudor. La operación de pago es iniciada por el acreedor, sobre la base del consentimiento dado por el deudor al acreedor, y transmitida por éste a su proveedor de servicios de pago.

- **Adeudos Directos B2B:** En el adeudo directo B2B, el deudor y el acreedor tendrán que ser obligatoriamente empresas o autónomos (no consumidores) que han acordado utilizar el servicio de adeudos directos B2B para los pagos/cobros relativos a sus transacciones comerciales.

ISO20022

Dentro de la normativa del SEPA uno de los elementos más importantes era definir exactamente como se transmitirían los mensajes entre los intervinientes en el proceso:

- Entidades bancarias (Bank) del acreedor y del deudor

- Emisor (Creditor)

- Deudor (Debtor)

Para ello, el EPC decidió adoptar para la SEPA los estándares ISO20022[11]. Estos fueron creados por el ISO/TC68[12] y describen una plataforma común para el desarrollo de mensajes usando:

- una metodología de modelado para capturar, con una sintaxis independiente del área de negocio financiera, el flujo de transacciones y sus mensajes asociados
- un diccionario centralizado de los elementos usados en las comunicaciones financieras
- un conjunto reglas para convertir los modelos de mensajes en esquemas XML o ASN1, dependiendo de si se prefiere usar una sintaxis ISO 20022 XML o basada en ASN.1.

Por tanto, para cada una de los servicios SEPA (transferencias, adeudos…), el EPC ha publicado una serie de normas y pautas para implementar los mensajes que han de intercambiarse los actores siguiendo el correspondiente estándar ISO20022.

4.2.1.1. Los Adeudos Directos Básicos SEPA

Como práctica de BDD/TDD vamos a implementar uno de los servicios SEPA: el envío de recibos domiciliados.

Tanto el esquema básico como el B2B utilizan el estándar ISO20022 a la hora de transmitir sus mensajes. El EPC especifica, a través de sus documentos, como implementar, bajo ISO20022, los detalles de de los mismos (Figura 4-1)

En particular nos centraremos para nuestro desarrollo en los Adeudos Directos mediante el Esquema Básico, pues es el que utilizan normalmente las empresas para remesar sus recibos domiciliados a sus clientes, ya que el B2B está diseñado para intercambio entre empresas o autónomos, no para particulares.

Las dos sitios principales donde obtener la regulación sobre los mismos son la página web que a ellos dedica el EPC[13] (en ingles) y la página web del Banco de España[14]

El documento principal sobre de los adeudos directos es el *Sepa Core Direct Debit Scheme Rulebook (EPC016-06)*[15] , donde se recoge toda la información referente a:

- Alcance de la normativa
- Roles de los actores (emisores, deudores, entidades de crédito…)
- Reglamento y operativas de negocio (recogida de autorizaciones, formas y plazos de entrega de los mensajes…)
- Tratamiento del esquema para el SEPA

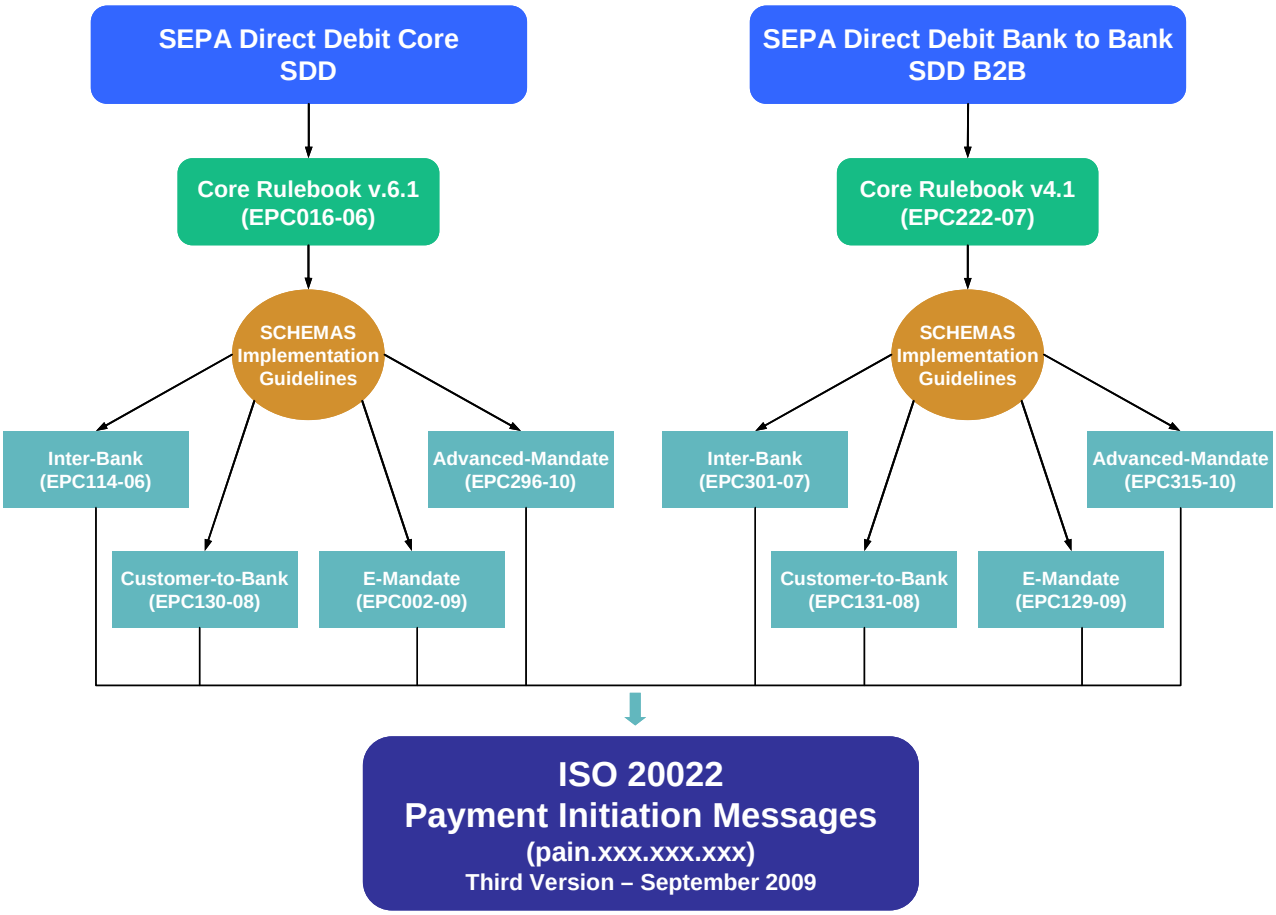


Figura 4-1: Esquema de los documentos que definen los estándares para los envíos SEPA

Intercambio de mensajes de adeudos directos en SEPA

En nuestro caso, de todo lo anterior, nos focalizaremos en cómo implementar los mensajes intercambiados entre los actores. En particular, los que el emisor intercambia con su banco (ordenar los adeudos, solicitar retrocesiones, recibir las devoluciones). Veamos un esquema de los mismos (Figura 4-2):

Los mensajes que un emisor de adeudos intercambia con su banco son:

- Del emisor al banco:
    - Presentación de los adeudos directos
    - Retrocesiones para reintegrar el importe al deudor en caso de adeudo indebido
- Del banco al emisor
    - Rechazos de operaciones/mensajes enviados
    - Devoluciones

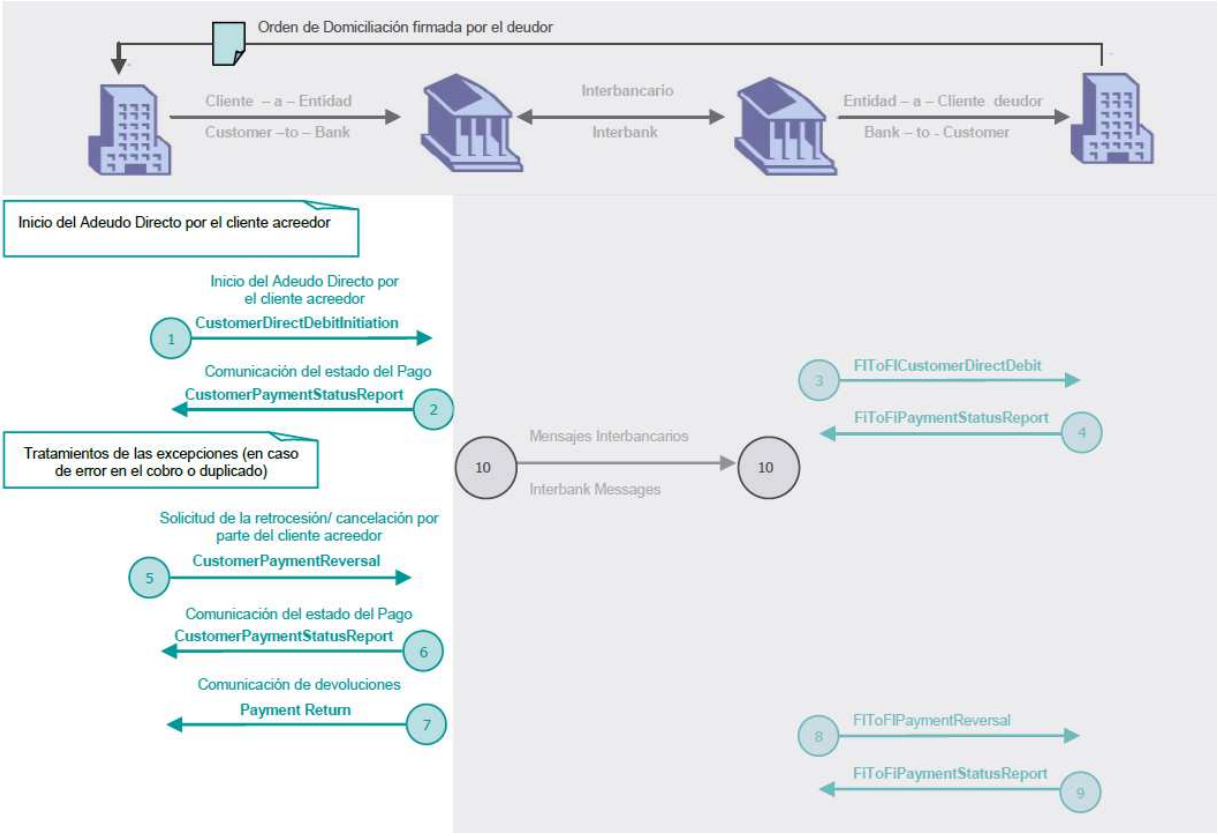


Figura 4-2: Intercambio de mensajes de adeudos directos SEPA

Estos mensajes se definen a través de los diferentes esquemas SEPA, y su correspondiente formato de mensajes ISO 20022XML

Esquema Adeudo Directo SEPA	Estándares ISO 20022 XML
DS-03 Presentación de los Adeudos Directos	CustomerDirectDebitInitiationV02 **(pain.008.001.02**)
DS-05 Rechazos de operaciones/mensajes presentados	CustomerPaymentStatusReportV03 (**pain.002.001.03**)
DS-05 Devoluciones	CustomerPaymentStatusReportV03 (**pain.002.001.03**)
DS-07 Retrocesión para reintegrar el importe al deudor en caso de adeudo indebido	CustomerPaymentReversalV02 (**pain.007.001.02**)
Implementation Guidelines	
SEPA EPC Customer-To-Bank Implementation Guidelines (EPC130-08)	

Tabla 4-1: Relaciones entre los esquemas SEPA y los estándares ISO20022

Como indica la Tabla 4-1, por ejemplo, para realizar una presentación de adeudos directos he de seguir las normas establecidas por el SEPA en su esquema **DS-03** (que se describe en el anteriormente citado documento EPC016-06), y he de construir el mensaje según la norma ISO 20022 XML **CustomerDirectDebitInitiationV02 (pain.008.001.002)**,

siguiendo las especificaciones particulares para implementación de envíos SEPA Cliente-Banco que aparecen el documento EPC130-08.

Es importante recalcar que no todos los elementos de los mensajes 'pain' son usados por el SEPA. Siempre manteniendo la integridad del esquema XML, las diferentes especificaciones DS del SEPA imponen restricciones y reglas de uso sobre elementos y atributos. Es por ello que, además de estudiar cómo se construye un mensaje 'Payment Initiation', se debe también tener en cuenta las guías de implementación SEPA publicadas por el EPC.

Las especificaciones completas para la creación de los mensajes XML, son, por tanto, bastante complejas. Es por ello que, a continuación, incluyo una referencia resumida de toda la documentación consultada.

Referencias

- **ISO 20022 Message Definition Report - Payments Maintenance 2009 - Edition September 2009[16]:** Es el documento principal, que incluye la definición de todos los esquemas XML Payment Initiation (pain), así como otros tipos de mensajes relacionados con pagos (Bank To Customer Cash Management -camt-, Payments Clearing and Settlement -pacs-).

- **ISO 20022 Customer-to-Bank Message Usage Guide - Customer Credit Transfer Initiation, Customer Direct Debit Initiation and Payment Status Report - Version 3.0**[17]: Un documento de ayuda para la construcción de los mensajes XML muy completo en cuya redacción han participado por una serie de corporaciones y particulares, y que esta almacenado por la sociedad SWIFT[18]

- **Sepa Core Direct Debit Scheme Customer-to-Bank Implementation Guidelines (EPC130-08)**[19] : Guía publicada por el propio EPC para la implementación de los mensajes Customer-To-Bank. En ella especifica como transportar los esquemas DS-03, DS-05 y DS-07 a los mensajes ISO 20022 XML, tal y como antes comentamos.

- El ISO20022 guarda un archivo[20] de todas las definiciones de mensajes. En particular, en su sección 'Third version of the Payment Initiation messages'[21], además del documento '*ISO 20022 Message Definition Report',* incluye:

    - **Fichero XML Schema Definition (.xsd)** para cada uno de los mensajes *pain,* que resulta ABSOLUTAMENTE IMPRESCINDIBLE a la hora de realizar un desarrollo TDD, pues permite ir validando cada uno de los elementos y atributos XML a medida que se van desarrollando.

    - **Ficheros de mensajes XML de ejemplo**

- **Ficha de Adeudos SEPA[22]:** Pequeña ficha que recoge las características básicas de la implementación española del Sepa, así como las pautas para la migración desde el anterior formato de texto plano CSB-Cuaderno 19[23]

- **Migración a SEPA de los Adeudos Domiciliados españoles**[24]: Por último, existe reglamentación española, recogido en un documento publicado por la AEB[25], CECA[26] y UNAC[27], donde aparecen las especificaciones sobre cómo realizar la migración del anterior sistema de envío de adeudos, el Cuaderno CSB-19 en texto plano a los nuevos adeudos SEPA. En el se indican las correspondencias entre las figuras de ambos esquemas.

- **Órdenes en Formato ISO 20022 para la emisión de Adeudos Directos SEPA – Esquema Básico – Guía de Implantación – Noviembre 2012**[28]**:** La Confederación Española de Cajas de Ahorros ha confeccionado un documento que reúne en un solo folleto en español la práctica totalidad de las especificaciones anteriores. De especial interés, en esta última versión de Noviembre 2012, es que recoge una variación del esquema básico 'core' (COR), llamado COR1, que apareció por primera vez en el Core Rulebook 6 (EPC016-06), el cual permite un ciclo de presentación más corto.

Dado que los mensajes SEPA son una implementación de un subesquema de 'pain', hay que vigilar ciertos factores:

- Un mensaje que es conforme a SEPA es siempre conforme a 'pain'.

- Un mensaje que es conforme a 'pain' puede no ser conforme a SEPA, ya que puede usar elementos o valores no reconocidos en el subesquema SEPA.

- Esto implica que el uso del fichero XML Schema Fefinition (.xsd) que facilita ISO20022 ha de usarse con suma cautela, o adaptarlo previamente a los esquemas DS, especialmente en las restricciones de valores para elementos y atributos.

Por último, existe reglamentación española, donde aparecen las especificaciones para la migración del envío de adeudos cuaderno 19 clásico en texto plano (Cuaderno CSB-19)

5. ESTADO ACTUAL

5.1. Los problemas de las metodologías clásicas

5.1.1. De los requisitos al producto final

¿Cuántos de nosotros, desarrolladores y analistas, no hemos visto mil y una veces esta viñeta?

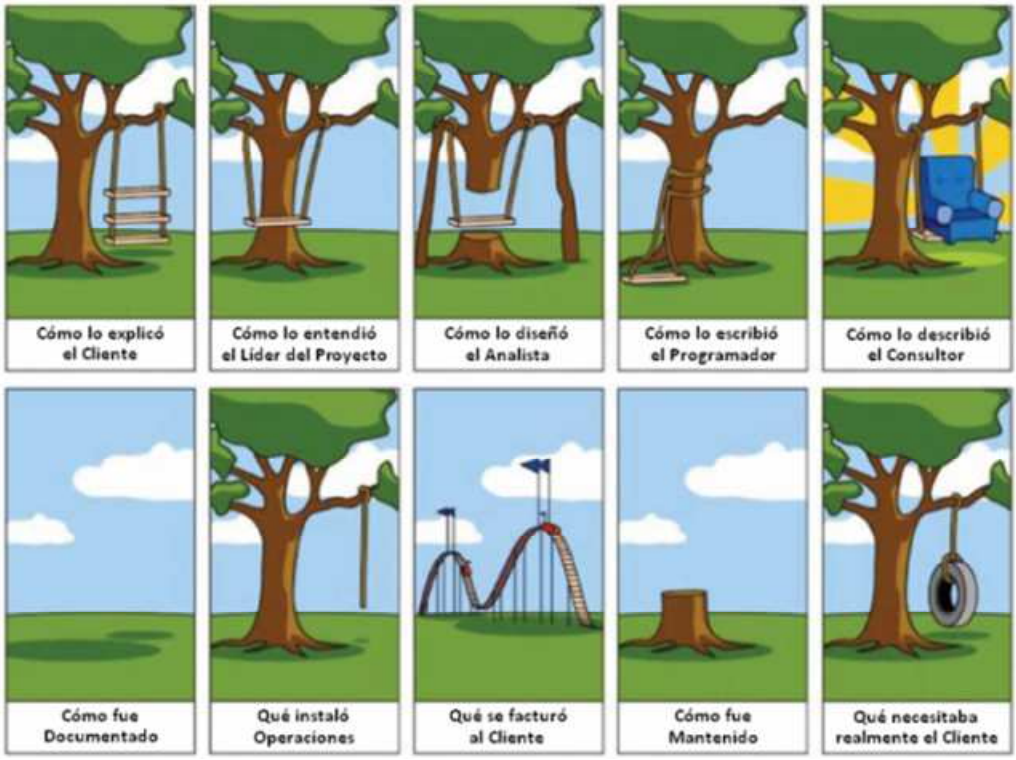


Figura 5-1: El problema de la interpretación de los requisitos

Uno de los mayores problemas en el desarrollo de software es la enorme dificultad que supone el llegar a entender la mente del cliente, lo que realmente necesita, y terminar entregándole exactamente lo que demanda. Todos conocemos algún caso de una aplicación 'que no se adecua' a lo que el usuario había pedido. Tras meses y meses de espera, al recibir su producto, el cliente nos mira con desaprobación y, de manera más o menos educada, más o menos indirecta, nos suelta alguna variación de 'esto no es lo que había pedido'

¿Cómo podemos evitar estas situaciones?

Desde hace décadas se ha abordado el problema desde la Ingeniería de Software, que se centra en el ciclo de Análisis-Diseño-Implementación, con dos aproximaciones distintas:

- En las metodologías 'en cascada' el software se desarrolla en un silo ciclo, en el que se planifica con cautela y se establecen de manera firme los pasos que se dan desde el inicio hasta el final

- Métodos Iterativos que permiten un feedback con el cliente, realizándose tantos ciclos como sea necesario.

La segunda aproximación resulta menos sensible al problema de la conformidad con los requisitos, por lo que, a este nivel, el código debería ser mejor. Sin embargo, ambos se basan en un elemento base, la aproximación de ingeniería análisis-diseño-implementación, donde, entre otros elementos:

- No se desarrolla software sin una rígida fase de diseño previa

- En proyectos de cierto nivel hay una estructura claramente jerarquizada de participantes, separando totalmente las tareas realizan los analistas, los diseñadores y los programadores finales que implementan el producto.

Por tanto existe una fuerte dependencia del desarrollo a la correcta elaboración e interpretación de la documentación, a lo largo de un largo viaje, que, en la metodología en cascada, puede llevar meses, e incluso años, antes de poder comprobar que 'todo ha ido bien'. Podemos decir que las metodologías clásicas son sensibles al problema del *'teléfono roto'*.

5.1.2. El teléfono roto

Como perfectamente ilustra la viñeta de la siguiente página, creada por el genial artista del humor gráfico que es Quino, todo mensaje que es intercambiado y repetido a través de una cadena de interlocutores es proclive a registrar una enorme distorsión.

Algo tan simple, y que muchos de nosotros conoceremos a través del clásico juego del 'teléfono roto'[29], supone uno de los mayores problemas que registra el enfoque clásico de la Ingeniería de Software. A la hora de transmitir los requerimientos desde el cliente hasta el programador final de la aplicación, el mensaje puede llegar a encontrar por su camino a un comercial, a un analista, a un diseñador y a un jefe de equipo de desarrolladores. La ingeniería de Software ha ideado multitud de herramientas y protocolos para intentar que dicha información no se pervierta, pero a veces, esto solo añade un elemento más al problema, que es la traducción e interpretación de dichos documentos.

Es simplemente un hecho físico registrable en cualquier ámbito, muy bien estudiado en la Teoría de la Información[30]: al intentar pasar información de un dominio a otro SIEMPRE se produce una cierta pérdida de información debido a los canales y medios de transmisión, a los transductores, a los sistemas de almacenamiento intermedios y finales… Por muy bien que se intenten definir los protocolos, por muy exhaustivamente que se intente documentar, siempre nos podemos encontrar con una toma de especificaciones incorrecta debido a haber sabido escuchar al cliente, unos casos de uso incompletas al traducir estas especificaciones, un análisis erróneos, unos esquemas UML

mal diseñados…. Cuantas más herramientas y diagramas usemos, cuanto más completos intentemos ser, más riesgo de distorsión de la información se corre. Hagan una prueba: en su próxima reunión social con amigos échense una partida a un 'party game' llamado "Eat Poop, You Cat"[31], un juego clásico conocido con otros nombres no tan malsonantes como "Paper Telephone", en el que tan solo hay dos tipos de transducciones: de lenguaje formal a un dibujo, y viceversa. Tras unas pocas iteraciones el resultado es sorprendentemente cómico ☺

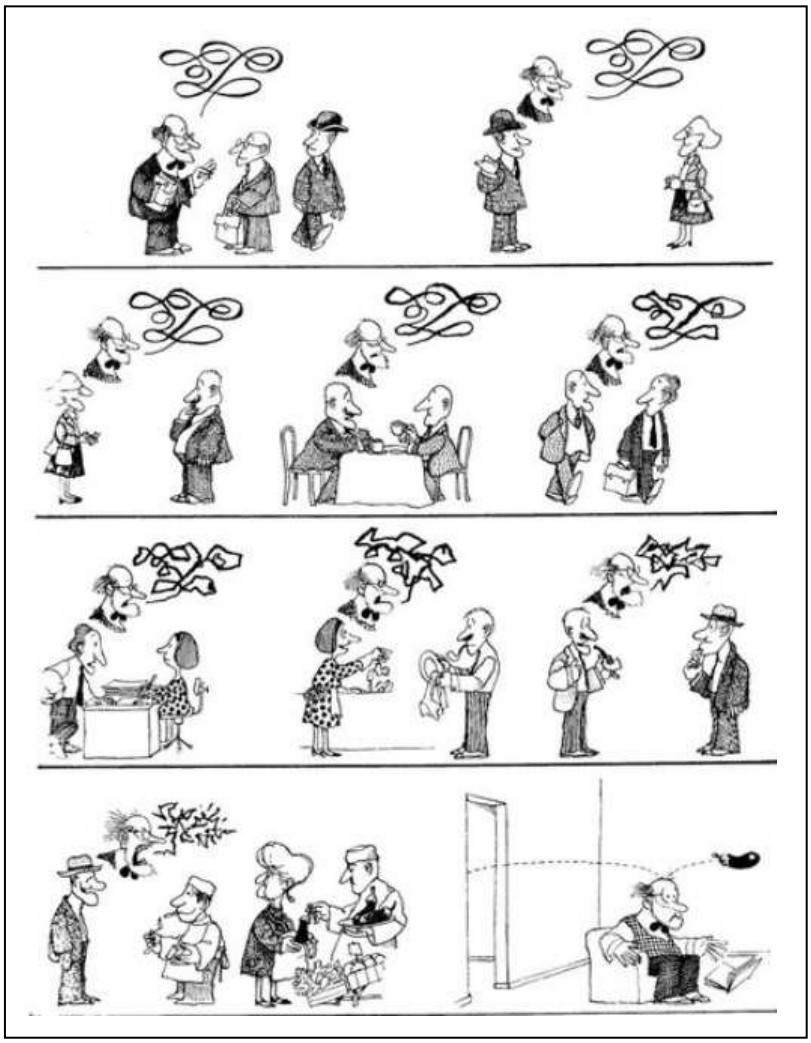


Figura 5-2: Cada interlocutor intermedio en una cadena de mensajes aumenta la probabilidad de distorsión
Fuente: Quino

5.1.3. El modelo en cascada

El Modelo en Cascada[32] es, de todos los modelos, el más simple, y, aunque existen muchas otras metodologías más flexibles, es el bloque sobre el que se construyen la mayoría de ellas, por lo que nos servirá para ilustrar los problemas de que adolecen la mayoría de los enfoques clásicos

Se basa en que el desarrollo de un software ha de seguir una secuencia definida de fases. Cada fase consta de una serie de etapas y metas claramente definidas, y una serie de herramientas y procedimientos a seguir. Si se detecta una falla en una de las fases, tan solo es necesario volver un paso atrás, a la fase anterior, y corregir.

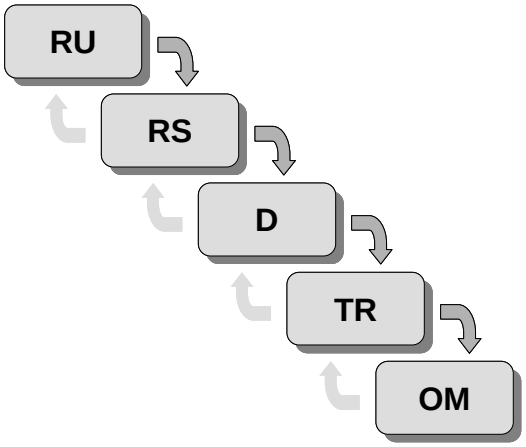


Figura 5-3: Esquema del Modelo Lineal o en Cascada

Las fases de la Metodología en Cascada son:

- ***Adquisición de Requisitos de Usuario***: Se adquieren a través de los clientes los requisitos del sistema, y se deciden cuales de ellos deberán ser cubiertos por el software

- ***Especificación de Requisitos de Software***: Se describe el comportamiento esperado en el software una vez desarrollado. El ingeniero ha de comprender el ámbito de la información del software así como la función, el rendimiento y las interfaces requeridas.

- ***Diseño***: Definimos completamente la estructura de los datos, la arquitectura del software, la lógica de programa y el interfaz.

- ***Implementación***: Es en este punto, y no antes, cuando empezamos a desarrollar código siguiendo las especificaciones del diseño. El desarrollador no necesita tener conocimientos sobre requisitos. Si el diseño está bien realizado, la traducción a código será casi mecánica.

- **Prueba**: Tras finalizar completamente la implementación, empieza la fase de prueba, en la que se generan entradas y se deberían registrar salidas de acuerdo a los requisitos.

- **Mantenimiento**: El software puede sufrir cambios, debido a errores detectados que hay que cubrir, o a mejoras que se solicitan.

Como vemos, en la **Figura 5-4**, el modelo es bastante sencillo, y parece ser robusto. Siguiendo los preceptos de la Calidad en los Procesos, confiamos que, si cumplimos escrupulosamente con los procesos definidos en cada una de las etapas, el resultado final se ajuste a los requisitos iniciales.

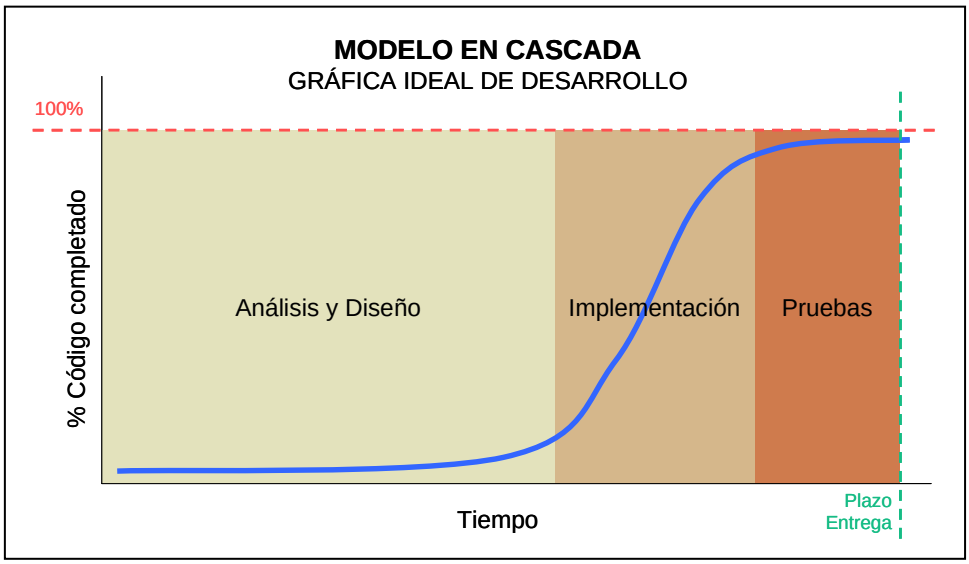


Figura 5-4: Gráfica ideal de desarrollo del modelo en cascada

Sin embargo, este modelo adolece, quizá, de demasiados inconvenientes en la mayoría de las situaciones, precisamente por su rigidez:

- Es muy difícil establecer completamente los requisitos desde un principio, por lo que, para evitar tener que volver atrás en fases posteriores, la fase de análisis puede alargarse produciendo estancamiento en el desarrollo.

- Rara vez se cumple exactamente con la secuencia y aparecen ciclos inesperados, como pueden ser que se presentes nuevos requisitos, o detectar que, por errores en el diseño, estos no han sido contemplados, siendo éstos más catastróficos cuanto más adelante en desarrollo se localicen. Y, precisamente, en este modelo, la fase de pruebas está al final…

- El cliente puede terminar perdiendo la paciencia, especialmente si se superan los plazos. Solo va a ver su producto al final del desarrollo. Es más, si quisiera echar un vistazo de 'por donde va su programa' en un momento dado, se puede encontrar con que, pasado más de la mitad del plazo de entrega, no se ha desarrollado una sola línea, pues aún se está en la fase de análisis y diseño.

El resultado final viene pareciéndose más a la **Figura 5-5**

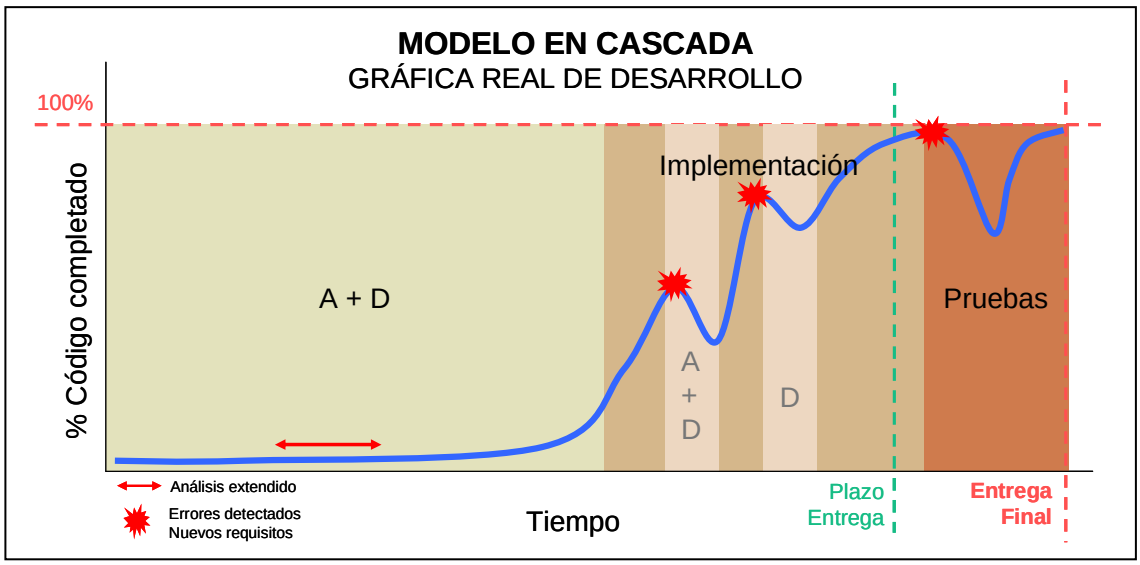


Figura 5-5: Gráfica real de desarrollo en el modelo en cascada

5.1.4. Los modelos evolutivos

La excesiva rigidez del modelo en cascada en la actualidad solo lo hacen en cierto modo válido para situaciones muy particulares en los que se disponga de tiempo más que suficiente para efectuar el desarrollo, sin la presión del cliente. Proyectos muy complejos y extensos se pueden abarcar siguiendo por ejemplo las especificaciones de la Métrica 3[33]

Como alternativa, se han planteado modelos evolutivos que permiten ir presentando versiones parciales del software al cliente, tales como le modelo incremental o el modelo espiral.

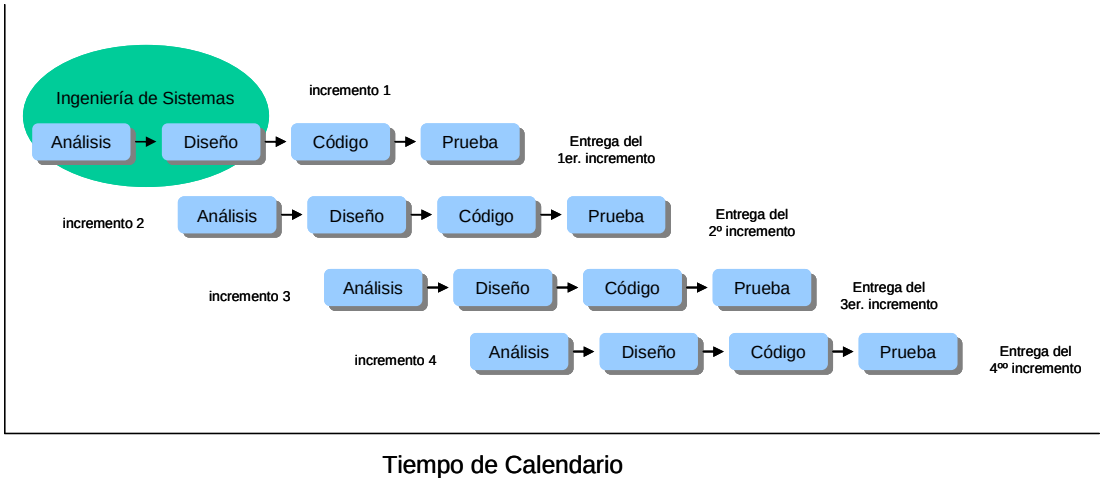


Figura 5-6: El Modelo Incremental

El modelo incremental (**Figura 5-6**) tiene como filosofía aplicar secuencias lineales de forma escalonada mientras progresa en el tiempo del calendario. Cada secuencia produce un 'incremento' en el software, que se entrega al cliente, que recibe software útil y mejorado cada cierto tiempo. Es importante ver que los desarrollos se solapan en el

tiempo, por lo que *la adquisición de requisitos se realiza al principio*, y se espera que no sea necesario que el cliente vaya a presentar nuevas especificaciones, aunque, si llegaran a producirse, podrían integrarse en alguno de los incrementos subsiguientes.

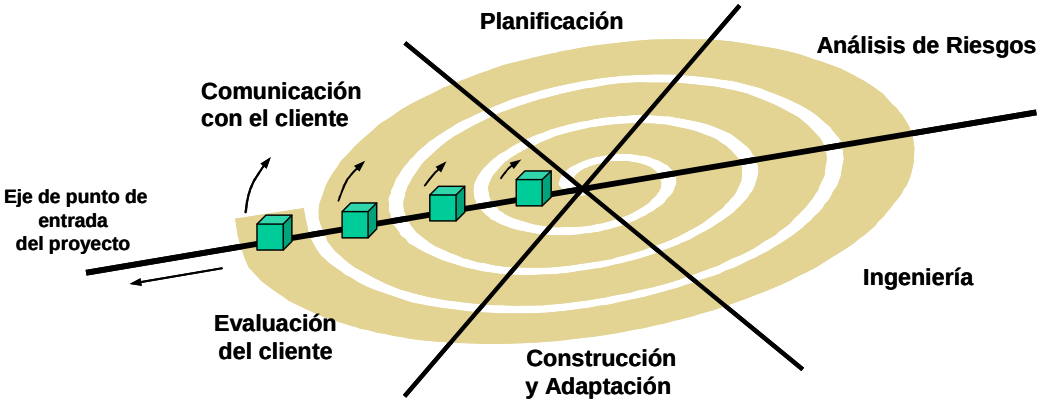


Figura 5-7: Modelo espiral

El modelo espiral (**Figura 5-7**)es también iterativo por naturaleza, pero en el mismo los desarrollos no se solapan, sino que cada vuelta de la espiral comienza escuchando al cliente y termina entregándole un producto, que el cliente evalúa y para darnos su opinión y quizá proponer mejoras o cambios. En las primeras iteraciones se pueden presentar modelos en papel o prototipos para adquirir requisitos del cliente, mientras que en las últimas se van entregando versiones más completas del sistema diseñado.

Los métodos incrementales que hemos visto supusieron un avance en la manera de gestionar los proyectos de software, y solucionan, en cierto modo, el problema de tener satisfecho al cliente:

- Nuestro cliente no tiene que esperar hasta el final para ver su software. Puede verlo, tocarlo, usarlo. Observa con satisfacción como el producto que ha pedido va tomando forma.

- Al mismo tiempo, puede ir proponiendo mejoras, o advertirnos de los errores que hayamos podido cometer al adquirir y plasmar los requisitos.

Pero este último punto, los requisitos cambiantes, seguía causando problemas, por una razón: cada uno de los ciclos seguía basándose en un ladrillo fundamentalmente rígido, el modelo lineal secuencial Analisis-Diseño-Código-Prueba (**Figura 5-8**)

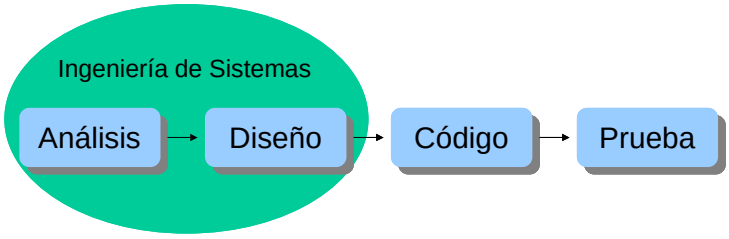


Figura 5-8: El modelo lineal secuencia de la Ingeniería

En el modelo secuencial una pequeña propuesta del cliente puede suponer que debamos rehacer por completo todo un diseño y tener que volver a implementar una enorme cantidad de código. Además, el código resultante de usar este modelo suele ser costoso de modificar, ya que está concebido para cumplir de manera ajustada a un diseño específico.

5.2. La 'Alternativa Ágil'

Tenemos entonces metodologías iterativas o evolutivas que nos permiten interactuar con el cliente, pero la aproximación a través de la ingeniería clásica hace que implementar los nuevos requisitos que pueda aportar resulte casi prohibitivo. ¿Qué hacer?

Aquí donde viene a ayudarnos el 'agilismo'. Una de sus premisas centrales es la flexibilidad. Como dice Carlos Blé en su libro: *"Los requisitos cambiantes son bienvenidos, incluso en las etapas finales del desarrollo"(Blé, Carlos et al, 2010)[4]*

El agilismo es el 'ladrillo perfecto' para los modelos evolutivos o iterativos. El SCRUM[34], por ejemplo, es una metodología iterativa, que, en vez de usar el modelo secuencial de la ingeniería de software, ha hecho uso del agilismo.

6. El Agislismo

El 17 de Febrero de 2001, un grupo de programadores y analistas, muy críticos con los rígidos modelos de desarrollo imperantes hasta el momento, se reunieron para tratar sobre técnicas y procesos para desarrollar software. En la reunión se acuñó el término "Métodos Ágiles" para definir a los métodos que estaban surgiendo como alternativa a las metodologías formales (como, por ejemplo, la anteriormente comentada Métrica3[33]) a las que consideraban excesivamente "pesadas" y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo. Entre ellos estaban Ken Beck, el escritor del libro *Extreme Programming Explained(Beck, K., 1999)[3]* y que, mas tarde, junto con David Astels impulsarían el surgimiento del TDD *(Beck, K., 2003)[4] (Astels, D., 2003)[1]* , o Jeff McKenna, según la mayoría , reconocido como creador de la metodología SCRUM[34].

Juntos, resumieron los principios sobre los que se basan los métodos alternativos en cuatro postulados, lo que ha quedado denominado como el **Manifiesto Ágil[35]**.

6.1. El Manifiesto Ágil

El Manifiesto Ágil plantea que hay que cambiar la escala de valores de ciertos elementos del desarrollo del software.

MANIFIESTO ÁGIL

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas
Software funcionando sobre documentación extensiva
Colaboración con el cliente sobre negociación contractual
Respuesta ante el cambio sobre seguir un plan

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Figura 6-1: El Manifiesto Ágil

- **Valorar más a los individuos y su interacción que a los procesos y herramientas:**

  Este es posiblemente el principio más importante del manifiesto, pues reivindica la labor final de programador y ha dado pie al mas reciente movimiento del *'Software Craftmanship[36]'(artesanía de software)*, que ve al desarrollador más como un 'maestro artesano' que a un simple obrero de una cadena industrial de montaje. Por supuesto que los procesos ayudan al trabajo. Son una guía de

operación. Las herramientas mejoran la eficiencia, pero sin personas con conocimiento técnico y actitud adecuada, no producen resultados

- **Valorar más el software que funciona que la documentación exhaustiva:**

  Poder ver anticipadamente cómo se comportan las funcionalidades esperadas sobre prototipos o sobre las partes ya elaboradas del sistema final ofrece una retroalimentación muy estimulante y enriquecedor que genera ideas imposibles de concebir en un primer momento; difícilmente se podrá conseguir un documento que contenga requisitos detallados antes de comenzar el proyecto. Los documentos son soporte de la documentación, permiten la transferencia del conocimiento, registran información histórica, y en muchas cuestiones legales o normativas son obligatorios, pero se resalta que son menos importantes que los productos que funcionan. Menos trascendentales para aportar valor al producto.

- **Valorar más la colaboración con el cliente que la negociación contractual:**

  Las prácticas ágiles están especialmente indicadas para productos difíciles de definir con detalle en el principio, o que si se definieran así tendrían al final menos valor que si se van enriqueciendo con retro-información continua durante el desarrollo. También para los casos en los que los requisitos van a ser muy inestables por la velocidad del entorno de negocio.

  En estos casos el valor del resultado no es consecuencia de haber controlado una ejecución conforme a procesos, sino de haber sido implementado directamente sobre el producto. Un contrato no aporta valor al producto, sino la interacción con el cliente.

- **Valorar más la respuesta al cambio que el seguimiento de un plan:**

  Para un modelo de desarrollo que surge de entornos inestables, que tienen como factor inherente el cambio y la evolución rápida y continua, resulta mucho más valiosa la capacidad de respuesta que la de seguimiento y aseguramiento de planes pre-establecidos. Los principales valores de la gestión ágil son la anticipación y la adaptación; diferentes a los de la gestión de proyectos ortodoxa: planificación y control para evitar desviaciones sobre el plan.

De todo ello dimanan los 12 principios del agilismo[37], que no reproduciremos aquí para no extendernos excesivamente, pero de los que destacaremos estas ideas:

- Diluir las fuertes fronteras jerárquicas entre analistas-diseñadores-programadores, de manera que todos los integrantes del proyecto participan en cada una de las fases.

- Continua colaboración con el cliente: se adquieren unos pocos requisitos, se desarrolla la parte del software que las cumple, se le muestra, y se toman nuevos requisitos, ya sea modificaciones a lo existente o ampliaciones de la funcionalidad.

A primera vista esta forma de enfocar la solución de un proyecto choca frontalmente con la de la ingeniería, ya que:

- Se trabaja con adquisiciones parciales de requisitos, que, además pueden ser cambiantes.
- No existe un diseño arquitectónico o de datos previo a la implementación, ya que sería absurdo, dados los requisitos cambiantes.
- Sin unas especificaciones completas de diseño, la persona que implementa trabaja sobre los requisitos.

En un primer vistazo el agilismo se nos antoja, a las personas que, como yo, hemos realizado desarrollos 'en cascada' (mi Proyecto de Fin de Carrera para la Diplomatura fue, precisamente, un trabajo de Análisis y Diseño usando Yourdon*(Yourdon, E. (1982)[15])*, algo absolutamente caótico. Seguidor convencido hasta ahora de ladrillos como la 'Métrica3', siempre había promulgado que lanzarse a programar sin una completa planificación previa, era un absurdo. ¿Cómo puede funcionar bien algo así?

Este es, precisamente, el argumento principal de sus detractores, que las metodologías ágiles adolecen de control y planificación, mientras que los 'agilistas' exponen que, si a lo largo del proyecto aparecen desviaciones sobre el plan, éstas no se deben controlar para eliminarlas, sino adaptarse a ellas, o el producto final no será lo que en verdad necesita el cliente. Según estos últimos, una planificación inicial excesivamente rígida lo único que consigue reducir la calidad final del producto.

Adaptarse al cambio
(Agilismo)
.Vs.
Planificación y Control para evitar desviaciones del plan
(Metodologías clásicas)

Figura 6-2: Agilismo .vs. Metodologías Clásicas. Adaptación .vs. Planificación y Control

6.2. El Agilismo y sus herramientas

¿Cómo puede el agilismo sobreponerse al caos?¿Por qué no sucumbe un proyecto 'ágil' ante las incertidumbres de la falta de planificación exhaustiva, ante los requisitos inesperados? Porque hace uso de herramientas que le permiten 'adaptarse al cambio' con éxito:

- Metodologías como SCRUM para coordinar el proyecto
- Herramientas como BDD para interactuar con el cliente a la hora de adquirir los requisitos y hacer seguimiento de su cumplimiento.

- El TDD como metodología para el desarrollo del software, que se basa en estos pilares para tener éxito:
    - Sofware Craftmanship
    - Continua Refactorización
    - Clean Code
    - Code Quality

Intentemos mostrar una visión conjunta de cómo todas estos elementos actúan en perfecta sinergia, y luego entraremos en profundidad sobre cada una de ellas, viendo como se han usado en el desarrollo de este TFG.

Partimos de una idea base: usar métodos iterativos-evolutivos. El cliente ha de recibir nuevas versiones mejoradas de su software cada cierto tiempo, probarlas, y plantear las mejoras pertinentes, si son necesarias. Un marco de trabajo o *framework* diseñado específicamente para esta 'respuesta rápida al cambio' es el SCRUM, en la que en cada ciclo, de 3 o 4 semanas, se presenta al cliente una nueva versión mejorada del producto. Cada ciclo empieza seleccionando la serie de requisitos que se van a implementar, y planificando su distribución entre el equipo, y finaliza comprobando que dichos requisitos funcionan. Cada día hay una sesión de 'sincronización' en la que se revisa el estado de cada equipo de trabajo y se actúa en consecuencia.

¿Cómo se gestionan los requisitos? ¿Cómo los adquirimos, se los pasamos al equipo de desarrollo, y comprobamos que han sido añadidos a la nueva versión del software? Aquí viene en nuestra ayuda otra nueva herramienta, el BDD, o Behavior Driven Developement. El BDD[38] fue una evolución del ATDD[39] (Aceptance Test Driven Deveolpement) que permitía servir como nexo entre las peticiones de los clientes y los ciclos TDD. Con cada ciclo TDD, con cada prueba, comprobamos si un elemento del sotware (un módulo en un test unitario o la interacción entre varios sistemas en un test de integración) responde a conforme a lo esperado. Pero a veces resulta difícil ver cosas tales como ¿Dónde empiezo a trabajar con el TDD para cumplir con el requisito de mi cliente? ¿Qué debo testear y que no? ¿Cuándo parar, cuándo mis test han cubierto completamente lo que mi cliente me pide? El BDD, como luego veremos, permite obtener los requisitos del cliente en un lenguaje natural, que él mismo entiende, y, al mismo tiempo, especificarlos de manera que sean fáciles de seguir e implementar a través de TDD.

Ya tenemos las herramientas que permiten adquirir los requisitos, hacer un seguimiento, y comprobar que se están cumpliendo. Ahora necesitamos la herramienta que nos permita ir agregando, paso a paso, por acreción, dichos requerimientos al software. Y dicha herramienta es el TDD.

El TDD nace de una necesidad. En sistemas de trabajo como el Extreme Programming o SCRUM hablamos de la constante modificación en el software, que crece a medida se le añaden las funcionalidades que los requisitos de usuario exigen. Todos los programadores

sabemos lo peligroso que es ir retocando líneas de código, en como 'un cambio aquí' termina 'haciendo que algo falle allá'.

Para mantener el control sobre un código fuente en permanente cambio, se hizo casi imprescindible el uso de una herramienta por entonces no demasiado popular, los 'test units', pequeños fragmentos de código escribíamos para comprobar que las funciones ya desarrolladas seguían funcionando correctamente tras cada inevitable refactorización. Por fin teníamos nuestro código bajo control. ¿Seguro?

En realidad era muy difícil garantizar que los test cubrieran todos y cada una de las diferentes funciones del software. Usar como guía los 'casos de uso' resultaba absolutamente insuficiente. Había que, a posteriori, analizar cada método, cada clase, e imaginar como podría responder ante cualquier entrada. Siempre quedaba la duda de si había algo que no habías tenido en cuenta. A veces quedaban funciones sin testear, intencionadamente o no.

Finalmente, todo ello desembocó en una idea: en vez de desarrollar y luego crear los test que garantizaran que lo ya desarrollado seguiría funcionando tras los futuros añadidos, ¿por qué no hacerlo al revés?:

- La idea es que cada requisito de software se definiera en base a ejemplos (casos) que el código debía cumplir. En cada incremento implementamos el código que satisface un ejemplo, en ciclos de tres pasos:
    - Una vez elegido el requisito (caso, aceptación del cliente, ejemplo), se plantea el test que comprueba que el software implementa el requisito. Obviamente, el test FALLA.
    - LUEGO se añade al código ya existente las líneas que hagan que dicho requisito, solamente dicho requisito, se cumpla. Tenemos 'luz verde' en el test.
    - Es el momento de refactorizar el código, tanto como sea necesario, hasta que:
        - Estemos satisfechos con la calidad del código resultante
        - TODOS los test den luz verde.

El software va creciendo, por tanto, en base a pequeños microincrementos. Un caso cada vez, una aceptación cada vez, un test cada vez. Pasamos a 'Desarrollo Dirigido por Tests", Había nacido el TDD.

Esta nueva manera de entender el desarrollo del software, necesita, sin embargo, para funcionar estar bien apoyada en sólidos principios:

- Realizar una correcta, completa y exhaustiva refactorización*(Fowler, Martin., 1999)[7]* a cada paso. No se debe dudar en tener que desechar código

implementado. En cada iteración, nuestro código va a mejorar, y podemos confiar en que nuestros test nos avisarán cuando toquemos algo que no debamos.

- En este entorno en el que debemos reestructurar el diseño de las clases continuamente, la 'Calidad del Código' (Code Quality) es imperativa. Mantener los principios SOLID[40] en las clases es a la vez una necesidad y un resultado final de la continua refactorización en TDD. Para ello nos podemos ayudar de herramientas que, durante el desarrollo, nos permitan comprobar la calidad de nuestro código a través de ciertas métricas[41].

- El código desarrollado, al tener que ser continuamente retocado, a veces por distintos equipos de desarrollo, exige CLARIDAD. No hablamos de que este comentado y documentado, sino que siga los preceptos del *'Clean Code'(Martin, Robert C. 2008)[11]*.

- Todos estos principios requieren de un equipo de desarrolladores motivados y cualificados que garanticen la productividad del procedimiento. Se impone absolutamente la visión de "Software Craftmanship" que comentamos como fundamental en el primero de los valores del agilismo.

Con todo ello, el Agilismo proclama su ventaja frente a los métodos clásicos en la mayoría de los proyectos. Existe una gran controversia sobre las ventajas de uno sobre otro

- La rigidez en la planificación de los proyectos en cascada normalmente lo hacen:
    - Mas predecible en plazos de entrega y costes, pues se definen claramente desde el principio, lo cual es precisamente lo que desean ciertos tipos de grandes clientes, como los organismos públicos.
    - También es, en cierto modo, menos sensible a los cambios en los integrantes del equipo, por la misma razón.
    - Por el contrario, introducir un cambio inesperado en el diseño puede ser catastrófico, lo que supone desde enormes costes y cambios en plazos de entrega, hasta abandonos del proyecto. Estos cambios surgen con frecuencia como resultado de errores detectados en las pruebas, que en esta metodología se realizan al final, cuando las modificaciones son más complejas y costosas.

- Los proyectos ágiles son exactamente lo contrario:
    - Su flexibilidad los hace menos predecibles en plazos de entrega y costes, pero más adaptables a cambios. Por tanto, si no son proyectos de gran envergadura (es decir, para la mayoría de los proyectos) resulta una alternativa más válida.
    - Al no haber documentación formal exhaustiva, son más sensibles a los cambios en los integrantes del grupo.

- La intensa colaboración del cliente puede llegar a ser conflictiva en algunas ocasiones, por lo que se necesita una firma implicación del mismo.

Es por ello que, según algunas fuentes fuentes, la tasa de éxitos resulta mucho mayor en los proyectos ágiles, como muestra la Figura 6-3:

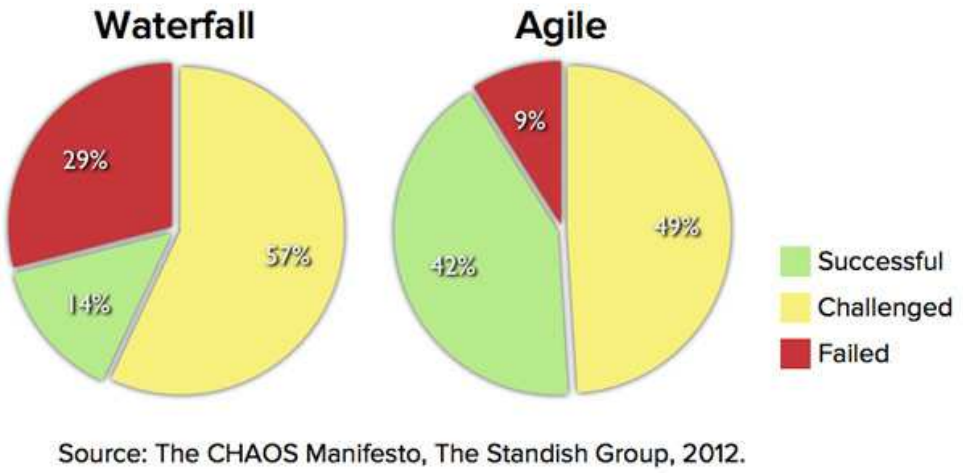


Figura 6-3: Comparativa de éxito entre métodos Ágiles y Modelo en Cascada

Ahora bien: ¿Qué pasa con la calidad del código? Según algunos estudios, como el realizado por Shine Technologies[42] o Scott Ambler[43] parece ser que también supera a las metodologías más clásicas. El integrar en sus equipos a programadores altamente cualificados o 'senior programmers' (primer valor del manifiesto ágil) y utilizar la continua refactorización prestando mucha atención al Clean Code y a las clases SOLID, produce mejores resultados que las implementaciones hechas por programadores normalmente menos cualificados de las arquitectura de software que presentan los ingenieros en su documentación, aún con la existencia de un sistema de SQM (Software Quality Management)[44].

Pasemos ahora a ver en mayor profundidad cuáles son estas herramientas que parecen garantizar su éxito.

6.2.1. SCRUM

El SCRUM es un framework de trabajo, esbozado con ese mismo nombre por primera vez en 1986 por Hirota Takeuchi[45] e Ikujiro Nonaka[46] en su artículo *"The New Project Developement Game" (Takeuchi, H. – Nonaka, I 1986)[14]*, aunque no fue hasta el año 1993 cuando se hizo un primer desarrollo siguiendo esta metodología (Jeff Sutherland, John Scumniotales y Jeff McKenna). En 1995 Ken Schwaber[47] normalizó su metodología. Es uno de los métodos que sirvió de inspiración en el Manifiesto Ágil, en cuya creación participó Jeff McKenna.

No vamos a extendernos mucho en su definición, ya al tratarse de un proyecto parcial con un equipo de una sola persona, no se ha utilizado. Pero es importante explicar su funcionamiento para entender cómo se integran el resto de herramientas del agilismo.

El Proceso

El SCRUM es un método iterativo en que cada ciclo o iteración es de corta duración (un mes natural o hasta dos semanas). En cada iteración se obtiene un resultado completo, un incremento sobre el producto final que puede entregarse al cliente.

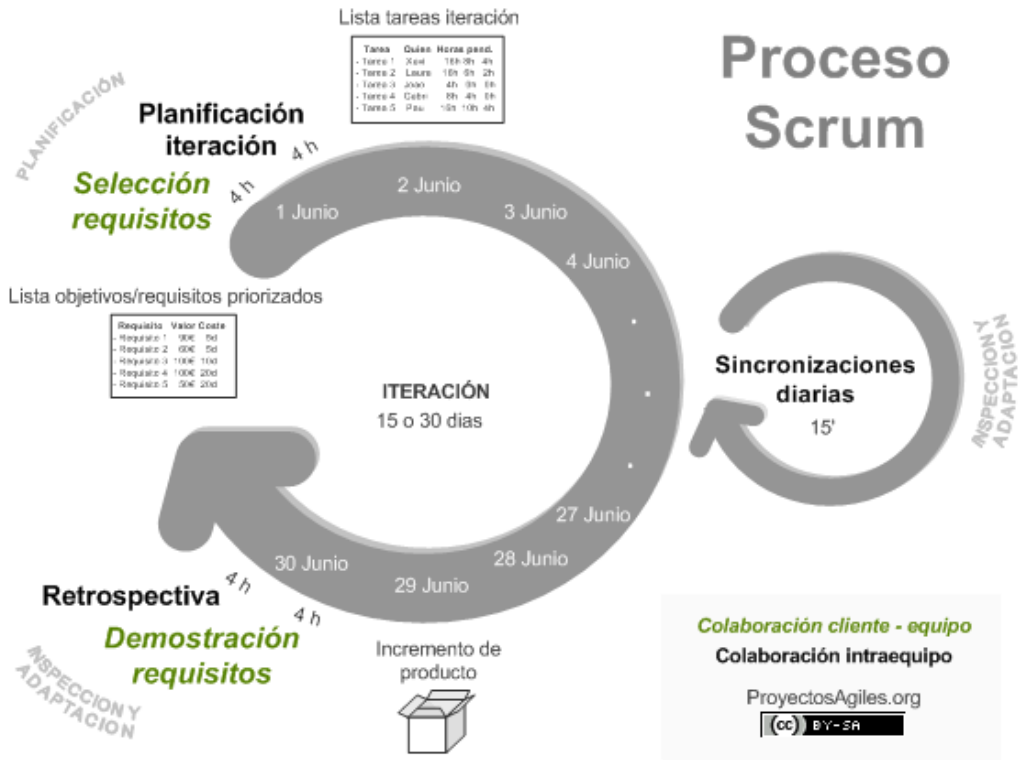


Figura 6-4: El proceso SCRUM
Fuente: ProyectosAgiles.org

Como observamos en la Figura 6-4, en el SCRUM se parte de una lista priorizada de objetivos, que actúa como plan del proyecto. En esta lista, el cliente prioriza los mismos balanceando el valor que le aportan respecto a su coste, y quedan repartidos en las correspondientes iteraciones y entregas. De esta manera, el cliente puede maximizar la utilidad de lo que se desarrolla y el retorno de inversión. Al mismo, tiempo, el sistema queda abierto a adquisición de nuevos requisitos.

Las actividades del SCRUM son:

- **Planificación de la iteración:** El primer día de cada iteración, se realiza la planificación, en dos partes:
    - Selección de requisitos (4 horas máximo): Entre el cliente y el equipo se revisa la lista priorizada de requisitos, resolviendo las dudas que pudieran surgir por ambas partes. Se elige la siguiente lista de requisitos a cumplir
    - Planificación de la iteración (4 horas máximo): El equipo elabora la lista de tareas de la iteración necesarias para cumplir los requisitos. Los miembros del equipo estiman conjuntamente el esfuerzo y se autoasignan las tareas.

- **Ejecución de la iteración:** En cada iteración, también llamados 'sprint', cada miembro del equipo realiza las tareas asignadas en la planificación, teniendo en cuenta que dispone de dos elementos importantes de ayuda:

    - Reunión de sincronización: Cada día de produce una reunión de sincronización de 15 minutos en la que cada miembro inspecciona el trabajo del resto del equipo (dependencias de tareas, progreso y obstáculos) para hacer las adaptaciones necesarias. Este punto es crítico, principalmente por las dependencias. En cada reunión cada miembro del equipo plantea

        - ¿Qué he hecho desde la última reunión?
        - ¿Qué tengo que hacer?
        - ¿Cuáles son mis impedimentos?
        - ¿Existen dependencias con las tareas de otros equipos?

    - Existe la figura del 'facilitador', líder del equipo, que se encarga de resolver las dudas y cualquier problema que pudiera reducir la productividad, así como velar por el cumplimiento de los procesos del SCRUM: coordina las reuniones con el cliente y guía las reuniones con el mismo e intraequipos, prepara las listas de requisitos, ayuda al equipo en su autogestión lo separa de las interrupciones externas, siendo el nexo con cualquier necesidad exterior.

- **Inspección y Adaptación:** El último día de la iteración hay una reunión de revisión, en dos partes:

    - Demostración(4 horas máximo): El equipo presenta al cliente los requisitos completados, en forma de producto completado. El cliente plantea las adaptaciones que considere, replanteándose el proyecto si es necesario.

    - Retrospectiva (4 horas máximo): El equipo analiza cómo ha sido su manera de trabajar, y los problemas que le han impedido progresar adecuadamente, si los hubiera. El facilitador se encargará de ir eliminando los obstáculos identificados

Los fundamentos

Como hemos visto, el proceso del SCRUM es bastante simple, y se fundamente en estas premisas:

- Desarrollo incremental en bloques temporales cortos y fijos
- Priorizar los requisitos por valor del cliente y coste del desarrollo

- Control empírico del proyecto, no basado en documentación. El equipo se sincroniza diariamente y al final de cada iteración se escucha al cliente
- Se potencia a los individuos del equipo, que se comprometen a hacer unas entregas y se les otorga la autoridad para organizar su trabajo
- Para evitar problemas con el punto anterior, se sistematiza la colaboración y la comunicación
- Hay una fuerte 'timeboxing' en las actividades: se fija el tiempo máximo para realizar una tarea, conseguir unos objetivos o tomar las decisiones.

Los requisitos

Como se puede apreciar, el funcionamiento del SCRUM es radicalmente diferente a otras metodologías, y para que funcione correctamente, se deben reunir ciertos requisitos:

- Cultura de empresa basada en equipo, delegación, creatividad y mejora continua.
- Compromiso del cliente en la dirección de los resultados del proyecto
- Compromiso de la dirección de la organización para resolver los problemas endémicos, realizar cambios organizativos si es necesario, formar equipos autogestionados y multidisciplinares y fomentar la cultura de la empresa citada al principio.
- Compromiso conjunto y de colaboración de todo el equipo
- Fuerte relación de colaboración y transparencia entre proveedor y cliente
- Facilidad para realizar cambios en el proyecto
- Tamaño del equipo entre 5 y 9 personas
- Equipo trabajando en un espacio común para facilitar la comunicación
- Dedicación del equipo a tiempo completo
- Estabilidad en los miembros del equipo.

6.2.2. Behaviod-Driven Developement (BDD)

Hemos visto en el SCRUM que cada iteración parte de una "lista de requisitos priorizada".

Existen muchas maneras de establecer dicha lista de requisitos. En la ingeniería de software se definen una gran cantidad de métodos formales. Por ejemplo:

- En Análisis Estructurado partimos de un entorno y de una lista de eventos del sistema, cada uno de los cuales se convertirá en una burbuja que la procesará.

- En el diseño orientado a objetos tenemos actores y casos de uso.

Sin embargo, para las metodologías ágiles, como el SCRUM, si bien pudieran ser usadas, presentan el inconveniente que expusimos al principio de este documento: el 'teléfono roto'. Existen tres transducciones básicas entre en cliente y el desarrollador final :

- El cliente expone sus requisitos en lenguaje formal, que un analista deberá traducir en alguno de los esquemas anteriormente mencionados, en la confianza de sean claros, concisos y completos.

- El programador interpretar dichos diagramas, extraer el requisito, e intentar dilucidar cómo atacar su integración en el desarrollo a base de iteraciones TDD.

- Finalmente, en la reunión de demostración, deberemos convencer al cliente de que dicha 'burbuja' o dicho 'caso de uso' se ha integrado en su software, cuando el pobre usuario no tiene por qué tener los conocimientos técnicos que le permitan interpretarlo.

Si finalmente, en la demostración, el software no cumple con lo que el cliente quería, no sabremos si es porque se tomaron mal los requisitos, si porque el analista no hizo bien su traducción a diagramas, si es que el desarrollador no hizo bien su traducción de diagramas a código, o si, simplemente, el desarrollador no cumplió con sus objetivos.

Las metodologías ágiles, como el SCRUM, donde los desarrolladores participan en las reuniones con el cliente, que disponen de reuniones de sincronización y la labor del facilitador, son mucho menos sensibles a estos 'errores de traducción', pero, aún así, funcionan mejor con un sistema de recogida y registro de requisitos más adecuado a su idiosincrasia.

Y es aquí donde viene en su ayuda el BDD, que nació como una evolución del TDD y el ATDD.

TDD y ATDD

En el TDD, como luego veremos, vamos desarrollando el software paso a paso. A base de tests comprobamos si cada caso posible, cada ejemplo que entre en nuestro sistema, es tratado adecuadamente y se recibe el resultado correcto. Con esta metodología, poco a poco, paso a paso, nos aseguramos que cada funcionalidad, cada requisito que añadimos está correctamente integrado en el sistema. Pero, como reza un viejo quiasmo "Son todos los que están, pero… ¿están todos los que son?"

El TDD nos ayuda a caminar con paso firme, pero no nos guía en cuál debe ser nuestro próximo paso.

El problema radica en que, a través de una sola herramienta, los tests, estamos intentando solucionar dos problemas diferentes:

- Comprobar si estamos cumpliendo con un requisito
- Comprobar si nuestro software funciona de la manera adecuada

En realidad, lo que ocurre es que debemos ser conscientes de que nuestra herramienta funciona A DOS NIVELES. Hay dos tipos y usos del TDD:

- **ATDD:** En ATDD (Aceptance Test-Driven Design) para cada requisito escribimos un test de aceptación (aceptante test), en el que especificamos un comportamiento (o requisito) que nuestro software debe cumplir. Luego escribimos el código necesario para que dicho test pase.
- **TDD de Desarrollo:** Mediante un test de desarrollo que comprobamos es si funcionan correctamente los elementos de nuestro diseño, es decir, si, dada una entrada en el sistema, se produce la salida esperada. Funciona de la misma manera que el anterior: primero creamos el test y luego añadimos el código que hace que dicho test pase.

El ATDD es, por tanto, quien garantiza que se cumplen todos los requisitos. El TDD de desarrollo, o, simplemente, TDD, es el que se encarga de que los requisitos que se integren en el sistema lo hagan correctamente.

Dan North y BDD

El tándem ATDD/TDD es el que nos hace, por tanto, progresar en el desarrollo del software. En el desarrollo ATDD partimos de aceptaciones basadas en requisitos, y cada aceptación la expresamos en un test ATDD, y desarrollamos toda la lógica necesaria para cumplirla con TDD.

Sin embargo, sigue existiendo un vacío:

- ¿Cómo obtenemos los requisitos que sirven para nuestra lista de aceptaciones ATDD?
- ¿Cómo aseguramos que está completa?

Podríamos, como antes dijimos, utilizar listas de eventos o casos de uso, pero quedó claro que resultan demasiado crípticas para el cliente. Al mismo tiempo, y si no se tiene demasiada experiencia en TDD, es frecuente encontrarse en callejones sin salida:

- ¿Qué testear y qué no testear en cada aceptación? ¿Por dónde empiezo el los TDD de desarrollo? ¿Cuándo acabo?
- ¿Cuál debe ser mi siguiente paso?

Era necesario dar un cierto enfoque formal, sin incurrir nuevamente en complejas documentaciones.

En el año 2004, Dan North[48] y otros colaboradores, tras varios años de desarrollar ATDD/TDD, propusieron una solución a este problema. Ideó una manera de estructurar el proceso de ATDD, los test de 'alto nivel', centrándose en los 'comportamientos' esperados del software, no en como debe ser implementada la lógica del mismo. Lo llamó Behaviour-Driven Developement o BDD[49].

En el interesante artículo *'Introducing BDD' (North, D. 2004)[13]*, Dan explica como fue madurando la idea del BDD, basada en los conceptos de 'Historia de usuario'[50] y en cómo traducirlas de manera formal para que fueran fácilmente interpretables tanto por un desarrollador TDD como por el cliente final.

Lo que necesitamos, por tanto, es tomar cada requisito y expresarlo con una fórmula que:

- Sea lo suficientemente simple para que el cliente pueda entenderla y seguir su cumplimiento.

- Sea lo suficientemente estructurada como para indicar al desarrollador los pasos que debe ir dando para, a través de TDD, integrar dicho requisito en el software.

La idea es brillantemente simple: tomar cada requerimiento del usuario y expresarla formalmente mediante una 'plantilla', tal como la de la Figura 6-5:

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
  And [some more context]...
When  [event]
Then  [outcome]
  And [another outcome]...

Scenario 2: ...
```

Figura 6-5: Plantilla para una 'historia de usuario' en BDD
Fuente: dannorth.net

Como podemos apreciar, se compone de tres partes:

- Un título, donde se expresa de manera absolutamente narrativa, en lenguaje natural, cuál es el requisito que se desea implementar. Normalmente, expresada directamente por el cliente

- Una narrativa, donde se especifica quien necesita la función (rol), que es lo que se necesita (feature) y cuál es el beneficio esperado (Benedit). En este punto, normalmente trabajando conjuntamente el cliente y el equipo de desarrollo, se define de de manera formal el requisito que queremos satisfacer.

- Finalmente, los criterios de aceptación, expresados en forma de escenarios. En cada uno de ellos se especifica el contexto [context], el evento que activa nuestro sistema [event] y los resultados esperados [outcome]. Todo ello con una fórmula fácil de recordar y repetir: '*Given-When-Then*' En esta parte es donde el desarrollador TDD interviene en mayor medida, pues, prácticamente detalla cómo ha de dar forma a su test de aceptación: la carga de precondiciones, el elemento de código que ha de invocar, y de qué elemento ha de comprobar el resultado.

Como vemos, esta simple plantilla nos resuelve todos los problemas que planteamos al inicio:

- ¿Cómo obtenemos los requisitos? Tomamos una historia de usuario y nos centramos en la narrativa: *'I want…'*

- ¿Cómo aseguramos que esta completa? Cuando el cliente no tenga más historias de usuario que contarnos

- ¿Qué testear y que no testear en cada aceptación? ¿Por donde empiezo? ¿Donde acabo? ¿Cuál debe ser mi siguiente paso? Implementa cada escenario, paso a paso, de uno en uno, siempre teniendo en mente cuál es el siguiente.

Éste es un proceso iterativo que se integra perfectamente en la forma 'ágil' de trabajar, y en el modelo SCRUM en particular. Podemos claramente identificar esas primeras 4 horas de selección de requisitos, en las que se establecen las historias de usuario a cubrir en la iteración y cómo cada miembro elige los escenarios a cumplir, y como en la reunión final de la iteración, en la demostración de requisitos, historias de usuario en mano, el cliente puede comprobar fácilmente que sus requerimientos han sido cumplidos.

Es importante seguir algunas reglas a la hora de crear las historias de usuario, tanto para que el cliente no se pierda con especificaciones demasiado complejas, como para que el desarrollador obtenga los suficientes detalles que le guíen en el trabajo. Vemos, en la Figura 6-6, un ejemplo de 'una buena historia', que ha de cumplir:

- **El título debe describir una actividad**, que será la que nuestro software facilite, si hubiera sido 'Account management' o 'ATM Activity' nos resultaría más difícil de determinar cuando hemos cumplido con la historia de usuario. Los límites quedarían confusos, difuminados.

- **La narrativa debe incluir rol, requerimiento y beneficio**. No basta solo con el requerimiento, aunque se trate del objetivo final. El rol nos ayuda a identificar con quién debemos hablar sobre el requerimiento a cumplir. El beneficio nos aclara por qué debemos cumplir el requisito, lo que nos ayuda a enfocarnos mejor. Por otra parte, si encontramos que el requerimiento a cumplir, en realidad, no nos da el beneficio indicado, esto es un indicativo de que la historia está mal planteada.

- **El título cada escenario debe ser descriptivo del ejemplo, y ser claramente diferentes unos de otros.** Nombres vagos como 'caso 1', 'caso 2' son absolutamente inapropiados. Solo con mirar el título ya deberíamos entender claramente qué estamos comprobando en el escenario, y con un vistazo general tener claro si se han contemplado todas las posibles situaciones.

- **El título del escenario debería indicar solo lo que es diferente.** Si un contexto similar se repite en todos los escenarios, no es necesario especificarlo en el título. En el ejemplo, no necesitamos empezar cada título del escenario con *"Cuando el cliente va al cajero a sacar dinero…"*

- **El escenario debe especificarse en términos Given-When-Then.** Es lo que da su potencia al BDD. El utilizar esta terminología elimina por completo las ambigüedades, y ayuda a los desarrolladores a traducirlos en test de aceptación, ya que identifica claramente los requisitos (y ayuda a determinar si se nos escapa alguno), qué debemos testear y, exactamente, qué debemos comprobar.

- **Los 'Given' deben ceñirse al contexto exclusivamente necesario para el escenario.** A la hora de realizar un test, ya sea de aceptación o unitario, debemos de intentar aislar su comportamiento de elementos externos que pudieran falsear su resultado. Por tanto, si un pre-requisito no es necesario para el escenario, debemos descartarlo.

- **El evento (when) debe especificar claramente cuál es la función a testear.** Si está bien realizado, normalmente se traducirá en una llamada simple a un método o función de nuestro código.

- **El número de escenarios debería poder cubrirse en una sola iteración.** El objetivo es que, cada vez que nuestro cliente venga a comprobar los requisitos, vea la historia, el requisito, completado. No es elegante presentarle 'medio requisito cumplido'. Si la historia resulta demasiado compleja, habrá de fraccionarse en historias más simples, cada una representando un 'subrequisito' completo que podamos presentar al usuario. En el caso anterior, podríamos dividirlos en 'El cliente retira dinero del cajero', 'El cliente intenta retirar dinero del cajero con una tarjeta inválida' y 'El cliente intenta retirar dinero de un cajero averiado'

- **El número de escenarios para cada historia no debería ser muy grande.** Normalmente debería caber en una hoja, o en un folio a dos caras. Le resulta más fácil de seguir al cliente.

- **En un escenario debemos hablar de comportamientos, no sobre detalles de implementación.** No es válido un escenario que diga: 'Si se introduce un valor inesperado, el programa lanza una excepción'. Esos son detalles del implementador, que deberá cubrir con sus TDD de desarrollo. Un escenario correcto sería 'No se aceptan ciertas entradas'

```
Story: Account Holder withdraws cash

As an Account Holder
I want to withdraw cash from an ATM
So that I can get money when the bank is closed

Scenario 1: Account has sufficient funds
Given the account balance is \$100
 And the card is valid
 And the machine contains enough money
When the Account Holder requests \$20
Then the ATM should dispense \$20
 And the account balance should be \$80
 And the card should be returned

Scenario 2: Account has insufficient funds
Given the account balance is \$10
 And the card is valid
 And the machine contains enough money
When the Account Holder requests \$20
Then the ATM should not dispense any money
 And the ATM should say there are insufficient funds
 And the account balance should be \$20
 And the card should be returned

Scenario 3: Card has been disabled
Given the card is disabled
When the Account Holder requests \$20
Then the ATM should retain the card
And the ATM should say the card has been retained

Scenario 4: The ATM has insufficient funds
...
```

Figura 6-6: Un ejemplo de 'una buena historia de usuario'
Fuente: dannorth.net

Con el BDD tenemos, por tanto, la herramienta perfecta para integrar en SCRUM:

- Sirve de intermediario entre los requisitos de usuario y la implementación con TDD de desarrollo, eliminando los problemas de 'teléfono roto' que presentan, por ejemplo, los casos de uso clásicos del desarrollo orientado a objetos.

- Sirve de intermediario entre los desarrolladores y el cliente, creando un documento intermedio, que contiene tanto lenguaje natural como una estructura formal, y ayuda a ambos a hacer un seguimiento del cumplimiento de las especificaciones del usuario, eliminando los problemas de 'teléfono roto' entre cliente y equipo, y asegurando que lo que se entregará al final es exactamente lo que pidió y necesita.

6.2.3. TDD

Ahora que en nuestra reunión de SCRUM nos han dado los requisitos en BDD, podemos empezar a integrarlos en el software mediante desarrollo TDD.

En el TDD vamos desarrollando el software paso a paso: elegimos una función que queramos añadir al sistema (si tenemos la ayuda del BDD, habremos elegido un escenario), y comprobamos si está integrada en nuestro software diseñando un primer test. En él alimentamos nuestro sistema con un ejemplo, un caso particular del que sabemos exactamente cual debe ser el resultado, y comprobamos si nuestro software devuelve lo esperado. Si no lo hace, añadimos a nuestro programa las líneas exclusivamente necesarias para que genere la salida prevista. Cuando finalmente funciona, arreglamos nuestro código para que resulte lo más fácil posible el añadir nuevas funciones en el futuro.

Hemos completado un ciclo.

Y vamos repitiendo ciclos, vamos probando ejemplo tras ejemplo, todos los que se nos ocurran, hasta estar seguros de que nuestra función responde adecuadamente en toda la posible casuística. Cuando todos los test pasen, habremos integrado en nuestro software una nueva función usando TDD.

Sencillo, ¿no?

En realidad el TDD es de una enorme simpleza conceptual. Como vemos se trata de repetir tres simples pasos:

- **Crear un test:** Un test es un pequeño fragmento de código que, al ejecutarlo, alimenta nuestro sistema con un ejemplo, un caso, del que sabemos exactamente el resultado, y recoge la salida producida, cotejándola con lo esperado. El test se crea ANTES de programar la función que lo satisface, por lo que inicialmente, el test no 'pasa'.

- **Escribir código:** Lo siguiente que hacemos es añadir a nuestro programa estrictamente las líneas necesarias para que la ejecución del test de la salida esperada, sin reocuparnos en exceso de la calidad de las mismas. Programamos para **'pasar el test'**.

- Una vez el test ya pasa, significa que nuestro programa esta preparado para responder adecuadamente el caso o ejemplo propuesto en el test. Lo que nos queda **REFACTORIZAR** el código, arreglarlo para integrar las nuevas líneas añadidas.

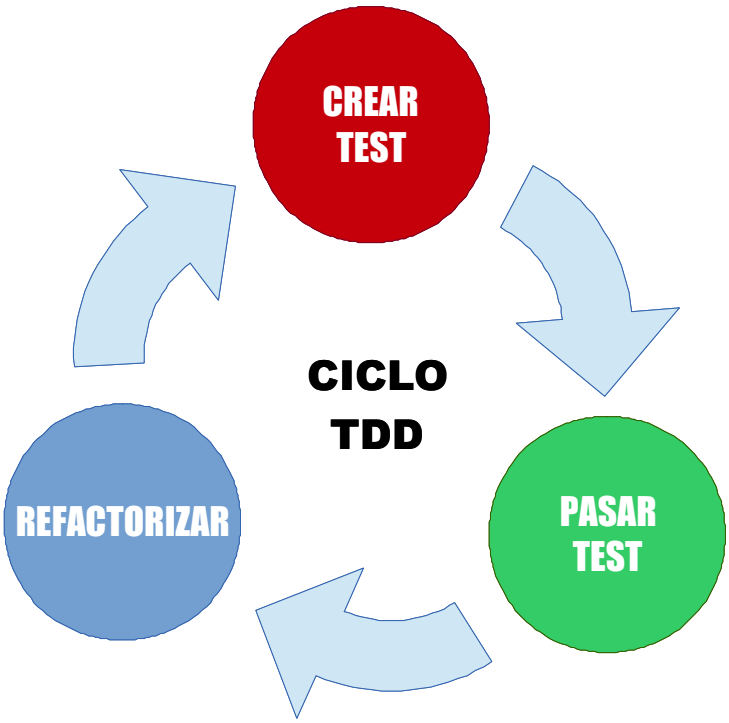


Figura 6-7: El ciclo TDD

En alusión al esquema de colores comúnmente utilizado en los diferentes marcos de trabajo para indicar si un test ha pasado o no, al ciclo TDD se le suele conocer como **Red-Green-Refactor.**

Test unitarios y test de integración

Dentro del TDD de desarrollo (dejamos fuera los test de aceptación), podemos distinguir dos tipos de test principales:

- **El test unitario (Unit Test):** Son la base sobre la que se fundamenta el TDD. Normalmente construimos en base a secuencias de test unitarios. Un test unitario ha de ser:
    - Atómico: Comprueba la mínima funcionalidad posible, normalmente un método de una clase. Si dicho método tiene diferentes entradas o contextos, cada Unit Test comprobara uno de ellos, un caso, un ejemplo. Si el método que probamos llama a su vez a otros métodos, hablamos de que el test unitario tiene menor granularidad.
    - Independiente: Un test no puede depender del resultado de otros test. No puede haber llamadas entre los test, ni el resultado final de uno intervenir como pre-requisito del siguiente, ni ser sensible al orden en que se ejecuten los tests.

- Inocuo: No debe alterar el estado del sistema. Se deben de ejecutar en un entorno aislado. No deben modificar ficheros, ni bases de datos… Da lo mismo si se ejecuta una o mil veces (normalmente lo último es lo frecuente)

- Rápido: En un entorno normal de trabajo podemos acabar con cientos o miles de tests, que deberemos ejecutar con frecuencia. Y no podemos estar esperando varios segundos cada vez.

- **El test de integración:** Cuando necesitamos ver cómo interactúan varios elementos de nuestro sistema, estamos realizando test de integración. Siguen siendo importantes la independencia, inocuidad y velocidad, pero ya no buscamos la comicidad ni la granularidad mínima. Existen varios tipos de test de integración, desde los que comprueban como interactúan un par de elementos previamente comprobados mediante test unitarios, hasta los que lanzan una entrada al sistema, recorre todo el mismo, y recoge la salida en el otro extremo.

Mocks y dobles de prueba

Hemos dicho que dos de los principios básicos que han de cumplir del los 'unit test' son el aislamiento y la inocuidad. Pero, ¿cómo podemos conseguir cuando prácticamente, todo elemento de un software (métodos, clases…) tienen dependencias?

La solución es sencilla: cada vez que nuestro código funcional vaya a hacer una llamada a otro elemento del sistema, sustituimos el método a llamar por un 'doble'. Como los de las películas de acción ☺ Las operaciones 'peligrosas' (interacciones con el sistema real tales como modificar una base de datos), las SIMULA el doble, sin interactuar realmente con nuestro sistema, y devuelve una respuesta, normalmente predefinida.

Hay varios tipos de dobles:

- *Dummy:* El más simple, normalmente un objeto que creamos para rellenar una llamada, y que no hace nada.

- *Fake:* Tiene una implementación que devuelve un resultado, pero usando un atajo, sin llamar a los métodos del código al que suplanta.

- *Stub:* Proporciona respuestas predefinidas a llamadas hechas desde los test, y puede almacenar estados.

- *Mock:* Más complejo que un stub, al mock podemos programarle 'expectativas', es decir, acepta parámetros de entrada, por lo que no devuelve resultados fijos, como el stub.

Un artículo muy interesante sobre los diferentes tipos de 'dobles' y su uso es 'Mocks aren't Stubs'[51], de Martin Fowler

A la hora de usar este tipo de dobles tenemos dos opciones: crear los nuestros propios o ahorrarnos trabajo y utilizar diferentes frameworks que permiten crear clases a medida en tiempo de ejecución que respondan a un interfaz predefinido.

Desarrollando TDD: Disciplina y requerimientos

Desarrollar en TDD es un proceso bastante repetitivo, que por sus características a veces puede resultar tedioso. Sin embargo, la calidad final del código depende de ser metódico y mantener la concentración. Requiere DISCIPLINA:

- **No escribir código sin haber añadido el test:** Debemos evitar el sucumbir a la impaciencia. Escribir el test antes de escribir código nos ayuda a centrarnos exclusivamente en lo necesario. Introducir código no testado es un riesgo, ya que podemos cometer el error de no evaluar correctamente, y podemos terminar a posteriori, con código no cubierto por test. Y código no cubierto es fuente segura de problemas en el futuro. Por tanto, algo que no debe faltar en el toolbox de un desarrollador TDD es alguna herramienta integrada el entorno de desarrollo que te informe continuamente del 'Code Coverage'[52], es decir, el porcentaje de código cubierto por los tests. Ha de mantenerse en el 100%.

- **Empieza cada ciclo con un test que falle:** Es una norma que cuesta mucho seguir cuando empiezas en el TDD. ¿Por qué no podemos directamente añadir código funcional que de 'verde'?

    - Porque has de empezar separando los errores de compilación que se producen al hacer llamadas a funciones no implementadas. Al crear un nuevo test, posiblemente introducirás errores de compilación por llamadas a elementos aún no implementados. Limpia esos errores. ¡Compila!

    - Una vez ya tenemos 'red', ya que disponemos del punto de entrada, el método sobre el que tendremos que trabajar para hacer que el test pase. Es muy importante disponer de un punto de referencia sobre el que construir el resultado del test. Si intentamos empezar en verde podemos encontrarnos escribiendo código distribuido por todo el programa, a veces sin haber siquiera limpiado los errores de compilación, y corremos el riesgo de perdernos.

- **No refactorizar hasta obtener verde:** Otro error común es intentar conseguir 'verde' a través de la refactorización en vez de añadir código. Se termina mezclando el objetivo principal (añadir una función) con el del paso siguiente, que sería integrarla. Además, puede ocurrir que, como consecuencia de la refactorización, otros test comiencen a dar 'rojo' añadiendo mayor complicación.

- **Evitar introducir nuevas funcionalidades al refactorizar:** A veces, por error, añadimos código no testado al refactorizar. Es por ello que es importante realizar esta labor de manera sistemática, y ceñirse lo más posible a una serie de reglas, y patrones, como los que reúne Sean Chambers en su publicación e-book "31 Days

Refactoring" *(Chambers, Sean. 2009)[6].* Nuevamente, una herramienta que controle el 'Code Coverage' resulta de lo más útil.

- **No codificar varios test a la vez:** Idealmente, el ciclo TDD ha de ser atómico. Esto es, un test cada vez. Si te surgen ideas sobre nuevos test, consecuencia del que estás implementando, una buena práctica es disponer de un bloc de notas abierto en el equipo para ir tomando nota. Algunas 'suites' de test permiten codificar el test como 'Inconclusive', marcándolo en amarillo, y, a veces resulta práctico como manera de mantener una lista de test 'to-do', pero es preferible mantener esa lista fuera del código.

- **No dejar de comprobar casos porque los resultados parezcan triviales**: Pueden resultar triviales, o invariantes, con la lista actual de tareas, pero más adelante con los requisitos cambiantes, pudiera causar problemas en no mantenerlos bajo control.

Por último, es necesario disponer de **un equipo de desarrollo con integrantes muy cualificados** para realizar TDD. Al no partir de documentación exhaustiva, al no existir diseño arquitectónico de clases previo, este último va emergiendo durante el desarrollo. Por tanto, el programador debe tener cierta experiencia, saber hacia donde ir, qué orientación debe dar al código con cada refactorización, cómo diseñar cada nuevo test. Sin esas destrezas, la productividad se reduce radicalmente, y, en última instancia, la misma calidad del software desarrollado. Por tanto, si no todos, al menos parte del grupo han de ser programadores 'senior', y adoptar la filosofía del 'Software Craftmanship', para que los expertos orienten en las reuniones de sincronización a los menos avezados hasta que mejoren sus habilidades.

Las ventajas del desarrollo TDD

Como vemos, el TDD revoluciona por completo la forma de trabajar. Siguiendo inalterablemente esta disciplina conseguiremos:

- **Código altamente fiable**: Por dos razones principales:

    - *Código 'a prueba de tontos':* Cuantas veces, tras desarrollar un software y entregarlo al cliente, nos encontramos que, al entrar en producción, un usuario tarde o temprano realiza algo inesperado y nuestro flamante programa finaliza lanzando un error, muchas veces no controlado. Si seguimos la metodología TDD esto es mucho más difícil que ocurra, ya que todos los escenarios previstos (gracias a BDD) se han implementado partiendo de test, y, si hemos sido diligentes, solo habremos añadido cada vez el código estrictamente necesario para hacer funcionar cada ciclo.

    - *'Si funciona, no lo toques':* Esta es una máxima que llevo toda mi vida oyendo. Los programadores solemos sentir apego por el código operativo que funciona, por una parte por el trabajo que nos ha llevado el implementarlo, y por otra por el miedo a que, al intentar arreglar algo,

termines fastidiando otra cosa. En TDD esto no tiene sentido, ya que una de las fases de cada ciclo es, precisamente, 'arreglar código que funciona'. En esta metodología podemos hacerlo con seguridad, pues durante cada reorganización se vuelven a lanzar TODOS los test, no solo el último, con lo que garantizas que cualquier elemento que hayas modificado sigue funcionando.

- **Código de alta calidad, muy reusable, y fácil de refactorizar:** El código de calidad es tanto un resultado del TDD como una necesidad del mismo. Cuando continuamente has de rediseñar las clases, partirlas, derivarlas, reorganizarlas, a menos que se acerquen lo máximo al paradigma SOLID, te encontrarás con un trabajo ingente. A veces, simplemente, imposible de afrontar, llegando a un callejón sin salida. El TDD es la herramienta perfecta para desarrollar con requisitos cambiantes.

- **Los propios son una magnífica documentación técnica:** Mirando los test podemos ver fácilmente cómo funciona y encajan cada una de las partes de nuestro software.

Además, conseguiremos a unos trabajadores motivados, pues la incertidumbre y frustraciones de un software que falla son sustituidas por una sensación diaria de un trabajo bien hecho.

TDD y productividad

Muchos detractores del TDD plantean como crítica la posible pérdida de productividad ¿No consume mucho tiempo hacer los tests?¿Y tener que reescribir continuamente el código?

Carlos Blé, en el prólogo de su libro *"Diseño Ágil con TDD"(Ble, C. 2010)[5]*, expone una parábola, un claro y entretenido ejemplo de por qué no es así. Si comparamos una metodología formal en cascada, una metodología informal, y un desarrollo TDD (**Figura 6-8**):

- La metodología en cascada tarda muchísimo tiempo en arrancar, y no produce resultados hasta el final. El cliente puede impacientarse por la falta de información.

- La metodología informal produce resultados espectaculares en los prototipos iniciales, pero suele 'atascarse' a medida que el proyecto crece, por no tener planificación ni herramientas que le ayuden en su avance. El cliente suele terminar desilusionándose ante las expectativas iniciales perdidas o ante un software final repleto de fallos.

- Las metodologías ágiles avanza lentamente, con paso firme. El cliente pronto dispondrá de resultados, muy básicos, pero, puntualmente recibirá mejoras, ve como el software crece, va mejorando, siempre siguiendo los requerimientos que

plantea tras cada nueva entrega. Al final recibirá un software robusto que se ajustará a lo que necesita.

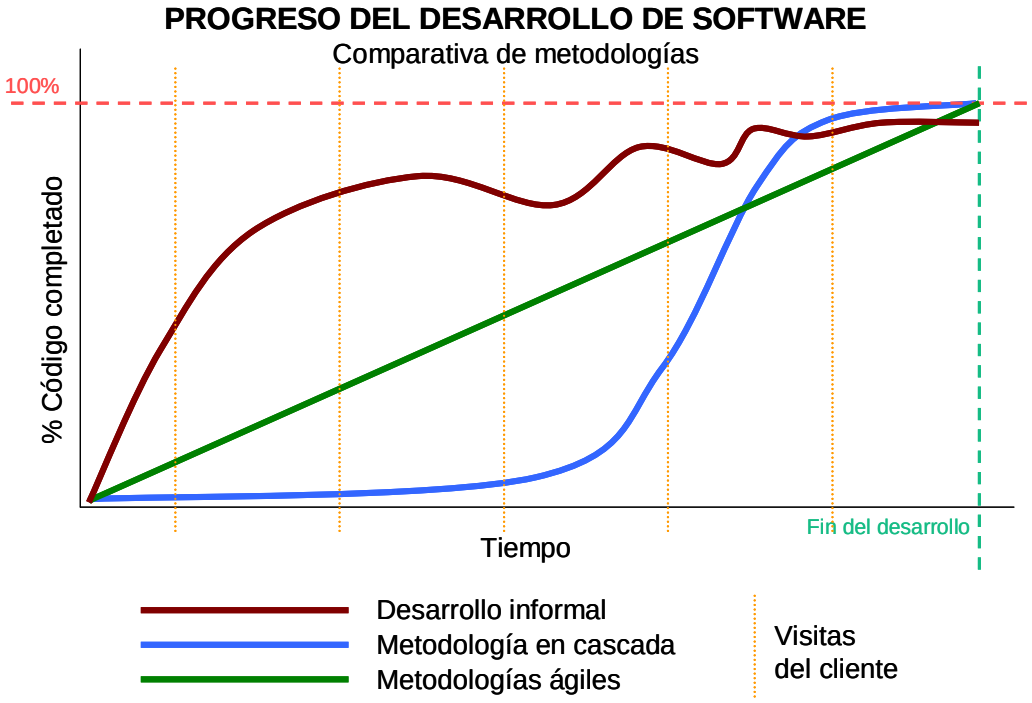


Figura 6-8: Comparativa de metodologías de desarrollo de software

Testear y refactorizar, no es, por tanto, un elemento negativo, sino que es la herramienta que mantiene al cliente satisfecho y le garantiza un software que cubra sus necesidades. En el caso, por ejemplo, del SCRUM, si su planificación es correcta y se dispone del equipo de desarrolladores adecuados, el software se entregará no solo en tiempo, sino, posiblemente, antes que con las otras dos metodologías.

6.2.4. Refactorización, code quality y clases SOLID

Atendiendo a la definición que Martin Fowler nos da al principio de su libro:

"Refactorizar es el proceso de cambiar un sistema de software de tal manera que no se altere su comportamiento hacia el exterior, pero se mejore su estructura interna"
Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional (p.8)

Por tanto, cuando refactorizamos lo que intentamos es mejorar el código existente.

- En un desarrollo clásico nos planteamos esta tarea cuando la implementación inicial de un diseño entregado por el analista ha sido totalmente desvirtuado debido a modificaciones ulteriores realizadas en el código.

- En TDD estamos continuamente modificando el diseño, luego, la única manera de 'mejorar la estructura externa sin modificar el comportamiento externo' es, precisamente, la permanente refactorización.

La refactorización, como todo en TDD, se realiza en pequeños pasos, simples, uno a uno: extraer código de un método para convertirlo en un método independiente, mover código arriba o abajo en una jerarquía, reunir código duplicado en un mismo lugar, … Para cada una de las situaciones que requieran mejora de la estructura interna, Martin Fowler define en su libro una serie de pasos sistemáticos. Las técnicas están claramente definidas, cada una con su nombre, del mismo modo que, por ejemplo, los Patrones de Diseño[53], que tan claramente especificaron sus autores en el libro del mismo nombre *(Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994)[8].* Para mantener la fluidez de este procedimiento repetitivo, las modificaciones han de ser leves. Es importante, por tanto, localizar rápidamente los puntos donde se necesita mejorar el código, antes de que sea necesaria una intervención substancial. Fowler habla de 'bad smells', código que 'huele mal', y nos da pistas sobre como localizarlo: código duplicado, métodos o clases demasiado extensas, listas de parámetros muy grandes, son pistas de que es necesaria una intervención importante

El objetivo final de la refactorización en TDD es, por tanto, MANTERNER el código dentro de unos estándares de calidad, permitiendo que la adición de nuevas funcionalidades se realice siempre de la manera más sencilla posible, con el gasto mínimo de recursos. Es esto, precisamente, lo que permite al TDD reaccionar de forma tan eficaz a los nuevos requisitos. ¿Cómo definimos esos estandares dentro de cuyos parámetros debemos movernos? Son lo principios SOLID.

Las clases SOLID

Para asegurar la portabilidad y reusabilidad del código en el diseño orientado a objetos, las clases deben de construirse siguiendo ciertos patrones. Muchos estudios se han elaborado al respecto, entre ellos el que realizó a principios del año 2000 Robert C. Martin para el website de *Object Mentor*, con el nombre de *Design Principles and Design Patterns (Martin, R, 2008)[10]*[54]

En él, Robert (conocido con el cariñoso sobrenombre de *Uncle Bob*), identificaba ciertas normas que más tarde resumiría en su ensayo The Principles of ODD[55]. En este artículo de obligada lectura, se exponen, entre otros, los 5 principios que deben seguir las clases en OOD, y que más tarde, como suele ser en el mundo anglosajón, se reunirían bajo el acrónimo SOLID. Con este nombre, las clases SOLID se han convertido en 'Los 5 Mandamientos' dentro del OOD.

Estos son los preceptos que toda clase debe seguir:

- **(S) SRP - Single Responsability Principle[56]:** Una clase debe tener una ÚNICA responsabilidad.

- **(O) OCP – Open Closed Principle[57]:** Una clase debería estar abierta a su extensión, pero cerrada a su modificación

- **(L) LSP – Liskov Substitution Principle[58]:** Una clase derivada debería poder sustituir a un su clase base.

- **(I) ISP – Interface Segregation Principle[59]:** Realizar un interfaz específico para cada cliente es mejor que no solo genérico.

- **(D) DIP – Dependency Inversion Principle[60]:** Las clases deberían depender de abstracciones, no de implementaciones.

El *principio de responsabilidad única* nos indica que cada clase debería tener una finalidad sencilla y concreta. Si detectamos que está ejerciendo varias tareas, suele ser síntoma de que deberíamos partirla y crear una clase para cada una de ellas. De esta manera, por ejemplo, si necesitamos extender una funcionalidad de una clase (por ejemplo, derivándola), no tenemos por qué 'arrastrar' con nosotros los otros cometidos. Este mismo principio podríamos aplicarlo a los métodos.

El *principio Abierto/Cerrado*, (atribuido a Bertrand Meyer *(Meyer, B. 1988)[12])*, indica que una clase debería estar abierta a su extensión, pero no deberíamos tener que modificar su implementación. Dado que durante el desarrollo, especialmente en TDD, habremos de hacer continuos cambios, y que las entidades dependen unas de otras, las modificaciones en el código de una de ellas pueden generar indeseables efectos en cascada. Aunque con los test tenemos una cierta salvaguarda, es importante seguir este principio. Debemos tener previsión y ajustarnos a SRP. A veces podremos salvarlo mediante herencia o inyección de dependencias (por ello la clase debe estar 'abierta'), pero, en otras ocasiones nos veremos obligados a romper este principio.

El *principio de Sustitución de Liskov* (postulado por Barbara Liskov[61] en 1987) habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando creamos clases derivadas debemos asegurarnos de no reimplementar métodos que hagan que los métodos de la clase base no funcionases si las instancias derivadas se tratasen como un objeto de esa clase base. Este principio esta también muy relacionado con el principio OCP.

El *principio de segregación de los interfaces* podemos verlo como una ampliación del primer principio, pero aplicado a los interfaces. Propuesto por el mismo Robert C. Martin, nos dice que cuando se definen interfaces estos deben ser específicos a una finalidad concreta. A veces, para simplificar trabajo, podemos estar tentados de obligar a los clientes a depender de clases o interfaces que no necesitan usar, por ejemplo, cuando una clase o interfaz tiene más métodos de los que un cliente (otra clase o entidad) necesita para sí mismo. En estos casos, es mejor crear nuevos interfaces más adaptados.

El *principio de la inversión de las dependencias* fue definido también por Robert C. Martin para reducir el acoplamiento entre las clases. En todo diseño siempre debe existir un

acoplamiento pero hay que evitarlo en la medida de lo posible. Un sistema no acoplado no hace nada pero un sistema altamente acoplado es muy difícil de mantener. Para ello debemos hacer uso de abstracciones siempre que sea posible, para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben los detalles de la clase inferior de la que depende. En TDD el uso de este principio es fundamental en muchos casos para poder mantener la atomicidad e independencia de los test unitarios. Existen diferentes patrones como la ***inyección de dependencias***[62] o ***service locator***[63] que nos permiten invertir el control.

Los conceptos de clases SOLID están muy relacionados con los de **alta cohesión** y **bajo acoplamiento**, que discutiremos cuando hablemos de las métricas de código.

Si estudiamos estos principios, veremos que el cumplimiento de los mismos es de mucha mayor importancia en el desarrollo TDD que en metodologías secuenciales, por dos razones:

- En TDD estamos refactorizando continuamente, y si las clases no son SOLID, supondrán un problema grave en consumo de recursos y pérdida de productividad. En otros métodos de desarrollo, donde el diseño de clases es estático (el que te entrega el ingeniero analista/diseñador), mientras cumplan con su función, como no va a haber que modificarlas ni reusarlas para otra cosa, a veces se descuida este aspecto.

- En TDD necesitamos hacer test unitarios, que deben ser atómicos y aislados. Por lo que es imprescindible reducir al máximo el acoplamiento de las clases.

Como consecuencia de ello, la labor que era responsabilidad de departamentos de QA (Quality Asurance), ha pasado cada vez a ser de mayor importancia para cualquier desarrollador. El concepto de las 'métricas de código', que veremos a continuación, y disponer de alguna herramienta que controle las mismas durante el desarrollo, es otro elemento crucial en TDD.

6.2.5. Métricas de Código

Como hemos visto, en TDD hemos de intentar mantener las clases dentro de los parámetros SOLID. De manera subjetiva podemos detectar, mediante los 'bad smells', cuando nos estamos alejando de estos patrones, pero… ¿existe alguna manera objetiva, reproducible y cuantificable de medir tales desviaciones?

Para ello disponemos de las llamadas 'métricas de código'. Mediante la medida de valores discretos de diferentes elementos de código y la aplicación de una serie de funciones, la ciencia de la computación ha definido un catálogo de valores que ayudan a los desarrolladores a tener una mejor visión de la calidad del código que están creando.

De todas ellas, las más relevantes para los principios de OOD son la cohesión y el acoplamiento

Cohesión

La cohesión es una medida de cómo las distintos componentes de un elemento de software están relacionados entre sí.

Esta métrica está vinculada con el primer principio de OOD, el de la única responsabilidad. En una clase con una sola misión, todos sus elementos (métodos y atributos) colaboran entre sí para llegar a este único objetivo. Existe una alta cohesión. Si la clase tuviera varias funciones, nos encontraríamos que dichos elementos se agruparían en torno a los diferentes cometidos, sin que existiera relación entre uno y otro grupo.

Para entenderlo mejor, veamos una métrica de la cohesión, conocida como **Lack of Cohesion of Methods (LCOM)**. Veamos unos ejemplos:

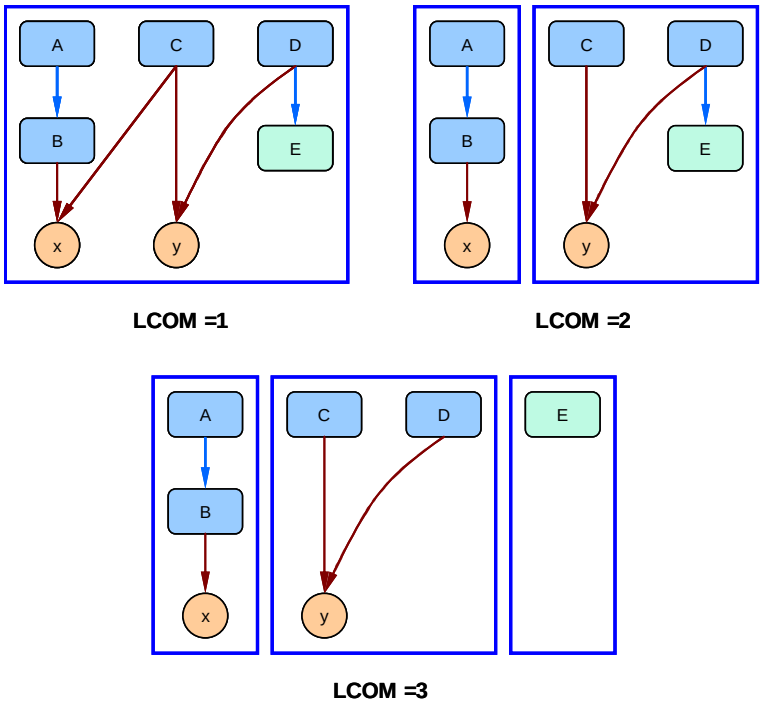


Figura 6-9: Cohesión en los métodos de una clase

En la Figura 6-9 vemos tres posibles configuraciones de una clase. Dicha clase tiene cinco métodos, A,B,C,D,y E, y dos atributos, propiedades privadas del mismo, x e y.

Premisa: Decimos que dos métodos A y B están relacionados si se cumple alguna de estas condiciones

- Dentro de A existe una llamada a B o dentro de B existe una llamada a A.
- Tanto A como B utilizan un mismo atributo de la clase

En la primera configuración de la clase, todos los métodos están relacionados entre sí: A depende de B, y b utiliza el atributo x, que también usa C. C y D comparten el uso del

atribuyo Y, y D depende de E. Dicha clase tiene la máxima cohesión, pues todos sus elementos forman un solo grupo.

En la segunda configuración los elementos se separan formando dos grupos distintos, sin relación entre ellos. Posiblemente lo que hemos detectado es que dicha clase está realizando dos funciones distintas e independientes. Según el principio SRP deberíamos partirla en dos.

En la tercera, detectamos incluso un método que 'va a su aire'. En este caso, posiblemente podríamos sacar tres clases de ésta.

La métrica LCOM nos ayuda por tanto a identificar posibles violaciones del *Single Responsability Principle.*

- LCOM = 1 → Máxima cohesión
- LCOM > 1 → Cuanto mayor sea LCOM, menor cohesión existe
- LCOM = 0 → No existen métodos en la clase. A menos que se trate de una estructura de datos, podemos considerarla una 'mala clase'

Acoplamiento

El acoplamiento hace referencia a los enlaces existente entre unidades separadas de un programa. Si existe gran dependencia a nivel de detalles de implementación entre dos clases, se dice que están fuertemente acopladas. Si todos los objetos integrantes de una clase hacen referencia directa a los objetos integrantes de la clase de la que se tiene dependencia, existe un alto acoplamiento

Idealmente las clases no deberían tener información sobre la implementación que realizan las clases de las que dependen. Si no accedemos a las dependencias a través de un interfaz, al menos deberíamos intentar al menos tener contacto con ellas a través de sus métodos públicos, nada más. Si no, se corre el riesgo de sufrir un 'efecto onda', ya que, en el caso de realizar una modificación en una clase, todas las que dependan de ella, si están demasiado acopladas, corren el riesgo de necesitar también ser modificadas.

Imaginemos, por ejemplo, el caso de una factura y su detalle. Tenemos la clase factura, entre cuyos atributos está una colección de instancias de la clase 'Linea de Detalle'. Podemos crear un método en la clase factura que vaya leyendo una a una las lineas de detalle, tomando de cada una de ellas el número de unidades y el precio por unidad, y calcular así el total de la factura. Sin embargo, esta implementación presenta alto acoplamiento. Si cambiáramos el objeto 'línea de detalle', para incluir nuevos elementos tales como descuento o impuestos para cada operación, esto nos exigiría a cambiar el método de cálculo en el objeto 'Factura'. ¿Por qué? Porque estamos obligando a la factura a usar detalles de implementación de 'Linea de Detalle'. ¿Solución? Que sea la línea de detalle la que calcule su propio total, y presente al exterior un interfaz (un simple método público) llamado 'Total de Línea', que es, en realidad, lo único que la Factura necesita conocer del detalle para calcular el total de la misma.

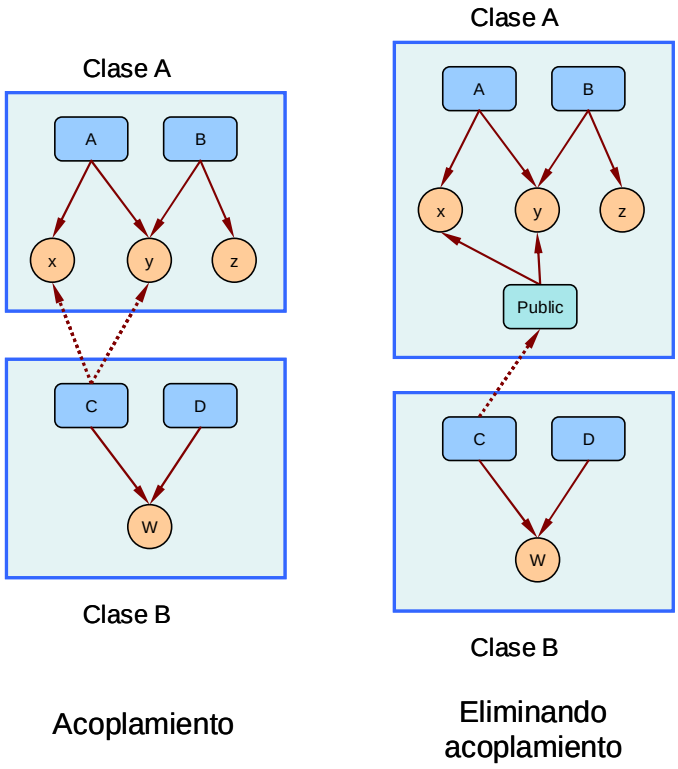


Figura 6-10: Eliminando el acoplamiento entre clases

En la Figura 6-10 esquematizamos lo anteriormente expuesto, donde eliminamos la referencia directa de un método a atributos de otra clase. No quiere esto decir que un se deba nunca acceder a los atributos desde las clases de las que se depende, sino que as clases deberían presentar solamente atributos que enmascaren detalles de implementación.

Un ejemplo[64] en C# muy simple, a la vez que ilustrativo, de acoplamiento y como solventarlo, lo podemos encontrar en la web de StackExchange por cortesía de Wedge

Las métricas principales que nos ayudan a discernir el nivel de acoplamiento son:

- **Afferent Coupling (Ca):** El acoplamiento aferente mide el número de elementos de código que dependen directamente de éste. Es decir: ¿cuántas veces me usan?

- **Efferent Coupling (Ce):** El acoplamiento eferente mide el número de elementos de código del que dependo directamente. Es decir: ¿a cuántos uso yo?

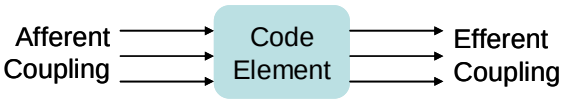


Figura 6-11: Acoplamientos aferente y eferente

Estas dos métricas se pueden definir a varios niveles:

- *A nivel de ensamblado*: ¿Cuántos tipos (clases, estructuras, enumeraciones) de otros ensamblados dependen de mi? ¿De cuántos tipos de otros ensamblados hago uso?

- *A nivel de espacios de nombres*: Exactamente igual que el anterior

- *A nivel de clases*: ¿Cuántas clases dependen de mí? ¿De cuántas clases dependo?

- *A nivel de métodos*: ¿Cuántos métodos dependen de mí, independientemente de la clase o ensamblado de procedencia? ¿De cuantos métodos dependo?

Las métricas sobre acoplamiento nos permiten ver nuestro grado de cumplimiento del quinto principio de OOD, el *Dependency Inversion Principle.*

Otras métricas de código importantes

Existen gran cantidad de métricas que nos permiten vigilar el grado de cohesión y el acoplamiento, aparte de otros indicadores sobre la calidad de código:

- **Número de líneas de código:** Aparte de ayudarnos a medir la productividad de nuestro equipo al ver como evoluciona temporalmente, el número de líneas de código en cada método es un indicativo de su índice de responsabilidad y legibilidad. Los métodos muy extensos son difíciles de mantener.

- **Porcentaje de líneas de comentario:** Como luego veremos al hablar del Clean code, dentro del agilismo se considera que comentar código es una mala practica, por varias razones

    - Si necesitas comentar código es síntoma de que éste es 'poco claro', ya sea por su extensión o complejidad. El 'Clean code' de TDD no necesita ser comentado, ya que sus métodos no se suelen extender más allá de 5 ó 6 líneas.

    - El código necesita ser continuamente refactiorizado, y aquello que se comentó puede termina no estando ahí, o ser solo una implementación parcial, con lo que los comentarios pueden terminar llevando a confusión.

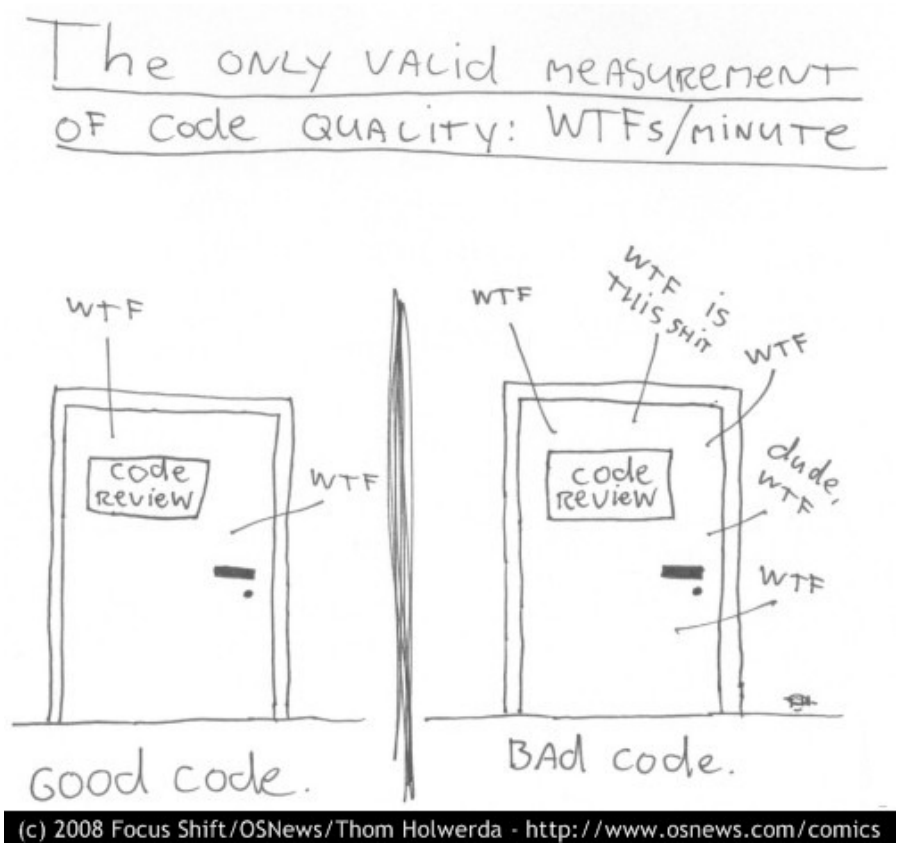
- **Porcentaje de código cubierto:** Absolutamente imprescindible en TDD es asegurarnos que todo el código que escribimos está cubierto por test. Si seguimos las directrices que expusimos al hablar de la disciplina en TDD, no deberíamos de tener problemas con ello.

- **Complejidad Ciclomática (Cyclomatic Complexity):** Esta métrica es una clara medida de la legibilidad el código, así como la dificultad que implicaría su refactorización. Indica el numero de caminos que toma el código a lo largo de un método, y viene definida por la cantidad de decisiones (bloques if, sentencias switch…) que lo integran. Está ligada en cierto modo al segundo principio de la OOD, el *'Open/Close Principle'*, que nos insta a, en vez de hacer uso de las

sentencias condicionales cuando entre un nuevo caso, optar por hacer clases derivadas o usar patrones decoradores.

6.2.6. Clean Code

¿Qué es el Clean Code? ¿Cómo medirlo?



Hasta ahora nos hemos preocupado de la calidad del código desde el punto de vista de la eficiencia y reusabilidad. Los buenos programadores son especialistas en crear este tipo de código. Y, además, suelen solucionar los problemas de manera ingeniosa. Pero ello no implica 'Clean Code'. A veces el resultado es exactamente el opuesto. Un programador puede sentirse tentado de sustituir una expresión relativamente extensa, por otra que hace todo en una sola línea, en un alarde de intelecto, regodeándose en su autocomplacencia, acompañándolo todo con un comentario: '//este fragmento de código sirve para…'

En la actualidad ha llegado una corriente que aboga por añadir a la ecuación del 'quality code' los conceptos de **legibilidad y expresividad**.

Un código limpio expresa claramente su intención, de forma simple y elegante. Ha de facilitar su entendimiento tanto para el propio programador como para los futuros mantenedores del código. Ha de acercarse lo máximo posible al lenguaje natural.

Podemos considerar al libro *Clean Code (Martin, R. 2008)[11]* como lectura de absoluta referencia dentro de este paradigma. En él 'uncle Bob' nos presenta algunos patrones para llegar a obtener 'código limpio'. Expongamos, por ejemplo, un par de ideas sobre la construcción de métodos y el uso de variables:

- **Usar 'meaningfull names', nombres con significado:** En los entornos de desarrollo actuales no suele haber limitaciones con la longitud de los nombres de variables, clases o métodos. En vez de simples y crípticas iniciales, las propiedades deberían nombrar la entidad a la que representan, los métodos la función que realizan. Tomemos el ejemplo las variables:

```
int d;            //día del mes
int diaDelMes;
```

  ¿No resulta mucho mejor la segunda opción? Podremos seguir la variable con mucha mayor facilidad en el código.

  Los métodos también deberían tener nombres descriptivos. Por, por ejemplo, en este 'snippet' de java:

```
public static String RenderPageWithSetupsAndTeardowns(
      PageData pageData, boolean isSuite) throws Exception {
      if (isTestPage(pageData))
            includeSetupAndTeardownPages(pageData, isSuite);
      return pageData.getHtml();
      }
```

- **¡Hazlo corto!:** Los métodos extensos son difíciles de seguir y mantener. Se deberían extraer bloques de código (patroón de refactorizacion 'Extract méthod') a submétodos con 'meaningfull names'. Los métodos, de tamaño medio, deberían ser 3 o 4 líneas, incluso menos.

- **Cada método, una tarea:** Del mismo modo que ocurre con el Single Responsability Principle para las clases, cada método debería hacer UNA sola cosa. Nuevamente, se deba hacer uso del 'Extract method'.

- **Los métodos deben tener pocos (o ningún) parmetro**

- **Usar excepciones antes de retorno de códigos de error.**

Y, siguiendo esta idea, nos ofrece decenas de principios a seguir a la hora de trabajar con atributos y métodos, dar formato al código, normas sobre cuando comentar (en realidad, casi nunca), objetos y estructuras de datos, clases, manejo de errores... Asimismo, como hizo en su libro de refactorización, nos da pistas sobre cómo localizar los 'bad smells'.

No existen métricas que directamente nos ayuden a detectar el Clean Code. Algunas, como el *número de líneas de código (LOC)* o la *complejidad ciclomática* nos pueden

alertar sobre 'bad smells' en código. Pero solo con echarle un vistazo, se nota la falta de 'WTF!!!' ☺

¿Hay que comentar el código?

Como regla general, en TDD no se comenta el código, por dos razones principales:

- No se debe comentar el código porque en las continuas refactorizaciones los comentarios a una función o fragmento de código pueden dejar de tener sentido, y llevar a confusión. Resulta normalmente un engorro el tener que 'refactorizar los comentarios'

- No es necesario comentar el código si seguimos las normas del Clean Code

Sin embargo, ésta es una máxima no absoluta. En ocasiones, cuando lo que se implementa es una biblioteca prácticamente estática, con unos objetos claramente definidos que deberán ser usados por otros, los comentarios son bienvenidos, no para aclarar el código, sino para documentar las clases y sus atributos.

Este caso queda claramente reflejado en las clases que implementan el estándar ISO20022 para el envío de recibos domiciliados, donde, dada su complejidad, resulta prácticamente imprescindible añadir estos comentarios que nos guíen en la construcción del esquema.

Productividad y 'Clean Code'

¿Es necesario dividir y dividir, utilizar nombre largos más lentos de escribir, convertir expresiones oscuras pero concisas en una serie de métodos y llamadas…? ¿No sería suficiente con comentar adecuadamente? Pues… no.

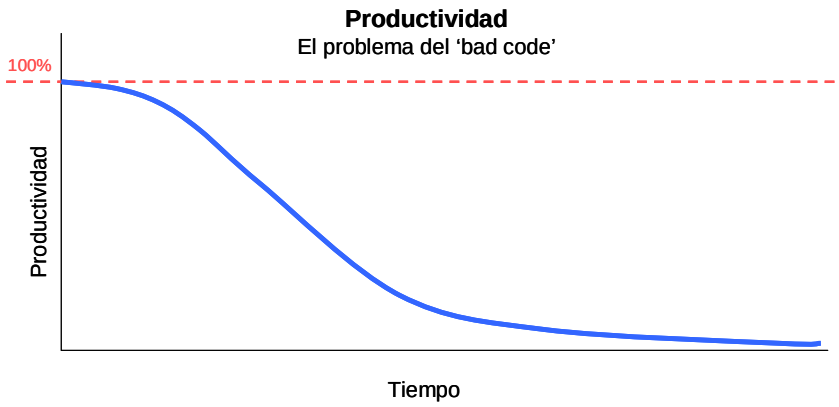


Figura 6-12: El problema de la productividad en el 'bad code'

Comentar no es la solución. Es más, es totalmente desaconsejable, especialmente en TDD, donde, como dijimos, al tener que modificar código ya escrito en cada ciclo, los comentarios pueden dejar de tener sentido. Los entornos de desarrollo actuales nos apoyan enormemente con el 'código limpio'. Incluyen ayudas a la refactorización y autocompletar.

La verdad es que la productividad en TDD aumenta enormemente usando Clean Code. A medida que crece, un código difícilmente legible se convierte en un enorme caos difícil de mantener y ampliar, y, teniendo en cuanta que debemos refactorizarlo continuamente, la productividad decae como podemos ver en la Figura 6-12.

Eficiencia .Vs. Clean Code

"Programs must be written for people to read, and only incidentally for machines to execute."
[Abelson, H., Sussman G. 1984][1][65]

Esta frase, escrita en el año 1984 por dos profesores del MIT, resume en ella misma la respuesta a esta cuestión.

Ha pasado mucho tiempo desde la época en la que programábamos en ensamblador, donde lo primordial era la eficiencia y la economía. En la actualidad podemos permitirnos escribir código claro, y confiar en la calidad de los compiladores para que la posible pequeña disminución de rendimiento sea llevada al mínimo. No quiere decir esto que debamos olvidar factores importantes de implementación. Una búsqueda en árbol es mucho más rápida que una secuencial. No debemos elegir la búsqueda secuencial porque sea más clara. Debemos centrarnos en cuál es la manera más 'limpia' de implementar un algoritmo, no que algoritmo es más claro.

La red está plagada sobre eficiencia de código, por ejemplo: *¿Qué es mejor, usar LINQ o un bucle foreach?* Pues… depende. Una sentencia LINQ tiene un ligero menor rendimiento que un bucle iterativo. Por el contrario, para expresiones simples, es mucho mas clara, legible y portable que todo un bloque de código. Por el contrario, a medida que las expresiones lambda se anidan y complican en una sentencia LINQ, empieza a ser preferible la más extensa, pero más clara, solución de bucle iterativo. Para casos como este, donde la diferencia de rendimiento es poco apreciable, no hay duda: tomar siempre la opcion 'limpia'. Y así será en la gran mayoría de los casos. Es por ello que casi todos los expertos coinciden en la importancia del Clean Code sobre la eficiencia del código.

En otras situaciones, en sistemas clave, donde la eficiencia es crítica, debemos estudiar bien las posibilidades de usar 'unclean' code. ¿Deberemos sacrificar el la productividad de nuestro equipo de desarrollo obligándoles a crear código 'sucio' pero eficiente, en pro de la eficiencia del código una vez en producción? Hagan números…

6.2.7. Software Craftmanship

El primer concepto de modelo en cascada[66] fue una adaptación de los métodos utilizados en otras ingenierías como la construcción o la manufacturación. Es por ello que, en esta metodología es posible establecer una muy fácil analogía entre los obreros de una cadena de montaje y los programadores. Cada uno es, simplemente, un elemento de una maquinaria que realiza un trabajo muy específico y puntual, atendiendo a unas normas rígidas (el documento de diseño), sin tener necesidad de tener conocimiento alguno de cual es la función de su trabajo. Como partes de una máquina, su perfil está perfectamente definido y especificado, por lo que en muchos casos se les considera prescindibles o intercambiables por otra pieza similar.

Frente a esta imagen, el agilismo reivindicó el papel del programador, no como una pieza de una maquinaria bien engrasada, sino cono un artesano, enfatizando las habilidades como programadores que cada desarrollador debe tener, y en como debe preocuparse por un software bien hecho, del que forman parte íntegra. Estamos hablando de la 'artesanía de software', o 'Software Craftmanship'.

En esta analogía con las cofradías de origen medieval, se presta gran atención al programador no como un ente aislado, sino como integrante de una comunidad, donde los programadores 'senior', maestros de la cofradía, se preocupan por ayudar a los 'junior', los aprendices. Forma parte de su rutina diaria aprender y enseñar, mejorar, e imprimir a su oficio ese sello de calidad que conlleva el orgullo del trabajo bien hecho.

En cada equipo de trabajo 'agil' se siguen los preceptos del software craftmanship, por lo que se trata de grupos de personas motivadas y de gran capacidad, realmente expertos. Cada uno de los integrantes conoce bien el trabajo de los demás, y participa del mismo en las reuniones diarias de sincronizaión. En un equipo 'agil' los desarrolladores menos expertos reciben en cada sincronización el apoyo de los más expertos. En el SCRUM, por ejemplo, tienen la ayuda de un 'facilitador'. A veces se utiliza la programación por parejas, el 'pair programming'[67].

Es por ello que la calidad del software en los proyectos ágiles es mejor que la resultante en la metodología en cascada, precisamente por la primera premisa del manifiesto ágil:

"Valorar más a los individuos y su interacción que a los procesos y herramientas"

7. ANÁLISIS

7.1. RCNGC Members Management

Para el presente estudio se ha realizado un análisis parcial de los requerimientos, centrándose en elementos de la facturación de recibos de una sociedad deportiva. Se han implementado unas bibliotecas de clases sin interfaz de usuario, suficientes para evaluar el TDD. Se ha prestado especial atención a los elementos necesarios para la implementación del sistema de adeudos domiciliados SEPA.

7.1.1. Requisitos

De manera muy resumida, los requisitos recogidos en la librería:

- *Creación Gestión de Miembros:* A nivel muy simple, tan solo el miembro propietario, y centrándonos en lo estrictamente necesario para gestión de las facturas

- *Servicios y Productos:* Los miembros del club pueden utilizar servicios y productos, y como resultado se generan ventas o cargos por servicios.

- *Gestión de Facturas:* Las ventas y cargos por servicios se anotan en facturas que se registran al socio.

    - Cada factura puede recoger varios cargos o servicios.
    - Existe la posibilidad de crear facturas pro-forma.
    - Las facturas pueden ser anuladas creándose una factura de anulación.

- *Gestión de Recibos:* Las facturas se cobran mediante recibos.

    - Una factura puede ser pagada en varios recibos
    - Los recibos pueden ser cobrados por caja o mediante adeudo en cuenta. En cualquier momento puede cambiar cual es su modo preferido de pago. Si se vence su fecha de cobro pasan a estar impagados. Los recibos pueden ser anulados, o ser considerados como 'fallidos' si es imposible su cobro (por ejemplo, en el caso de baja del socio)
    - Los recibos pueden ser renegociados en plazos, creándose un acuerdo de pago. Se hace un seguimiento de los acuerdos de pago de cada recibo.

- *Gestión de domiciliaciones bancarias:* El socio puede autorizar el envío de los recibos sobre su cuenta a través de domiciliaciones bancarias

  - El socio puede disponer de varias domiciliaciones bancarias, y elegir cuál desea usar para los diferentes pagos
  - El socio puede cambiar el número de cuenta bancaria asociada a una domiciliación, manteniéndose un historial de cuentas
- *Gestión de Remesas de Recibos:* Los recibos domiciliados se envían al banco mediante remesas que siguen el formato SEPA para el envío de Adeudos Directos
  - El club puede establecer diferentes contratos de envío con distintas entidades bancarias, incluso varios con una misma entidad.
  - Cuando desee iniciar una remesa de adeudos, elige el contrato a utilizar, y va añadiendo los recibos que desee cobrar. Cuando esté completa, lanza el proceso de generación del mensaje (fichero) *CustomerDirectDebitInitiation* conforme al estándar ISO20022 XML pain.008.001.02

En un análisis normal orientado a objetos, el siguiente paso sería confeccionar los Casos de Uso. Sin embargo, aunque son perfectamente compatibles, como estamos trabajando en TDD, y hemos confeccionado las Historias de Usuario de BDD, que detallaremos mediante esta herramienta.

7.1.2. Historias de usuario

La Especificación de requisitos la modelamos mediante Historias de Usuario BDD. Al ser de cierta extensión, recogemos los mismos en el ANEXO 1

7.1.3. Restricciones impuestas por el proyecto

En el desarrollo de las librerías se han establecido algunas restricciones que pasamos a referir:

7.1.3.1. Por las características del estudio

La necesidad de llevar un histórico de la evolución de las métricas, en ciertas ocasiones ha limitado las posibilidades de la refactorización, puesto que al cambiar los nombres de los tipos se pierde el enlace con su histórico de medidas. Hemos intentado limitar al mímino este tipo de situaciones, sin romper nuestro compromiso con la calidad del código. En casos como los espacios de nombres ha sido especialmente complejo, ya que cualquier cambio en su nombre implica la ruptura con el historial de todos sus tipos, y mover un tipo a una nuevo namespace produce el mismo resultado en el mismo.

7.1.3.2. Desarrollo sin 'persistencia de datos

Una de las cosas mas difíciles ha sido trabajar evitando el concepto de la 'persistencia de datos'. Se me impuso como requisito de software el no pensar en bases de datos, para no contaminar los requisitos de usuario con requerimientos de software, ya que llevo muchos años trabajando con sistemas de información.

¿Trabajar sin acceso a persistencia de datos? ¿Sin almacenar algo en disco? En un principio parece algo difícil de asimilar. Por ejemplo, cuando pensamos en una factura, automáticamente nos viene a la cabeza el cuerpo de la factura, que contiene un montón de líneas de conceptos. La primera idea que le viene a uno a la hora de hacer el diseño de clases (al menos, a mi), es: cada factura tiene un ID (cierto), y luego, para cada línea de la factura tendrá su propio ID, el ID de la factura de la que depende… ¡FALSO! ¡Esos dos atributos son requerimientos de la persistencia de datos, nada más! Si no hay persistencia, si solo hay objetos que tiene su vida en memoria, cada instancia de objeto 'Factura' contiene una colección de instancias de objetos tipo 'Concepto'. No hay necesidad de atributos externos que la relacionen.

Quizá podamos entenderlo mejor con un símil. Pensemos en una entidad, que llamaremos 'Habitación', que contiene una serie, una colección de objetos que abstraeremos como 'Muebles' (pueden ser sillas, mesas, lámparas…). La habitación contiene esos elementos, pero nosotros no tenemos ninguna necesidad de marcar cada mueble como 'está dentro de esta habitación'. Dicha relación queda reflejada en el diagrama de la Figura 7-1

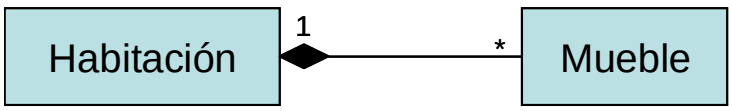


Figura 7-1: Relación simple de agregación entre una habitación y sus muebles

Pero, un día, nos toca hacer reformas en la habitación, por lo que tenemos que vaciarla y contratar los servicios de un guardamueble. Éste será nuestro 'proveedor de persistencia'. Pero, una vez se los entreguemos, ya no existe vinculación entre cada habitación y sus muebles. ¿Cómo los podremos recuperar luego, entre todos los muebles que nuestro proveedor almacena en su empresa? Sencillo. MARCAMOS nuestros muebles. Tenemos dos opciones:

- Marcamos cada mueble con una identificación, hacemos una relación, y la guardamos en casa
- O, simplemente, marcamos cada mueble con nuestro nombre

Con una de estas dos opciones, representadas en la Figura 7-2, podremos recuperarlos luego.

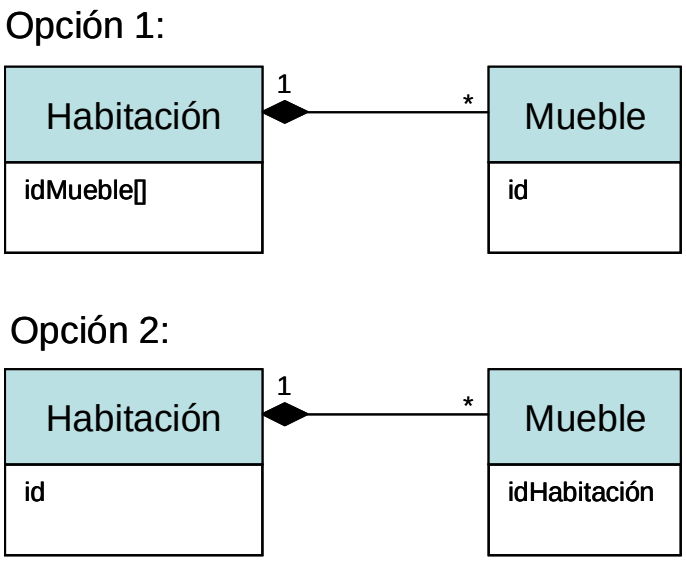


Figura 7-2: Dos opciones para añadir persistencia, y los atributos que serian necesarios añadir

Este ejemplo es fácilmente trasladable al caso de las facturas y sus conceptos.

Entonces, podemos trabajar sin persistencia, pero ¿no quedará el diseño incompleto? ¡No! En realidad el diseño será de mucha mayor calidad, pues la persistencia no forma parte de los requerimientos del usuario. En TDD la persistencia de datos es un problema que puede (y suele) abordarse 'a posteriori', para no mezclar lo que nos pide el cliente con lo que demanda la arquitectura final de datos.

¿Cómo lo hacemos? Mediante una sabia mezcla de clases derivadas y asegurarnos de seguir a pie juntillas el quinto principio SOLID en todas las clases que requieran persistencia.

- Para cada clase que requiera ser almacenada creamos una clase derivada que incluya los atributos de persistencia (los ID y demás campos necesarios para almacenar)

- Incluimos también los métodos que servirán para almacenar y recuperar los datos, pero NO LOS IMPLEMENTAMOS.

- Creamos una clase que es la que realmente accede a los datos, una clase 'repositorio', que ofrezca como interfaz publica unos métodos genéricos para leer y escribir datos y que dentro de sus métodos privados incluya toda la lógica necesaria para intercambiar los mismos con el sistema de almacenamiento elegido.

- Preparamos los constructores de las anteriormente comentadas clases derivadas con persistencia para INYECTAR instancias de los objetos repositorios. Es lo que se

llama 'Inyección de Colaboradores'. Esa inyección, como antes comentamos, ha de seguir el 5º principio SOLID: ha de depender de una abstracción. Es decir, el constructor ha de declarar una INTERFAZ, para que el colaborador pueda ser de cualquier clase que lo implemente.

- Ahora sí, implementamos los métodos de acceso a datos de las clases derivadas para que llamen a los métodos de acceso a datos del colaborador inyectado, que como sigue una interfaz, serán estándares.

Con todo ello hemos separado por completo el diseño del dominio del problema del dominio la arquitectura de base de datos, por lo que:

- Podemos trabajar en uno y en otro por separado, sin interferencias
- Podemos usar varios dominios de arquitectura de base de datos con el mismo dominio del problema. Tan solo, a la hora de inyectar el colaborador, elegimos el que necesitemos
- Y como consecuencia de lo anterior, el TDD nos está infinitamente agradecido, pues mientras que en el código que se esta implementando para producción las clases instancias se inicializan con colaboradores reales, instancias reales del repositorio, en nuestros test las inicializamos con STUBS.

En la Figura 7-3 tenemos un esquema de todo lo anteriormente expuesto.

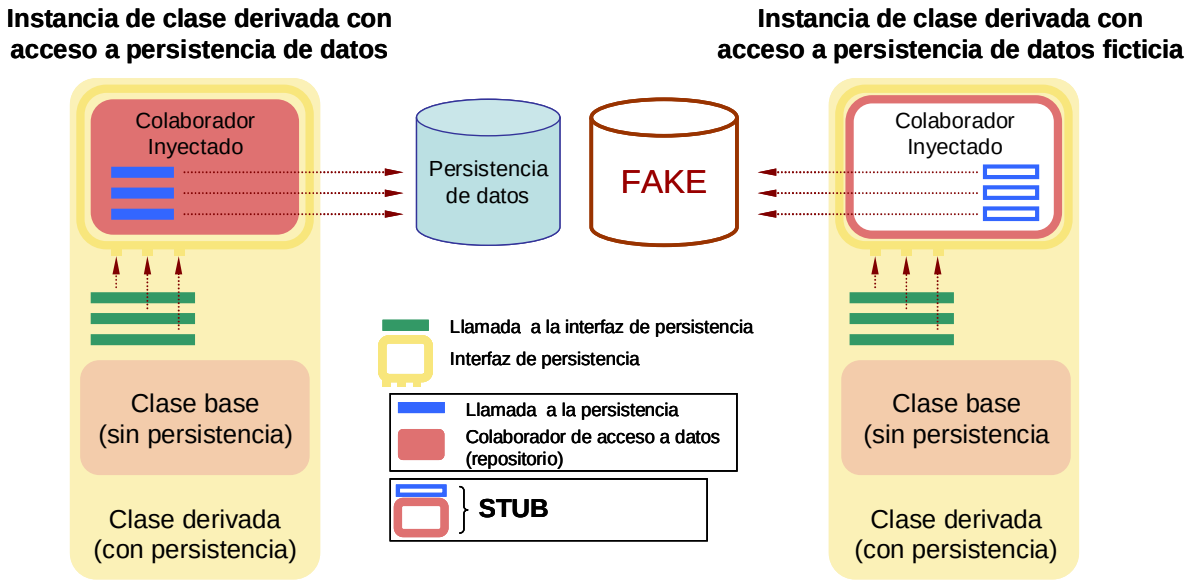


Figura 7-3: Implementando persistencia de datos mediante patrón repositorio, con inyección de colaboradores y uso de STUBS para simular acceso a datos

En el blog de Remondo.Net puedes encontrar una muy sencilla y clara implementación[68] del patrón repositorio en C#.

7.1.4. Diagramas de Clases y Dependencias

Como resultado del desarrollo TDD ha quedado conformada una arquitectura de software que representaremos mediante diagramas de clases y de dependencias. Al igual que en el caso anterior, dada su extensión, la recogemos en el ANEXO 1

7.2. NDepend Metrics Reporter

Para obtener las métricas de código disponemos de la herramienta NDepend. Sin embargo existían algunas necesidades que no estaban cubiertas con la versión 4.5:

- Gráficos históricos de evolución.
- Histogramas de frecuencias para análisis estadístico de las métricas
- Algunos valores estadísticos (medias y medidas de dispersión).

Inicialmente para satisfacer estas necesidades, se iba a utilizar un proceso de captura y volcado en base de datos, para interrogar posteriormente. Finalmente decidimos crear NDepend Metrics Reporter, que fue creciendo hasta incluir otras funciones. Los requisitos cubiertos con NDepend Metrics Reporter son:

- Consultar métricas individuales sobre elementos de código
    - Ensamblados
    - Espacios de Nombres
    - Tipos
    - Métodos
- Visualizar las métricas en formato tabular para su comparación, permitiendo su ordenación.
- Distinguir entre métricas sobre el código de la aplicación, código asociado a los test de aceptación de las historias de usuario, y código de los test de desarrollo TDD.
- Permitir inclusión de métricas personalizadas
- Visualización de valores estadísticos (medias, máximos, mínimos, desviaciones) sobre los valores de una métrica para todo un conjunto de elementos. Por ejemplo, media de la métrica *'número de líneas de código'* para todos los métodos de un ensamblado.
- Visualización de Histogramas de Frecuencias sobre los valores de una métrica para todo un conjunto de elementos
- Visualización de evolución temporal de una métrica

Dado que no era el objetivo del presente TFG realizar esta herramienta, para no extender la presente memoria más allá de lo necesario y ceñirnos a RCNGC Members Management, no incluiremos en ella diagramas referentes a este desarrollo.

8. REQUISITOS DE HARDWARE Y SOFTWARE

Dado que el presente TFG no tiene como resultado un producto final para producción, este apartado sería en un principio innecesario. Sin embargo, la herramienta NDepend Metrics Reporter hace uso de una librería propietaria, por lo que tanto para la correcta compilación del código como para la instalación del ejecutable final, existen una serie de requerimientos de software.

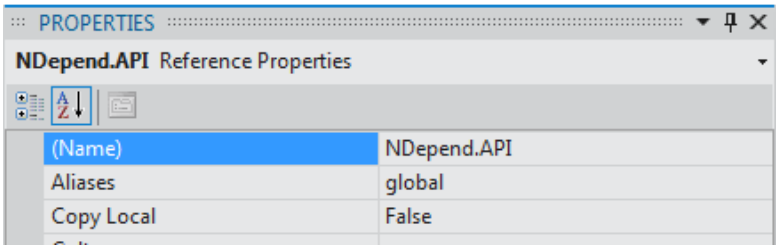
8.1. NDepend Metrics Reporter

NDepend Metrics Reporter (NDMR) hace uso de una API propietaria de NDepend. Para evitar problemas de copyright, no está permitido incluir el .dll de dicha API con la instalación de NDMR, sino que el sistema de destino debe disponer de su propia licencia de NDepend instalada y activa. Esto significa que, si bien deberemos incluir una copia de la librería en nuestro proyecto, para poder referenciarla durante nuestro desarrollo, no debemos copiarla junto con los ejecutables finales. La referencia a dicha librería de API en el ejecutable final deberá resolverse en tiempo de ejecución, localizando el directorio de instalación del paquete NDepend.

Por tanto, se requieren ciertos pasos previos y configurar algunos elementos del proyecto antes de poder comenzar:

- Disponer de una licencia activa del producto NDepend, que deberá estar correctamente instalada.

- En las propiedades de la referencia a NDepend.API hemos de indicar que no se copie la librería al directorio de destino de los ejecutables, estableciendo Copy Local a 'false'.



- El ejecutable debe tener como directorio de destino la carpeta de instalación de NDepend: `$NDependInstallPath$\`

- La librería debe ser cargada en tiempo de ejecución para ello:

    - Anadiremos a nuestro proyecto NDMR la clase que podemos encontrar en: `$NDependInstallPath$\NDepend.PowerTools.SourceCode\AssemblyResolverHelper.cs`

- Cargaremos la librería haciendo una llamada al evento al inicio del programa

```
AppDomain.CurrentDomain.AssemblyResolve +=
AssemblyResolverHelper.AssemblyResolveHandler;
```

```
namespace NDepend.PowerTools {
   internal static class AssemblyResolverHelper {
      internal static Assembly AssemblyResolveHandler(object sender,
                               ResolveEventArgs args) {
         var assemblyName = new AssemblyName(args.Name);
         Debug.Assert(assemblyName != null);
         var assemblyNameString = assemblyName.Name;
         Debug.Assert(assemblyNameString != null);

         // Special treatment for NDepend.API and NDepend.Core because they
are defined in $NDependInstallDir$\Lib
         if (assemblyNameString != "NDepend.API" &&
             assemblyNameString != "NDepend.Core") {
            return null;
         }
         string binPath =

System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location) +
               System.IO.Path.DirectorySeparatorChar +
               "Lib" +
               System.IO.Path.DirectorySeparatorChar;

         const string extension = ".dll";

         var assembly = Assembly.LoadFrom(binPath + assemblyNameString +
extension);
         return assembly;
      }
   }
}
```

Figura 8-1: Código que debemos incluir en nuestro proyecto para resolver y cargar en tiempo de ejecución la librería del a API de NDepend

- Los ensamblados de NDepend contienen el código tras los interfaces definidos en la API, por lo que todos los ensamblados de NDepend deben tambien estar presentes en `$NDependInstallPath$\` y `$NDependInstallPath$\Lib`

- Los equipos de destino para la instalación o re-deploy de NDMR deben disponer de su licencia, o copiar la licencia del sistema de desarrollo, respetando las condiciones establecidas para puestos 'Developer' y 'Build Machine'.

9. DESARROLLO Y HERRAMIENTAS

El presente TFG es un proyecto de estudio y análisis de nuevas metodologías. Por tanto, en la parte del desarrollo haremos una exposición de las herramientas empleadas para realizar el mismo. La presentación de las herramientas utilizadas suponen además una guía para cualquier equipo de desarrolladores que deseen introducirse en al Agilismo.

9.1. El sistema de desarrollo

9.1.1. Hardware

El equipo de desarrollo ha sido un sistema PC compatible de las siguientes características principales:

- Procesador Intel Core i5 3570K 3.4Ghz
- 8Gb Memoria DDR5
- Disco duro 1Tb SATA 3 7200
- Red cableada velocidad 1Gbit, con conexión a Internet 100Mb
- Monitor 27” 16:9 trabajando a una resolución de 1920:1080

Para facilitar el aislamiento y las copias de seguridad, se creó una máquina virtual con el siguiente hardware simulado:

- 3,5 Gb de memoria
- Disco duro de 80 Gb
- El dispositivo de red virtual mantuvo la velocidad de 1Gbit
- El monitor mantuvo la resolución nativa de 1920:1080

9.1.2. Software

El Sistema Operativo del equipo de desarrollo es un Windows 8 64 bits, en el que se ha instalado Oracle VM Virtual Box v.4.2.12 para crear una máquina virtual que corre el sistema final. En dicha máquina virtual, cuyas características antes expusimos, se ha montado un Windows 7 Embedded de 32 bits, y se ha desarrollado bajo Visual Studio 2012.

Las licencias del software de desarrollo son:

- Windows 8 64 bit: Licencia personal
- Oracle VM Virtual Box: Licencia GNU
- Windows 7 Embedded: Licencia de la ULPGC para sus alumnos, descargada a través de DreamSpark[69]
- Visual Studio 2012: Licencia de la ULPGC para sus alumnos, descargada a través de DreamSpark

Me gustaría destacar el perfecto comportamiento de la máquina virtual, que, con las especificaciones asignadas, ha respondido como si estuviéramos trabajando directamente con la máquina física.

Dentro del Visual Studio 2012 se han montado una serie de herramientas que pasamos a presentar.

9.2. Herramientas de desarrollo

9.2.1. Git

Dado que era necesario hacer un estudio temporal de la evolución de la calidad del código, desde un principio se planteó la necesidad de hacer uso de algún sistema repositorio donde almacenar el historial de cambios.

Finalmente se optó por el uso de un Sistema de Control de Versiones (SVC[70]), en particular la herramienta SCM (Source Control Management) conocida como Git[71], principalmente por la facilidad que permitía para almacenar una copia del código en la nube a través del GitHub[72], lo que proporcionaba como valor añadido un sistema de copia de seguridad, así como poder realizar el desarrollo distribuido desde varios equipos, cuando fuera necesario.

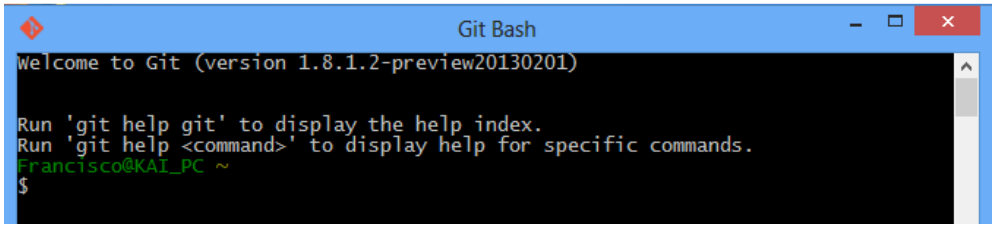


Figura 9-1: La consola del Git

Git es una herramienta que funciona en modo consola. Para mayor comodidad es común instalar algún interfaz[73] avanzado que facilite la interacción con el repositorio. La decisión que se tomó en este proyecto fue instalar el SmartGit, ya que es muy completa y gratuita para uso no comercial.

9.2.1.1. GIT

Git es un sistema de repositorio que presenta algunas características que lo hacer muy interesante:

- **Sistema de 'Branching and Merging':** En cualquier momento, si en necesario, se puede crear una nueva rama sobre el desarrollo principal, probar en ella las modificaciones de código oportunas, y si no te interesa, volver al desarrollo base en el punto que se creó la rama, o, si está todo correcto, fusionar la misma.

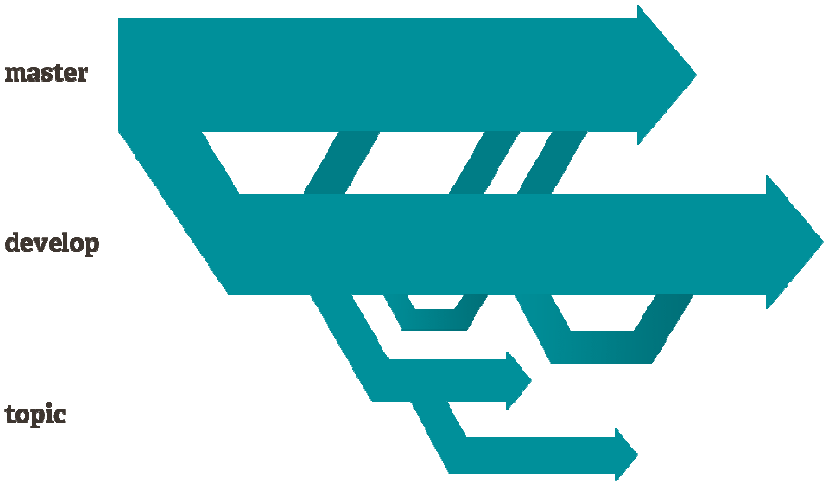


Figura 9-2: Sistema de 'Branching & Merging' del Git
Fuente: Git

- **Ligero y rápido:** El 'core' de git es una aplicación ligera programada en C que gestiona un repositorio local en disco, un solo fichero llamado .git que almacena de forma comprimida el estado original del directorio y las copias diferenciales con todos los cambios que se van produciendo en el mismo. El fichero se ubica en el mismo directorio origen del repositorio, lo que facilita la eventual copia de seguridad o borrado del mismo. Los cambios son almacenados localmente y no se envían hacia el repositorio central hasta que no se ejecute la orden de 'push'.

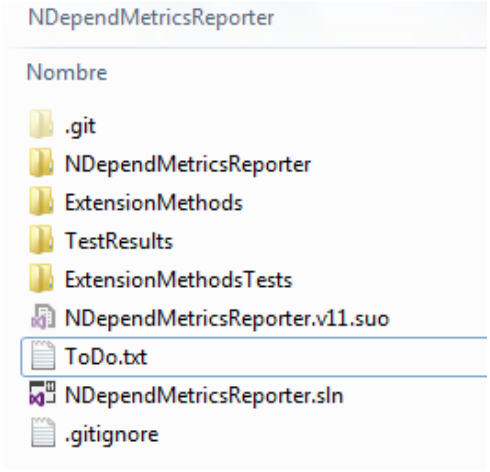


Figura 9-3: La carpeta comprimida .git conteniendo el repositorio del directorio

- **Trabajo distribuido:** Varias personas pueden participar de un mismo proyecto manteniendo una copia local del mismo y volcando sus modificaciones sobre el repositorio central cuando lo estimen necesario. Permite numerosas variantes a la hora de flujos de trabajo, con repositorios principales y secundarios, gestores de integración… pero, en nuestro caso, usamos el sistema más sencillo.

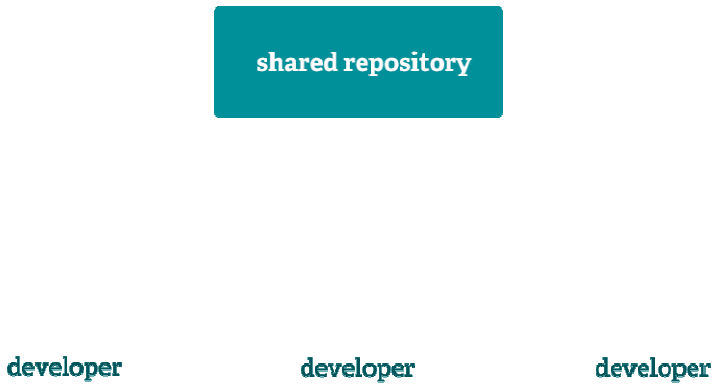


Figura 9-4: Sistema de repositorio compartido en Git
Fuente: Git

- **Seguridad de datos:** Cada aportación al repositorio está firmada criptográficamente con la ID del desarrollador que la realiza, por lo que, es fácil crear un sistema de autorizaciones de lectura/modificaciones, y hacer seguimiento de cada modificación.

- **El 'Staging Area':** Cuando defines un repositorio, estableces, mediante un fichero `.gitignore`[74], de qué ficheros deseas hacer seguimiento y de cuales no. Pero, además, antes de efectuar el 'commit' definitivo sobre el repositorio, siempre puedes revisar, en el 'Staging Area', los ficheros modificados que van a ser almacenados. Por tanto, cada aportación al repositorio se hace en dos pasos:

primero, 'add' al 'stage', y después, 'commit' al repositorio. Para cada 'commit' realizado podrás incluir un comentario descriptivo de los cambios añadidos.

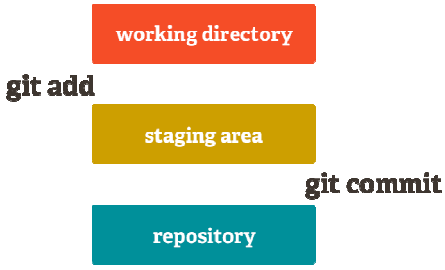


Figura 9-5: El 'staging area' en el Git
Fuente: Git

- **Libre bajo licencia GNU:** Obviamente, no se puede usar software ilegal en un proyecto ;)

Mediante el uso de un sistema repositorio de estas características, y a través de los mensajes del 'commit', se puede llevar un control de las iteraciones, historias de usuario y escenarios, test añadido y puntos del ciclo donde nos encontramos.

Una buena práctica a seguir es realizar un 'commit' después de cada 'green' y al final de cada refactorización.

Instalando el Git

La instalación en Windows del Git (msysgit[75]), es muy simple, como en casi todas las aplicaciones Windows. Tan solo nos pedirá que tomemos algunas decisiones sobre modificaciones de la variable 'PATH' o la forma de manejar CR/LF para mayor compatibilidad son sistemas Unix.

Una vez instalado, dispondremos de una consola estilo 'bash', llamada Git Bash y de un sencillo interfaz, el Git GUI, que permitirá realizar las tareas básicas de crear, clonar y abrir repositorios, o tener acceso a las claves SSH para la conexión con escritorios remotos.

Configurando el Git

Una vez instalado, deberemos configurar el usuario por defecto en Git, para que se identifiquen correctamente los 'commit' realizado a los repositorios. Para ello, abrimos el Git Bash e introducimos estos dos comandos:

```
$ git config --global user.name "Your Name Here"
# Sets the default name for git to use when you commit

$ git config --global user.email "your_email@example.com"
# Sets the default email for git to use when you commit
```

Figura 9-6: Configuración del usuario del Git

Creando las claves SSH para los repositorios remotos

Git utiliza conexión segura SSH para conectarse con los servidores remotos. En nuestro caso, que no trabajaremos solamente en local, sino que haremos uso de GitHub, necesitamos crear nuestro par de claves privada y pública. Para ello, en Windows, podemos hacer uso de la consola de Git Bash o usar el cómodo Git GUI

Generación del par de claves en Git Bash

Ejecutamos la consola de Git Bash e introducimos los siguientes comandos:

```
$ ssh-keygen -t rsa -C "your_email@example.com"
# Creates a new ssh key, using the provided email as a label
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/you/.ssh/id_rsa): [Press
enter]

$ ssh-add id_rsa

Enter passphrase (empty for no passphrase): [Type a passphrase]
Enter same passphrase again: [Type passphrase again]

Your identification has been saved in /c/Users/you/.ssh/id_rsa.
Your public key has been saved in /c/Users/you/.ssh/id_rsa.pub.
The key fingerprint is:
01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db your_email@example.com
```

Figura 9-7: Creación de las claves SSH para la comunicación segura con repositorios remotos

9.2.1.2. GitHUB

La razón principal por la que nos decidimos a usar el Git fue por le existencia del proyecto GitHUB, un sistema de repositorio en la nube que es libre para código abierto. Dado que el proyecto que ibamos a realizar no iba a dar como resultado ningún software comercial, sino que tan solo se trataba de algunas librerías sobre las que realizar estudios de métricas, esta solución era perfecta. Más tarde empezamos con el desarrollo de la aplicación NDepend Metrics Reporter, pero no tuvimos ningún problema en dejar libre su acceso de lectura.

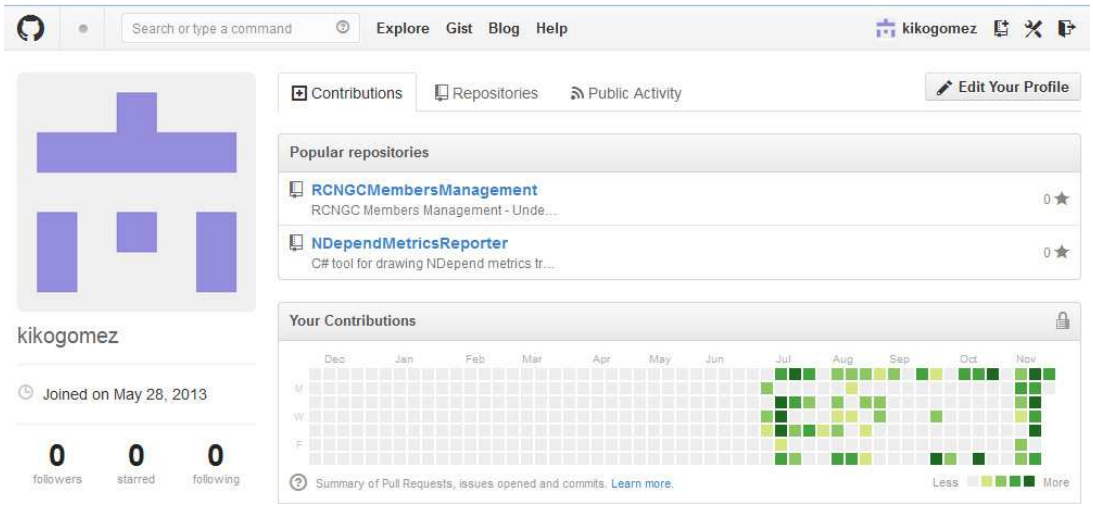


Figura 9-8: Interfaz Web del GitHub

Mediante un simple registro en la página del GitHub y la creación de una cuenta, podemos crear repositorios de forma gratuita siempre y cuando se mantengan públicos. La idea de GitHub público es la de compartir información de manera que cualquier usuario se puede suscribir a repositorios de otras personas para seguirlos, solicitar permiso para unirse a ellos, o clonarlos para iniciar un desarrollo propio basado en el anterior. Es un concepto que trata de comunidad de desarrolladores que comparte experiencias de programación, convirtiendose en una red social con 'seguidores' como puderan tenerlos los 'tweets'[76]. Muchos proyectos de código abierto se está iniciando aquí en vez de en 'SourceForge'[77].

En nuestro caso apenas se ha hecho uso de las capacidades de la red social, sino que se le ha dado tres usos muy definidos:

- Como sistema de copia de seguridad en la nube
- Para compartir el proyecto entre varios equipos donde he realizado el desarrollo
- Para mantener informado al tutor sobre la evolución del proyecto.

El GitHub dispone de una herramienta instalable en Windows para manejar todos los repositorios de forma remota y mantenerse informado de los cambios en el proyecto que realizan otros miembros del equipo, pero el interfaz web ha sido más que suficiente para nuestras necesidades.

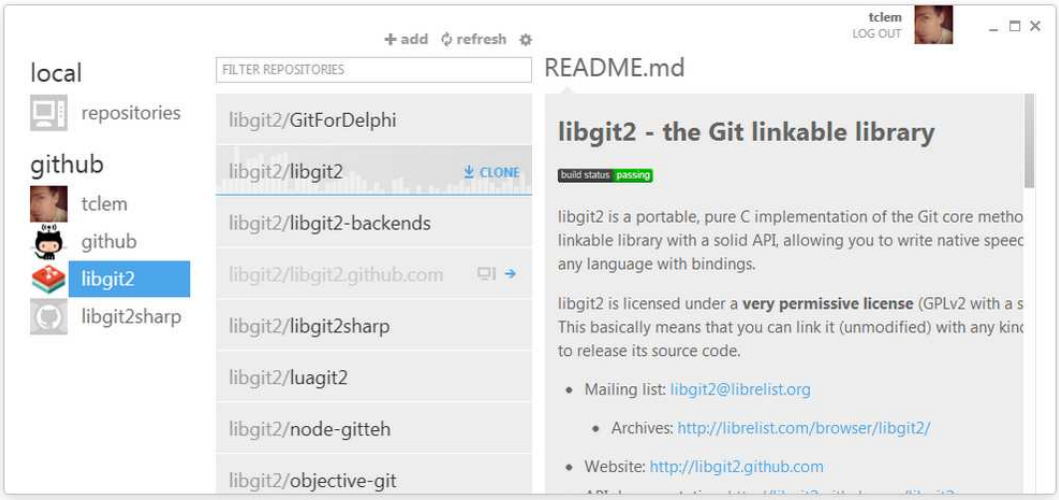


Figura 9-9: Aplicación de acceso a GitHub para Wndows

Configurando la conexión remota a GitHub

Lo primero es localizar las claves pública y privada del Git. Por defecto se encontrarán en el directorio "`C:/Users/<username>/.ssh/id_rsa.pub`". Si es así, Git GUI podrá localizarlas fácilmente, en Help->Show SSh Key.

Figura 9-10: Clave SSH del usuario por defecto Git

Si la clave no estuviera generada, mediante la simple pulsación del botón 'Generate Key' crearíamos una.

Una vez la tengamos en el portapapeles, pasamos entramos en nuestra cuenta de GitHub y seguimos estos pasos:

1. Entramos en 'Account Settings'
2. Pulsamos en "SSH Keys", en la barra de la izquierda
3. Hacemos click en 'Add SSH Key'

4. Pegamos la clave que teníamos en el portapapeles dentro del cuadro 'Key'
5. Pulsamos 'Add Key'
6. Confirmamos introduciendo nuestra clave de usuario

Todo el procedimiento de creación de claves, introducción de las mismas y prueba de conexión la podemos encontrar en este enlace:

https://help.github.com/articles/generating-ssh-keys

9.2.1.3. SmartGit

SmartGit es una herramienta realmente potente que permite realizar todas las operaciones sobre tu repositorio 'Git' de una manera visual y muy sencilla. Incluye un completo visualizador de las modificaciones realizadas sobre el código, un formidable gestor de 'branches', y un sistema de conexión con repositorios en la nube que hace muy fácil la sincronización. Además, añade un editor del 'index', la memoria que registra los cambios que se van produciendo en los fichero antes de ser añadidos al 'stage', permitiendo su modificación.

Instalando SmartGit

La instalación del SmartGit es inmediata, a través de un sencillo 'wizard', en el que no deberemos tomar prácticamente ninguna decisión.

Ventana principal

Desde la interfaz del SmartGit tenemos rápido acceso a las operaciones más comunes del Git:

- Stage, Unstage, Index editor: Gestiona los elementos de 'stage'
- Commit/Merge: Envía del stage al repositorio, y fusiona la rama actual del proyecto con la rama principal si se lo indicamos
- Pull/Push/Sync: Realiza las operaciones entre el repositorio local y la red

En la pantalla principal del SmartGit tenemos, en un solo vistazo, las carpetas que componen el repositorio y los ficheros que han sido modificados, así como su estado (si se encuntran pendientes de añadir al 'stage' o ya han sido añadidos para el próximo 'commit'). A través de diferentes filtros puedes pedir al programa que te presente todos los ficheros.

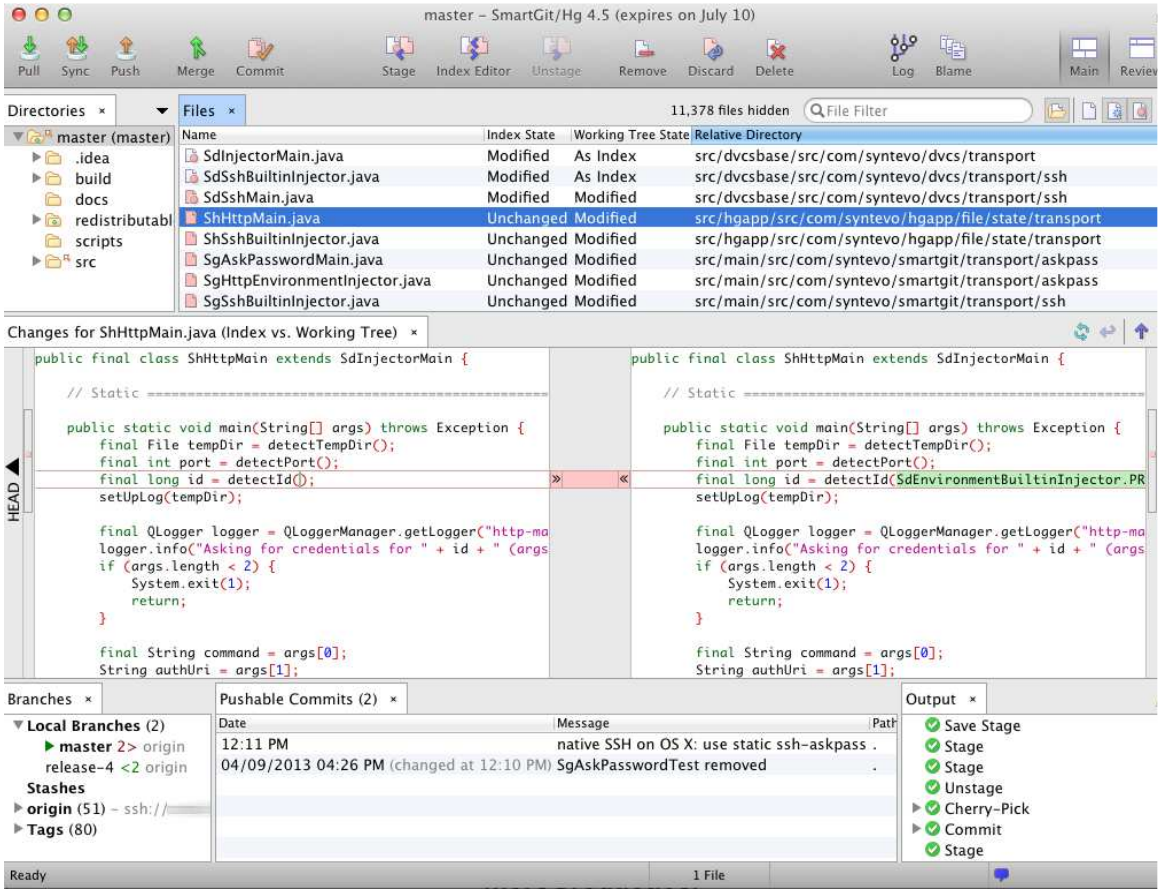


Figura 9-11: La ventana principal del SmartGit

También dispone de un visor en el que se presentan cuáles son las modificaciones producidas en los ficheros de texto, en nuestro caso, ficheros fuente, para un mejor seguimiento de la evaluación de código

Finalmente, presenta un resumen del las ramas del proyecto, operaciones más recientes realizadas sobre el proyecto, y últimos 'commits'.

Seguimiento de la evolución del repositorio

Desde esta ventana podemos acceder de manera muy visual al estado del repositorio en cualquier punto temporal, con una representación gráfica muy clara de la evolución de las diferentes ramas.

Permite llevar el control de cada rama, y revisar los cambios introducidos en cada 'commit'. Si el sistema de repositorio sobre el que corre la herramienta permite la creación y edición de perfiles completos de usuario, estos aparecen en los detalles.

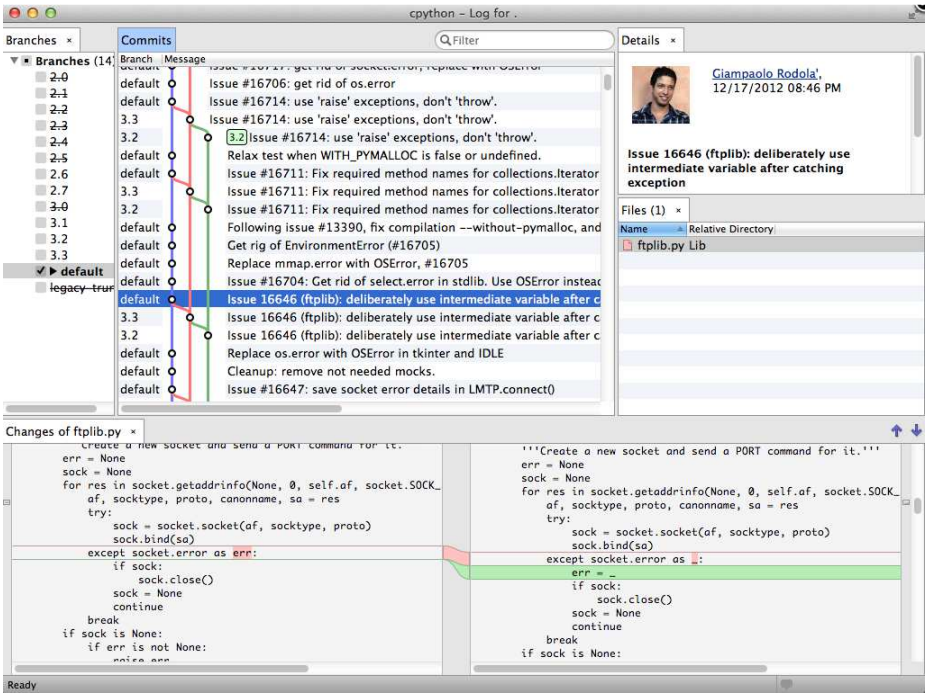


Figura 9-12: Seguimiento de la evolución del repositorio en SmartGit

Editor de cambios en índice

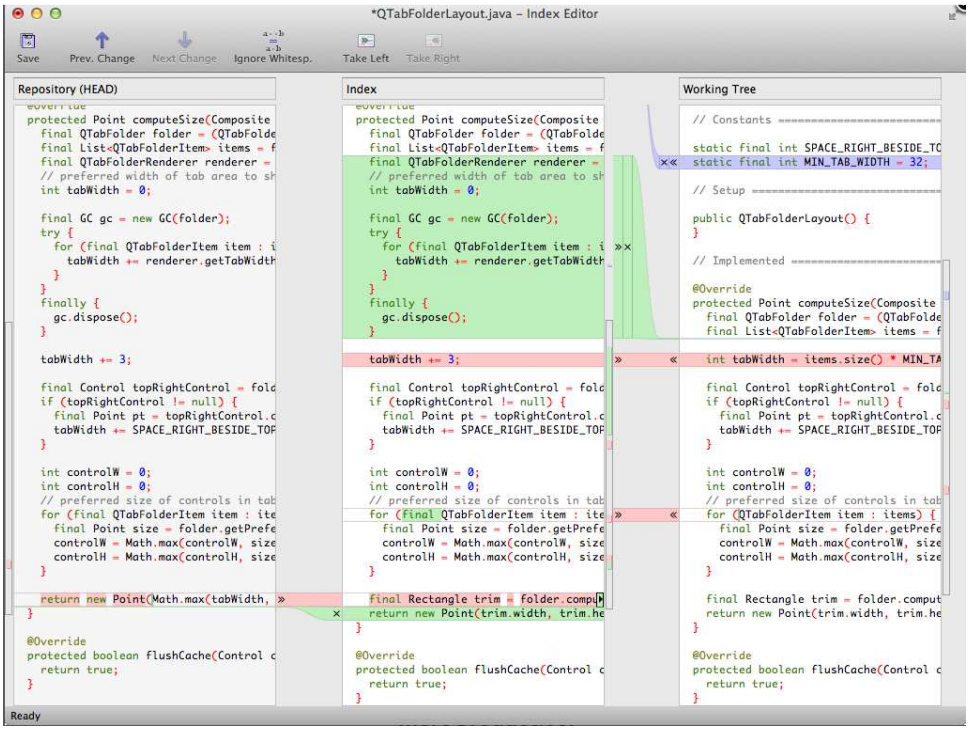


Figura 9-13: El 'Index Editor' del SmartGit

Desde el 'index editor' puedes ver claramente, para cada fichero, su estado en el repositorio, el fichero tal y como se encuentra actualmente en el directorio de trabajo, y los cambios (adiciones y elementos eliminados) que se van a producir en el stage, permitiendo una edición de los mismos.

Registro de modificaciones en un fichero: Blame

Una de las herramientas más potentes del Git es el 'Blame'. En él podemos ver cada cambio producido en el cada línea de código de un fichero, la fecha del mismo y quién fue el desarrollador que la hizo, de ahí el simpático nombre de 'Blame' (¡ya sabemos a quien echarle la culpa si algo falla!). SmartGit dispone de un visualizador para esta función.

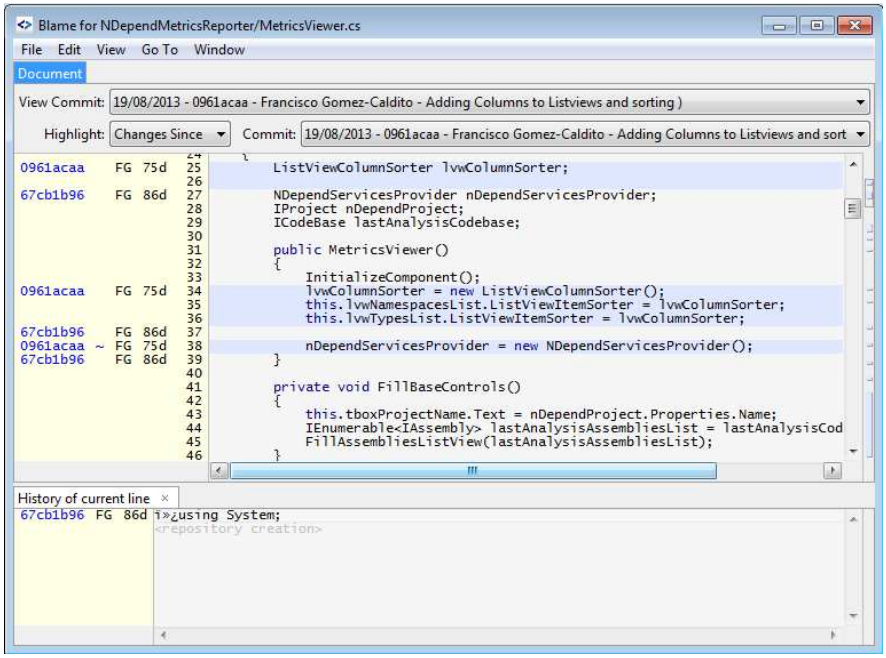


Figura 9-14: Blame para SmarttGit

Gestión de repositorios en la red

El SmartGit permite vincular cualquier repositorio local con otro remoto de manera sencilla. Para conectarse con un repositorio GitHub tan solo es necesario iniciar un proceso de clonado en local. La conexión con GitHub se realiza a través de SSH, debiendo cargarse en tanto en SmartGit como en GitHub la clave de la ID de usuario Git. Como medida adicional, en las operaciones 'push' se pedirá también confirmación de la clave de la cuenta GitHub.

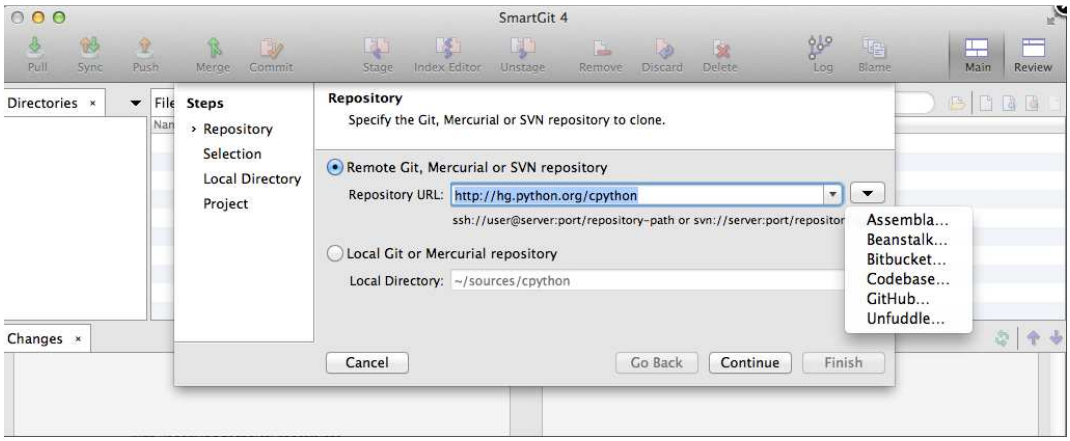


Figura 9-15: Conexión de SmartGit con repositorios remotos

Las claves SSH para la conexión son encontradas y añadidas automáticamente al SmartGit durante la instalación del mismo. Si necesitamos cambiarlas, podemos hacerlo a través de *Edit-> Preferences*

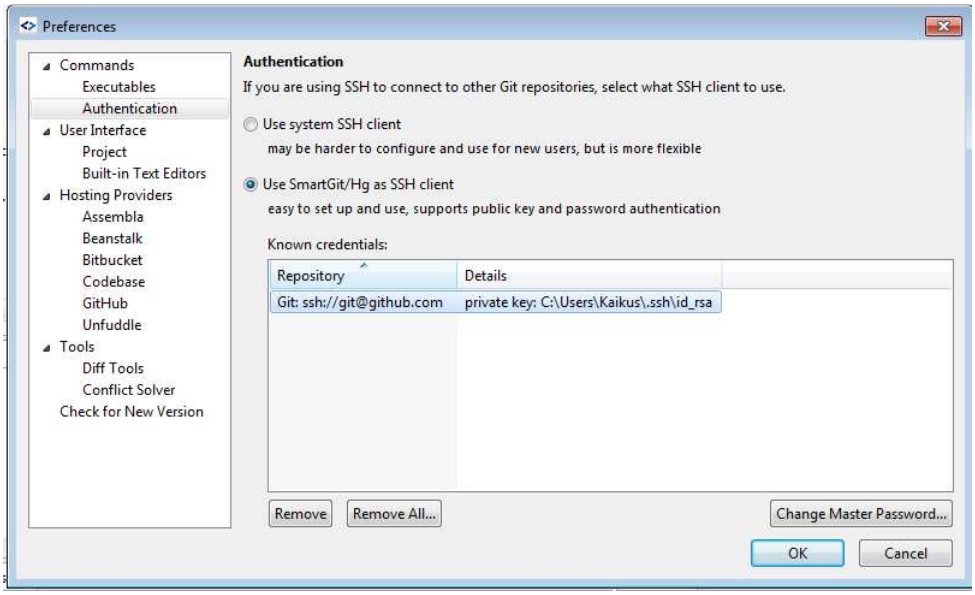


Figura 9-16: Gestión del claves SSH en SmartGit

9.2.2. SpecFlow

Para hacer más sencilla la gestión y el seguimiento de las historias de usuario BDD existe una serie de frameworks que facilitan la integración del BDD y la automatización de sus test de aceptación dentro del entorno del desarrollo.

Uno de los más conocidos es el Cucumber[78] para lenguaje de programación Ruby[79]

Para el entorno de desarrollo de Visual Studio podemos encontrar el SpecFlow[80], de código abierto bajo licencia BSD, basado en la misma idea que Cucumber y que también utiliza como lenguaje Gherkin[81], y que presenta una magnífica integración con la mayoría de frameworks de test para Visual, incluyendo el nativo MSTests, que es el que hemos usado en nuestro desarrollo.

Para completar el Specflow existen dos herramientas que, si bien no hemos usado por no se libres, merecen especial mención:

- SpecRun[82] mejora la interfaz para la ejecución y el seguimiento de los test de aceptación de SpecFlow.
- SpecLog[83] es la herramienta perfecta para organizar y gestionar todas las historias de usuario entre los distintos miembros de un equipo de desarrollo ágil, hacer un seguimiento de las mismas, y documentarlas.

Con SpecFlow, la conversión de las historias de usuario a test de aceptación es un proceso muy simple en dos pasos:

1.- Especificar la historia de usuario en lenguaje Gherkin

Para cada historia de usuario creamos una 'Feature' en Specflow

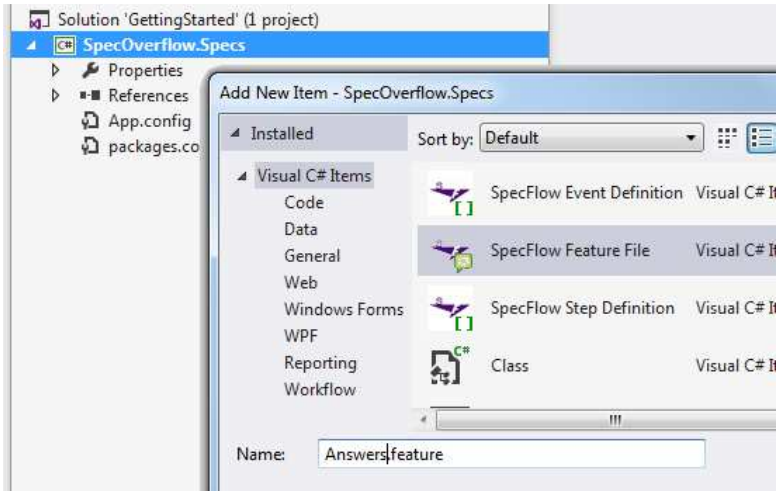


Figura 9-17: Añadiendo una nueva 'Feature' (Historia de Usuario)

Una 'feature' no es más que un documento de texto con una historia de usuario como la que ya hemos mostrado al hablar de BDD, con un formato específico:

- La historia de usuario se llama 'Feature'
- A la hora de especificar 'role-feature-benefit' se usa prácticamente la misma notación que la que presentó Dan North, solo que cambiando un poco el orden. En este caso es 'In order to – As an – I want to', es decir, 'benefit-role-feature'

- Seguidamente, vienen los diferentes escenarios, con su serie de precondiciones (Given), acciones (When) y resultados (Then)

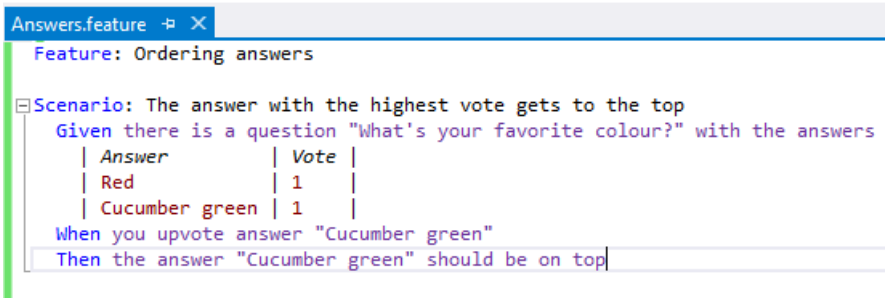


Figura 9-18: Una 'feature' o historia de usuario en SpecFlow

2.- Automatizar los escenarios

Podemos pedir a SpecFlow que nos cree y asocie un método a cada uno de los pasos Given-When-Then de la 'feature'. Cada uno de estos métodos es llamado por SpeFlow un 'Step Definition'. En vez de tener un solo test de aceptación, nosotros lo veremos dividido en fragmentos Given-When-Then, que iremos completando con el código de nuestro test de aceptación. En el momento de la compilación, SpecFlow reunirá todos los 'step definitions' asociados a los Given-Then_When de un escenario en solo test con su nombre, cuyo resultado veremos posteriormente al ejecutarlo desde nuestro framework de test.

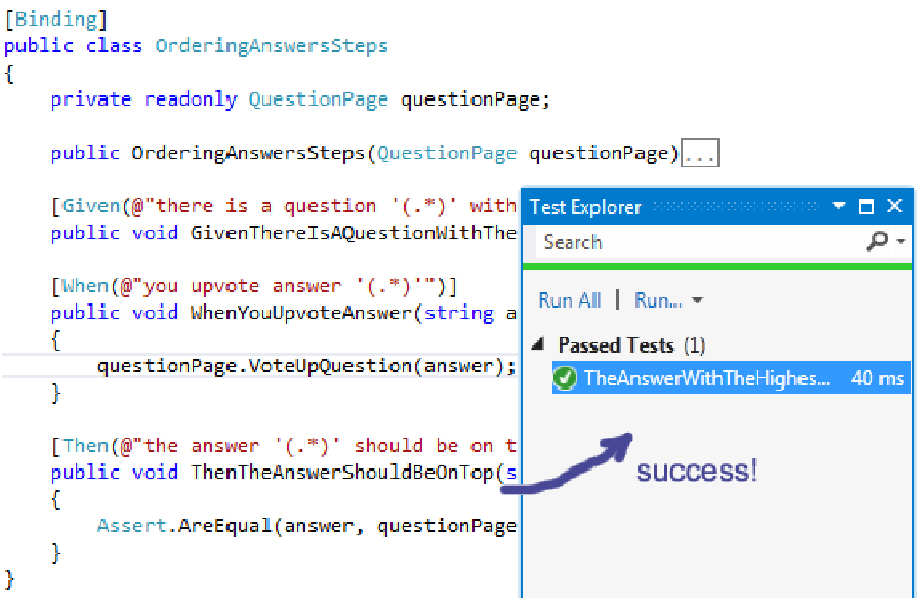


Figura 9-19: Specs binding en Specflow

Instalando el SpecFlow

El Framework de SpecFlow está compuesto de tres paquetes principales:

- El **integrador del IDE**, que customiza el editor y las funciones de generación de test en el entorno de desarrollo elegido
- El **generador**, que convierte las especificaciones Gherkin en clases de test ejecutables
- El **runtime**, que es necesario para ejecutar los test generados. Hay diferentes ensamblados según la plataforma de destino (.Net, Silverlight, Windows Phone)

En el caso de nuestro entorno de desarrollo, todo ello se automatiza a través de una sola descarga e instalación , a través de la herramienta de 'Extensiones y Actualizaciones' del Visual Studio.

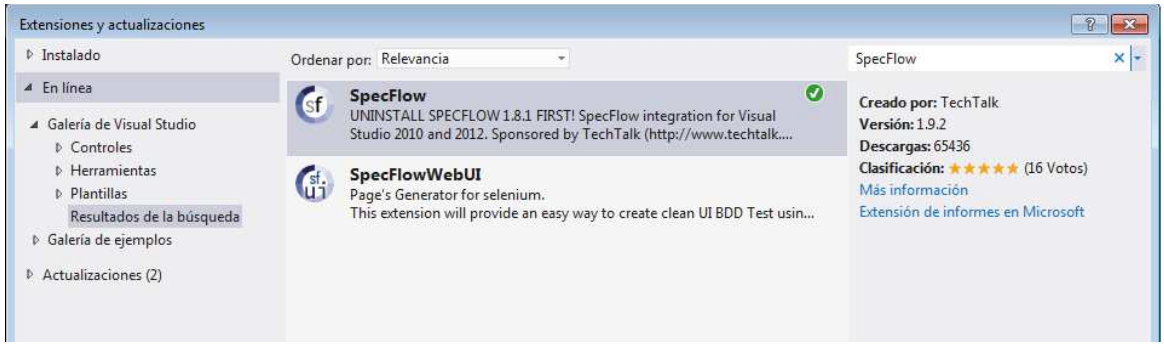


Figura 9-20: Instalación de SpecFlow en Visual Studio 2012

Una vez ya dispongamos de la herramienta integrada en nuestro entorno de desarrollo, podemos añadir SpecFlow a nuestra solución activa.

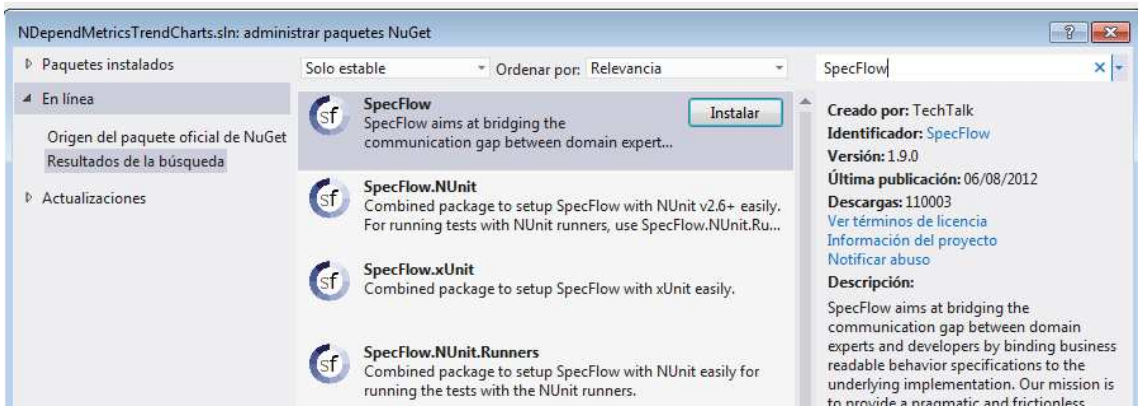


Figura 9-21: Añadiendo un proyecto SpecFlow a la solución a través del administrador de paquetes NuGet

Aunque no es imprescindible, una buena praxis profesional requiere que todo el sistema de BDD se monte en un proyecto de biblioteca de clases separado de la lógica de

aplicación. Una vez creado, añadimos SpecFlow través de la herramienta de administración de paquetes llamada 'NuGet'[84], incluida en Visual Studio 2012.

Configurando el proyecto SpecFlow

La configuración del proyecto SpecFlow se realiza por completo a través de la edición de un fichero .XML, llamado *'App.config'*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="specFlow"
      type="TechTalk.SpecFlow.Configuration.ConfigurationSectionHandler,
TechTalk.SpecFlow"/>
  </configSections>
  <specFlow>
    <unitTestProvider name="MsTest" />
  </specFlow>
</configuration>
```

Figura 9-22: Fichero de configuración de SpecFlow

De las múltiples opciones de configuración[85] la que realmente nos interesa es especificar cual será el proveedor de tests. En nuestro caso indicaremos que será MsTest, que esta integrado en Visual Studio. Si fuéramos a usar otro proveedor diferente, deberíamos, primeramente, instalar dicho Framework de test, y, posteriormente, indicar a SpecFlow que vamos a hacer uso del mismo.

Finalmente, no se nos debe olvidar añadir en nuestro recién creado proyecto, una referencia al ensamblado sobre el que ejecutaremos los test de aceptación.

Gherkin: el lenguaje para expresar historias de usuario

```
1: Feature: Some terse yet descriptive text of what is desired
 2:   In order to realize a named business value
 3:   As an explicit system actor
 4:   I want to gain some beneficial outcome which furthers the goal
 5:
 6:   Scenario: Some determinable business situation
 7:     Given some precondition
 8:       And some other precondition
 9:      When some action by the actor
10:       And some other action
11:       And yet another action
12:      Then some testable outcome is achieved
13:       And something else we can check happens too
14:
15:   Scenario: A different situation
16:       ...
```

Figura 9-23: Una historia de usuario expresada en Gherkin

Gherkin es el lenguaje que utiliza SpecFlow. Está diseñado para ser similar al lenguaje natural y ayuda a describir el comportamiento de un software sin detallar cómo se implementa dicho comportamiento. Establecer un nexo entre lo que el cliente necesita y una especificación más o menos detallada de los elementos de un test de aceptación. Cumple, por tanto fines: documentación y automatización de tests

En Gherkin cada declaración es una línea, y la conversión desde las historias de usuario de Dan Noh es trivial: un nombre para la historia, la especificación de role-feature-benefit, y una serie de escenarios, cada uno de los cuales se convertirá en un test de aceptación.

Al ser un lenguaje natural, con pocas palabras reservadas (en realidad, 12: name, native, feature, background, scenario, scenario_outline, examples, given, when, then, and, but), Gherkin se puede obtener en multitud de idiomas, para facilitar la interpretación al cliente.

Las especificaciones completas del lenguaje[86], incluyendo como añadir elementos de contexto y tablas de ejemplos, las podemos encontrar dentro de proyecto Cucumber que mantiene GitHub

'Feature'-'Scenario'-'Step': la automatización de los test SpecFlow

Las 'features' son la base del Specflow. Cada una de ellas se especifica en lenguaje Gherkin en un fichero de texto `.feature` diferente. Cada fichero `.feature` incluye una serie es escenarios, cada uno de los cuales se detalla en una serie de pasos (steps) Given-When-Then (precondiciones-acciones-resultados). Hasta aquí la parte que verá el cliente.

Para el desarollador, cada una de estos pasos deberá ser cubierto por un 'step definition', un método que se asocia al 'step', y que implementa, en el lenguaje del entorno de programación, el comportamiento del mismo. Los declaraciones de los 'step definitions' pueden ser creadas y asociadas por el desarrollador o dejar que sea el SpecFlow quien se encargue de la labor, dejando al programador la tarea de implementarlos. La unión de todos los 'step definitions' asociados a los pasos Given-Then-When de un escenario conformarían lo que sería el test de aceptación de dicho escenario.

¿Pero, cómo funciona?

Tras la compilación, SpecFlow generará de forma automática una clase (`.feature.cs`) por cada `.feature` del proyecto. En ella se creará un test por cada escenario, con el nombre del mismo. El contenido de esta clase no debe ser en ningún momento editado por el desarrollador, ya que es autogenererado por el framework. Si la visualizamos, podremos observar que incluye varias inicializaciones y los tests que luego veremos ejecutarse (en formato NUnit, XUnit, MsTest…, según la configuración del proyecto). Sin embargo, el contenido de estos test no es el código reunificado de los diferentes 'specs', sino que todo el trabajo se realiza a través de su propio módulo 'runtime', mediante llamadas a un objeto *testRunner* que va invocando por su nombre los diferentes 'specs' y ejecutándolos a través de reflexión.

Esto, que en un principio es un comportamiento perfecto para evitar duplicidades en el código, supuso un problema a la hora de realizar métricas sobre tests de aceptación, como luego explicaremos, obligándonos a tomar medidas 'antipatrón'.

Steps Binding

SpecFlow facilita la creación y asociación de los métodos 'step definition', con un patrón muy simple:

Dado el 'step'

```
Given I have an invoice
```

El método 'step definition' que se asociará al mismo tendrá esta forma

```
[Given(@"I have an invoice")]
public void GivenIHaveAnInvoice(){}
```

Figura 9-24: Forma en que SpecFlow asocia cada 'paso' con el código que lo implementa

Los 'step definitions' se agrupan en clases que han de estar decoradas con el atributo `[Binding].` El nombre del método es irrelevante, aunque resulta mejor si hace referencia al 'step'.

Lo más importante a recordar es que en SpecFlow los 'step definition' son, por defecto, GLOBALES al proyecto. Por tanto un método decorado con '`[Given(@"I have an invoice")]`' se asociará a cualquier '`Given I have an invoice`', independientemente de en qué fichero `.feature` se encuentre dicho step, o si se repite en varios escenarios diferentes. Tampoco importa en que `[Binding]` hayamos incluido el 'step definition'. Esto, que en un principio es una enorme ventaja a la hora de ahorrar código (un solo 'step definition' nos sirve para muchas repeticiones de un step), puede llegar a ser un verdadero rompecabezas, pues:

- En dos escenarios diferentes dos pasos con el mismo nombre pueden referirse a situaciones distintas y requerir cada una su propia implementación.
- Igualmente, no podemos duplicar steps definitions.

Para evitar este tipo de problemas, SpecFlow nos da una serie de consejos y nos facilita algunas herramientas:

- A la hora de crear los métodos 'step definitions' no debemos agruparlos en torno a una 'feature', sino a un dominio. Es decir, que no debemos intentar crear una clase con todos los 'step definitions' asociados a una 'feature 'en particular, ya que posiblemente terminaremos asociando esos mismos métodos a otros pasos con el mismo nombre y función en diferentes 'features'. Lo mejor es utilizar el dominio de negocio al que se aplican como norma para agrupar los 'step definitions'

- Esto nos lleva a que, a veces, un escenario se desarrolla a través de métodos que residen en clases (`[Binding]`) distintas. SpecFlow facilita algunos métodos para intercambiar[87] la información, entre las que se incluyen inyección de clases POCO[88], campos de instancia de clases, incluso a veces recomiendo el uso de clases estáticas para el intercambio. También facilita una clase llamada ScenarioContext, que se inicializa al principio de cada escenario, a través de la cual, utilizando un diccionario llamado ScenarioContext.Current[89], podemos pasar objetos de todo tipo a lo largo de la ejecución.

- Si nos vemos en la necesidad de duplicar 'step definitions' restringir el ámbito de los mismos mediante el uso de los 'Scooped Bindings'[90]. Siempre podemos decorar una 'feature', 'scenario' o 'step' mediante el uso de 'tags'[91], lo que normalmente hacemos para filtrar u organizar escenarios y features. Es posible establecer el alcance de un 'Step Definition' si especificamos su 'scope'. Sin embargo SpecFlow es muy claro al respecto, indicando que se debe evitar el uso de estas prácticas por considerarlas 'antipatrón'

```
[Scope(Tag = "mytag", Feature = "feature title", Scenario = "scenario title")]
```

  Sin embargo, en este desarrollo en particular, para poder adquirir métricas sobre las historias de usuario, a la hora de crear los 'Step Definition' para SpecFlow hemos seguido un comportamiento 'antipatron'. A veces hemos debido duplicarlos para reunir en una sola clase todos los todos los métodos asociados a los 'Steps' de los escenarios de un mismo 'Feature'. El problema reside en la manera que tiene SpecFlow de implementar los test de aceptación para cada escenario, ya que, en vez de llamar directamente a los métodos de los 'Step Definition', lo hace a través de su propio runtime, usando reflexión. Por tanto, las métricas efectuadas sobre dichos test de aceptación no arrojan ningún tipo de información sobre el modelo de la aplicación. Solo a través de esa 'clase antipatron', que reúne todos los métodos que procesan un 'Feature', podemos tomar alguna medida con respecto a la complejidad del mismo.

Estas son las características más básicas de SpecFlow y Cucumber. Para profundizar en mayor medida sobre sus funciones, podemos acceder a una documentación[92] muy completa en la página del desarrollador.

9.2.3. MSTests

Para nuestro desarrollo TDD hemos elegido el MSTest, el framework propio que trae integrado el Visual Studio

Su funcionamiento es similar al de otras infraestructuras de test más conocidas en sistemas abiertos como pudieran ser NUnit o XUnit. Sin embargo, dado que estamos desarrollando en Visual, hemos querido utilizar la herramienta nativa y comprobar su integración con las otros elementos como son SpecFlow, TestDriven o NCover.

Desfortunadamente, algunas características importantes, como pueden ser el Code Coverage, no está disponibles en la versión Profesional de Visual Studio, que es la que la

escuela pone a disposición de los alumnos para su descarga través de DreamSpark, por la que, para poder hacer seguimiento del mismo, instalamos la herramienta TestDriven, que incluye una versión limitada de NCover, suficiente para este propósito.

Agregando un proyecto MSTest

Al ser el Framework nativo de Visual Studio, agregar un proyecto de test es de lo más sencillo. Simplemente, como vemos en la Figura 9-25 abrimos en interfaz para añadir un proyecto más a la solución, y elegimos “Proyecto de prueba unitaria”

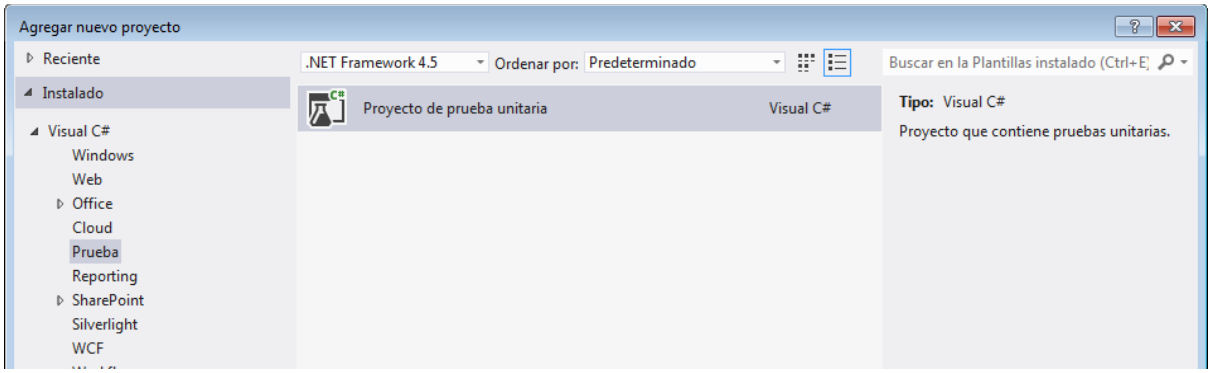


Figura 9-25: Agregando un proyecto MSTest a la solución actual

Al igual que con el SpecFlow, tan solo nos falta añadir la referencia al ensamblado que contiene el código sobre el que ejecutaremos los test unitarios.

La clase de test en MSTest y su instanciación

Una clase de test es, básicamente, una clase normal de C# en la que ha sido decorada con atributos que la identifican como tal. Dentro de la misma, cada uno de los métodos que la componen puede ser decorado con un atributo para realizar una función específica, siendo la más normal la de `[TestMethod]` que identifica al mismo como test.

A diferencia de otras suites de test, como puede ser NUnit, en MStest cada uno de los `[TestMethod]` se ejecuta en una instancia separada de la clase, cada uno de ellos en un hilo diferente. Este diseño implica 4 comportamientos que han de tenerse en cuenta:

1. **ClassInitialize y ClassCleanup:** Microsoft ha diseñado estos métodos como estáticos para ser ejecutados una sola vez. ClassInitialize antes de la primera instancia del primer `[TestMethod]`, y `[ClassCleanup]` lo hace al final de la última instancia. Su función es inicializar varuiables que deberán ser compartidas por los test. Al ser una clase estática, solo puede modificar variables estáticas.

2. **Orden de ejecución:** El orden en que se ejecutan los tests es absolutamente aleatorio, por lo que el resultado de un test no deberá afectar a los demás. Es por

ello que las variables estáticas inicializadas en `[ClassInitialize]` no deberían ser tocadas. Si necesitamos inicializar un grupo de variables, un contexto, que usaremos un modificaremos en cada test, la manera correcta de hacerlo es a través de los métodos `[TestInitialize]`, que se ejecutarán una vez por cada `[TestMethod]`. Del mismo modo existen los `[TestCleanup]`, que se ejecutarán después de cada `[TestMethod]` .

```
[TestClass]
public class VSTSClass1
{
    private TestContext testContextInstance;

    public VSTSClass1()
    {}

    public TestContext TestContext
    {
        get
        {
            return testContextInstance;
        }
        set
        {
            testContextInstance = value;
        }
    }

    [ClassInitialize]
    public static void ClassSetup(TestContext a){}

    [TestInitialize]
    public void TestInit(){}

    [TestMethod]
    public void Test1(){}

    [TestMethod]
    public void Test2(){}

    [TestMethod]
    public void Test3(){}

    [TestCleanup]
    public void TestCleanUp(){}

    [ClassCleanup]
    public static void ClassCleanUp(){}
}
```

Figura 9-26: Una clase de test en MSTest y sus atributos

3. **El Constructor de la clase:** Como vemos, cada `[TestMethod]` se ejecuta en una instancia separada de la clase, por lo que el constructor de la clase, si lo implementamos, se ejecutará una vez por método, asemejándose su comportamiento a `[TestInitialize]` en vez de a `[ClassInitialize]`

4. **TestContext:** MSTest facilita una clase especial '*TestContext*'[93], cuya forma de ser inicializada vemos en el ejemplo de la Figura 9-26, que podremos usar para obtener información sobre el contexto de los tests. Las variables de este tipo se inicializan en cada test, no pudiendo utilizarse para compartir información entre cada `[TestMethod]`

Estructura de un test en MSTest

La estructura de un método de test es, básicamente, siempre la misma:

1. Incialización de las variables locales del test (precondiciones), donde se carga el sistema con los valores a probar

2. Ejecución del método o llamada al elemento del sistema sobre el que queremos efectuar la prueba

3. Comprobación del resultado mediante el uso de la clase *'Assert'*[94]

```
[TestMethod]
public void NetAmmountisWellCalculatedOnTaxFreeInvoices()
{
    Invoice invoice = new Invoice("11111", new DateTime(2013, 4, 13));
    invoice.AddTransaction("Cuota", "Cuota Social Julio", 1, 79,0);
    Assert.AreEqual(79, invoice.NetAmmount);
}
```

Figura 9-27: Ejemplo de test unitario en MSTest

Manejo de excepciones en MSTests

MSTest gestiona los resultados de los test a base de excepciones[95]. Cada vez que un test falla, se lanza la excepción *'AssertExceptionFailed'*. De esta manera, si un test presenta varias sentencias *'Assert'* (lo que debería evitarse en la medida de lo posible), desde que falle una el framework MSTest se ahorra el tener que seguir con la ejecución del resto. Por tanto, el resultado de un test puede ser:

- **Test en rojo:** Se ha producido una *'AssertFailedException'*. Esto puede ocurrir porque:
    - El test ha tardado demasiado tiempo en ejecutarse
    - El test ha causado una excepción no controlada
    - El test falla
- **Test en amarillo:** Se ha producido una 'AsserInconclusiveException'. Esto ocurre cuando el resultado del test es 'inconclusivo'. Normalmente se hace uso de esta

función para marcar los test de aceptación incompletos, forzando esta excepción mediante la invocación del método '`Assert.Inconclusive()`'

- **Test en verde:** No se ha producido ninguna excepción. El test ha pasado

Por tanto, hemos de tener muy en cuenta dos cosas:

- Si un test finaliza en rojo no siempre significa que el test ha fallado.
- *Si lo que esperamos como resultado de nuestro test es que se produzca una excepción, no podemos comprobarlo a través de una sentencia 'Assert', ya que solo hay verde si no se producen excepciones.*

La manera que tiene MSTest de manejar las excepciones es decorando el método de test con un atributo que refleje el tipo de excepción decorado. Entonces, el propio MSTest interceptará la interrupción, verá si es del tipo esperado, y el test aparecerá en verde. Esta manera de manejar las excepciones obliga a realizar un trabajo 'extra' a la hora de comprobar que nuestra aplicación las lanza correctamente:

1. Decorar el método de test para que se quede a la espera del tipo de excepción que produciría un test con 'éxito'
2. Interceptar nosotros la interrupción dentro de un bloque try-catch.
3. Dentro del bloque catch, comprobar mediante 'Assert' que la excepción interceptada es la que esperábamos, por ejemplo, comparando el mensaje de la excepción con el valor esperado.
4. Lanzar hacia fuera la excepción, para que siga su curso y el framework de test la detecte. Si no la lanzáramos, el test daría 'failed'.

En la Figura 9-28 podemos ver un ejemplo

```
[TestMethod]
[ExpectedException(typeof(System.ArgumentException))]
public void AccounNumberMaxLenghtIs10()
{
     try
     {
         BankAccount testAccount = new BankAccount("", "", "", "1234561234578909");
     }

     catch (System.ArgumentException e)
     {
         Assert.AreEqual("número de cuenta", e.ParamName);
         throw e;
     }
}
```

Figura 9-28: Como comprobar las excepciones en MSTest

9.2.4. NCover

Dado que la versión Professional de Visual Studio 2012 no incluye la función de *Code Coverage*, para poder seguir en porcentaje de código cubierto por nuestros test, hemos tenido que recurrir a aplicaciones de terceros.

Una de las más completas que podemos encontrar para esta función es NCover, que, además, es compatible con NDepend, que luego comentaremos. Sin embargo, NCover es en la actualidad una aplicación comercial que tan solo incluye una versión de prueba de 30 días, absolutamente insuficiente para nuestro proyecto.

Como alternativa, pudimos localizar la herramienta TestDriven.Net[96], una herramienta para gestionar los proyectos TDD, facilitando un cómodo interfaz para la ejecución y seguimiento de los test, que integra una versión antigua de NCover. Tras varios días de prueba, pudimos constatar que esta versión limitada no es compatible con NDepend, por lo que no pudimos integrarla con el resto de métricas de estudio. Sin embargo, dispone de un interfaz en Visual Studio que permite hacer un completo seguimiento de la cobertura del código por los tests.

Instalando TestDriven

La versión Personal de TestDriven.Net es gratuita para entornos de estudiantes. El instalador es fácilmente descargado desde su página web, tras la ejecución de éste, queda instalado como un add-in integrado dentro de Visual Studio

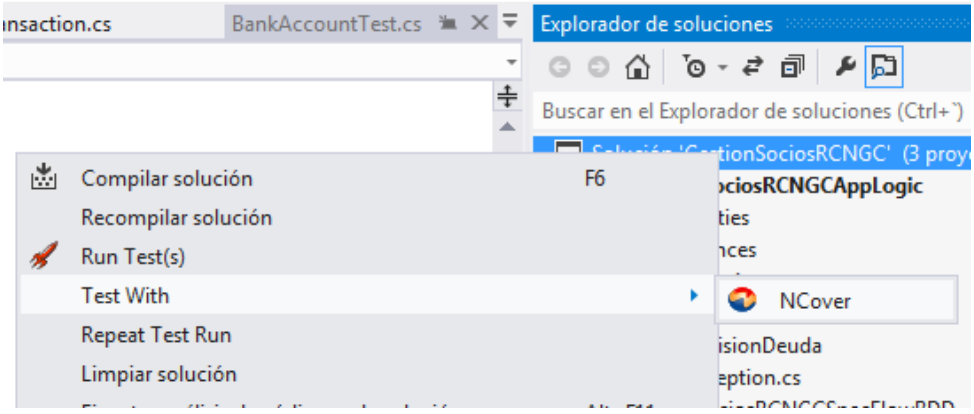


Figura 9-29: TestDriven y NCover integrados den Visual Studio 2013

Usando NCover

Tras lanzar NCover se realiza la comprobación de cobertura de código, y mediante su NCoverExplorer podemos localizar cualquier código no cubierto por los test.

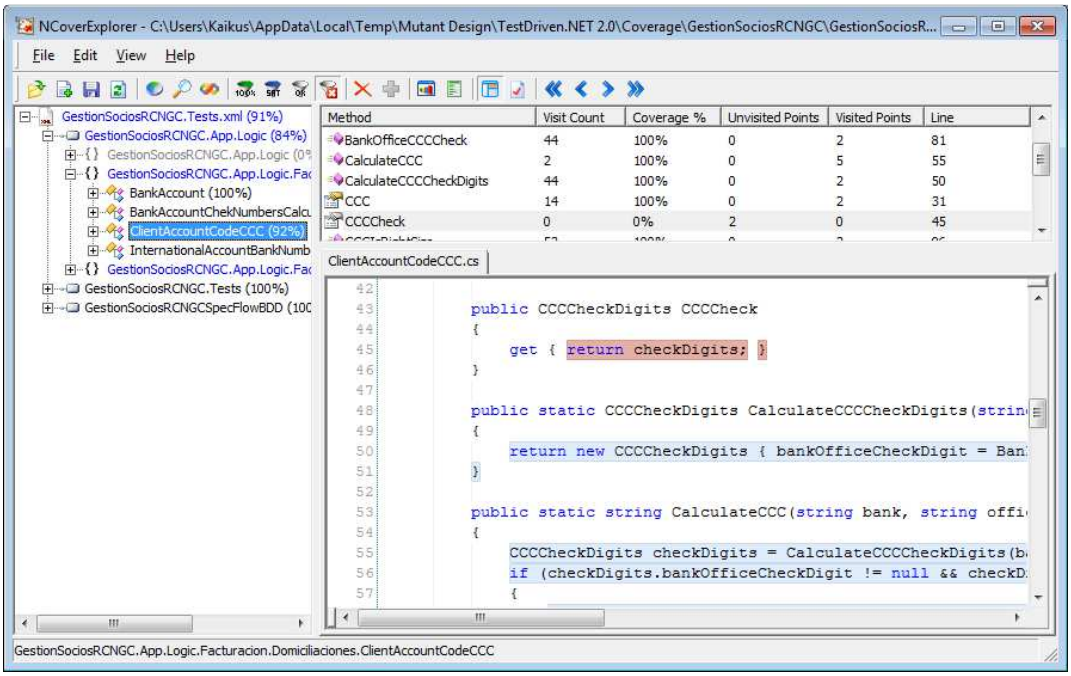


Figura 9-30: La herramienta NCoverExplorer en funcionamiento

NCoverExplorer contiene un cómodo navegador a través del cual, y guiado por códigos de color, podemos acceder a cualquier punto de nuestro código y comprobar si está cubierto por un test.

En la Figura 9-30 podemos encontrar un error muy común al iniciar el desarrollo en TDD, pues al crear una clase que necesitamos usar, podemos cometer el error de completarla con propiedades (para acceder a sus campos privados) que aún no han sido testeados. Al lanzar con regularidad esta herramienta nos aseguramos de mantener un CodeCoverage del 100%.

9.2.5. NDepend

NDepend[97] es una potentísima herramienta profesional que podríamos definir en dos frases que utilizan en su página web:

"Make your .Net Code Beautiful"
"Achieve higher .Net code quality"

NDepend se integra en Visual Studio y permite:

- Monitorizar la evolución del código a través de más de 82 métricas[98]
- Visualizar la arquitectura del software explorando las dependencias entre ensamblados y tipos a través de grafos y matrices de dependencias[99]

- Interrogar al código mediante sentencias LINQ[100]. Dichas sentencias se pueden convertir en reglas de comprobación que lanzan avisos cuando no son cumplidas.

- Desde la versión 5, salida en octubre de este año 2013, y posterior al inicio de este proyecto (que utiliza NDepend 4.5), incluye seguimiento a través de gráficas de tendencias[101].

- Representación de varios diagramas que reflejan la complejidad del código, así como su nivel de abstracción y posible inestabilidad[102].

A lo largo del desarrollo de este proyecto he aprendido a apreciar en todo su valor la tremenda utilidad de esta herramienta en su enorme variedad y complejidad. Sin embargo, desde un principio se definió su uso en este TFG tan solo para hacer el seguimiento y evaluación del código a través de métricas. Fue esta la razón por la que se desarrollo, haciendo uso de la API de Nedepend, la herramienta sencilla llamada NDepend Metrics Reporter, para poder presentar gráficas de evolución de las mismas, funcionalidad que la versión 4.5, la única disponible al inicio de proyecto, no tenía.

Instalación de NDepend

Simplemente facilitando un e-mail se puede descargar una versión de prueba de 14 días completamente funcional. En nuestro caso, tras usarlo un tiempo, pudimos comprobar que se ajustaba perfectamente a las necesidades del proyecto y nos pusimos en contacto con su departamento de atención al cliente, que nos amablemente nos ha facilitado una licencia *Academic Sponsor* por el tiempo de duración del proyecto, habiendo accedido incluso en una ocasión a la extensión de su fecha de caducidad cuando se produjo un retraso que impidió presentar el presente TFG en septiembre.

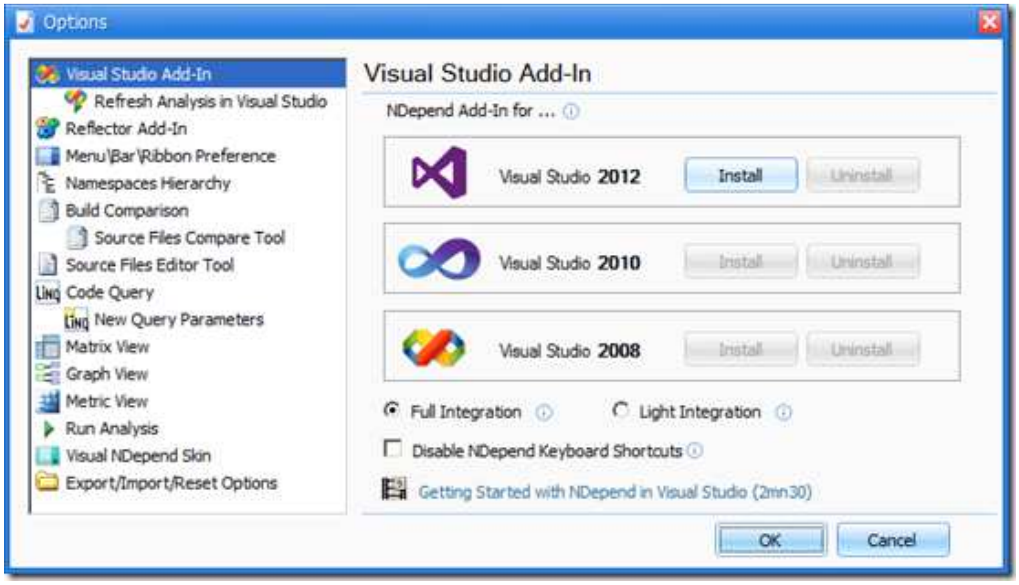


Figura 9-31: Instalador de NDepend

La instalación es a través de un cómodo interfaz. Las librerías y licencia quedan montadas en un directorio a nuestra elección, que deberemos recordar pues los ejecutables de todo desarrollo basado en la API de NDepend deben de ubicarse en dicha carpeta

Funcionamiento de NDepend

Una vez instalado, si se elige la integración completa, se accederán a sus funciones a través del menú principal de Visual Studio

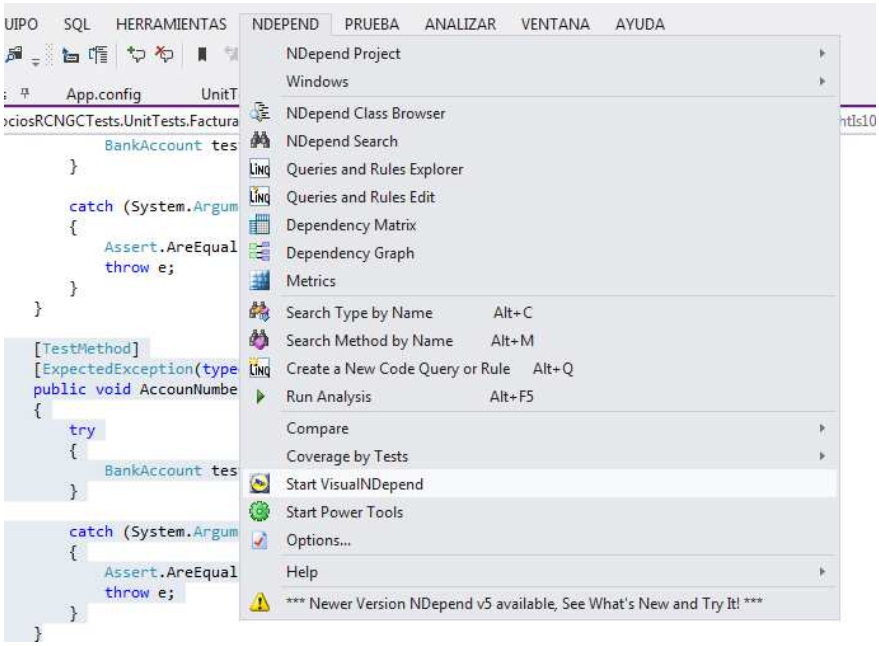


Figura 9-32: El NDepend integrado en Visual Studio

Una vez instalado, podemos crear un proyecto NDepend y asociarlo a la solución activa, definiendo los parámetros del mismo teles como ensamblados que se quieren monitorizar, características del análisis que se desea realizar (por ejemplo, si deseamos guardar histórico de los mismos o efectuar 'code diff'[103]) y la manera en que se desea que se presenten los resultados en los informes HTML.

Cada vez que abramos la solución, NDepend presentará dentro de Visual Studio los resultados de las reglas que hayamos definido mediante CQLINQ, y abrirá una ventana en el navegador con el informe HTML que hayamos definido en la configuración del proyecto NDepend. También podemos ordenar el análisis y ver un resumen del último desde un icono integrado en la parte inferior derecha de Visual Studio.

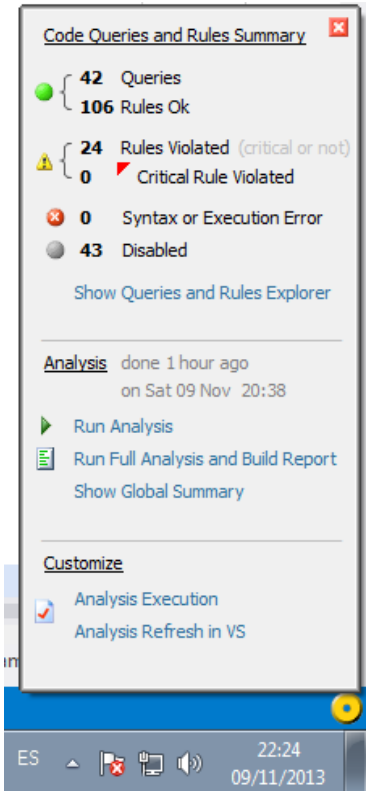


Figura 9-33: Ejecutando un análisis de código desde NDepend

NDepend incluye además una completa herramienta llamada Visual NDepend que integra todas sus funciones:

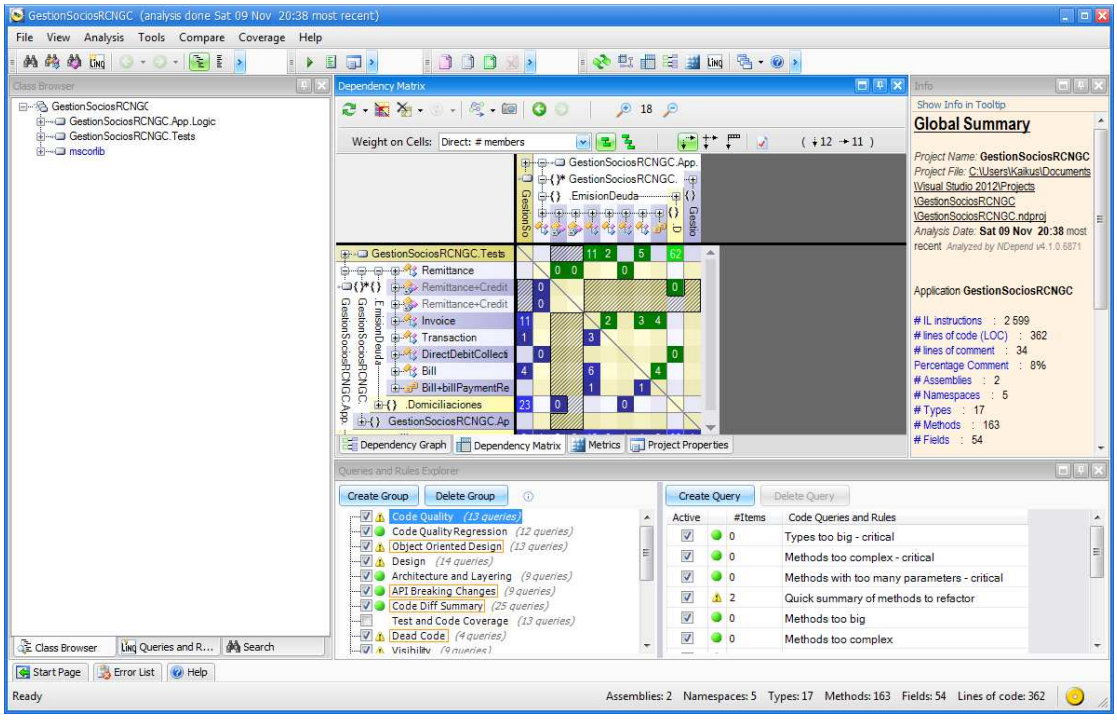


Figura 9-34: Visual NDepend

Métricas en NDepend

NDepend dispone de todas las métricas de las que hablamos durante la exposición sobre la Calidad del Código (cohesión, acoplamiento, complejidad ciclomática, líneas de código…), pero, además nos da acceso, entre sus 82 métricas, a otras realmente interesantes:

Métricas de Acoplamiento

- **Association Between Classes (ABT):** La Asociación entre clases de una clase o estructura particular es el número de otros tipos directamente usados en el cuerpo de sus métodos.

- **Type Rank:** El valor de esta métrica se calcula aplicando el agoritmo PageRank[104] de Google sobre el grafo de dependencias entre clases. Ha sido ponderado para que el valor medio de esta métrica sea 1. Los tipos con un alto TypeRank deben ser especialmente vigilados debido su relevancia.

- **Number of Overloads:** El número de sobrecargas de un método. Si un método no está sobrecargado, este valor es 1. También se aplica a constructores. Un método sobre el que se han hecho más demasiadas sobrecargas (por ejemplo, 6), resulta difícil de mantener y refactorizar, y es un posible síntoma de una acoplamiento excesivo.

- **Instability(I):** La inestabilidad es el ratio entre el acoplamiento eferente y el acoplamiento total

  $$I = Ce/(Ca+Ce)$$

  Es indicador de la resiliencia del ensamblado ante el cambio. El valor de esta métrica va de 0 (totalmente estable, pues no depende de nadie) a 1 (totalmente inestable, pues depende completamente del exterior). Una alta 'inestabilidad' no es mala, si la clase no es abstracta, como vemos en la Figura 9-35.

Métricas de Abstracción

- **Abstractness(A):** La abstracción de un ensamblado es el ratio entre el número de tipos abstractos (clases e interfaces) sobre el número de tipos internos de la clase. Su rango va 0 (un ensamblado absolutamente concreto) a 1 (formado tan solo por abstracciones)

- **Distance from Main Sequence:** Los valores individuales de abstracción o inestabilidad no son indicativos de problemas, pero su combinación sí nos avisa de ellos. Definimos la Distancia sobre la Secuencia principal como la distancia perpendicular normalizada sobre la línea A+I=1, que llamamos *secuencia principal*. Esta métrica nos ayuda a descubrir si existe un equilibrio entre la abstracción y la inestabilidad, como podemos observar en la Figura 9-35. Un valor por encima de 0,7 suele ser indicativo de problemas.

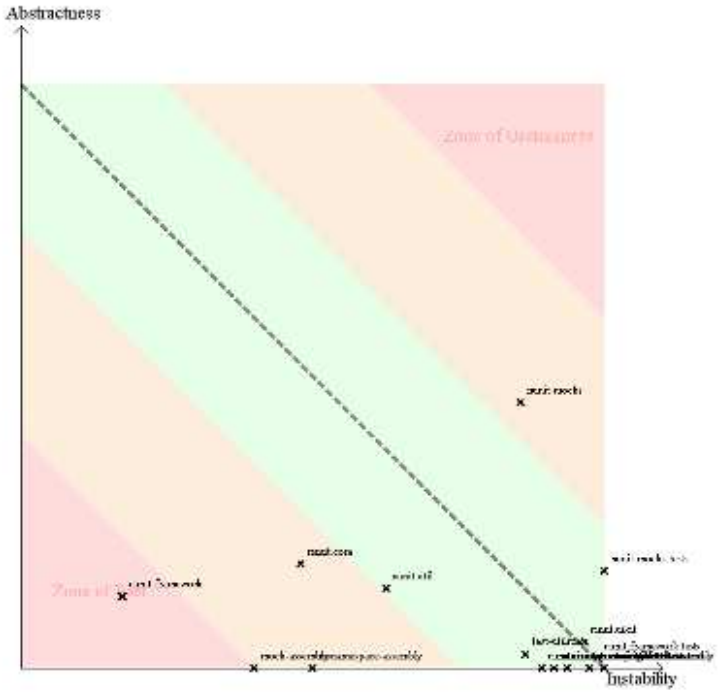


Figura 9-35: Abstraccion e Inestabilidad. Distancia sobre la secuencia principal

Un interesante artículo sobre como aplicar las métricas de NDepend a un estudio de acoplamiento lo podemos ver en el artículo 'Code metrics on coupling, dead code, design flaws and re-ingeneering[105] de Patrick Smachia

Métricas de Cohesión

- **Relational Cohesion(H):** La cohesión relacional es el número de relaciones internas por tipo. Sea R el número de relaciones entre tipos que son internas al ensamblado (no conectadas a tipos externos al ensamblado). Sea N el número de tipos del ensamblado. Entonces

  $$H = (R+1)/N$$

  (el +1 extra evita que H= cuando N=1)

  La cohesión relacional representa la relación que tiene en ensamblado entre todos sus tipos. Dato de las clases dentro de un ensamblado deberían estar normalmente fuertemente relacionadas, la cohesión debería ser alta. Por otra parte, si es demasiado alta podría haber excesivo acoplamiento. Un rango aceptable sería de 1.5 a 4.0. Valores fuera del mismo podrían resultar problemáticos.

Métricas referentes a Clean Code

- **IL Nesting Depth:** La profundidad de anidamiento es el número máximo de ámbitos encapsulados que podemos encontrar en un método. Profundidades mayores a 4 son difíciles de mantener y refactorizar. Si llega a 8, nos encontramos con un problema grave de claridad de código

Otras métricas interesantes

- **Number of Children(NOC**): El número de hijos es el número de subclases derivada de una clase dada, indpendientemente de su nivel dentro del árbol de herencia. En el caso de interfaces, es el número de clases que lo implementan.

- **Depth of Inheritance Tree (DIT):** La profundidad en el árbol de herencia es cuenta el numero de clases base desde la clase actual hasta System.Object, por lo que DIT>=1 siempre. Un DIT demasiado alto suelen ser difíciles de mantener.

- **Level:** Esta métrica está definida para ensamblados, espacios de nombres, tipos y métdodos, y se define, por ejemplo, para ensamblados, como sigue:

    - Level 0: El namespace no usa ningún otro namespace
    - Level 1: El namespace solo usa directamente espacios de nombres definidos en ensamblados de terceras partes.
    - Level 1+ (Max Level de los ensamblados que usa directamente)
    - Level N/A: El espacio de nombres se ve envuelto en un ciclo de dependencia

  Esta métrica ayuda a clasificar los elementos de código en alto, medio y bajo nivel, y, al mismo tiempo, perimite detectar dependencias cíclicas.

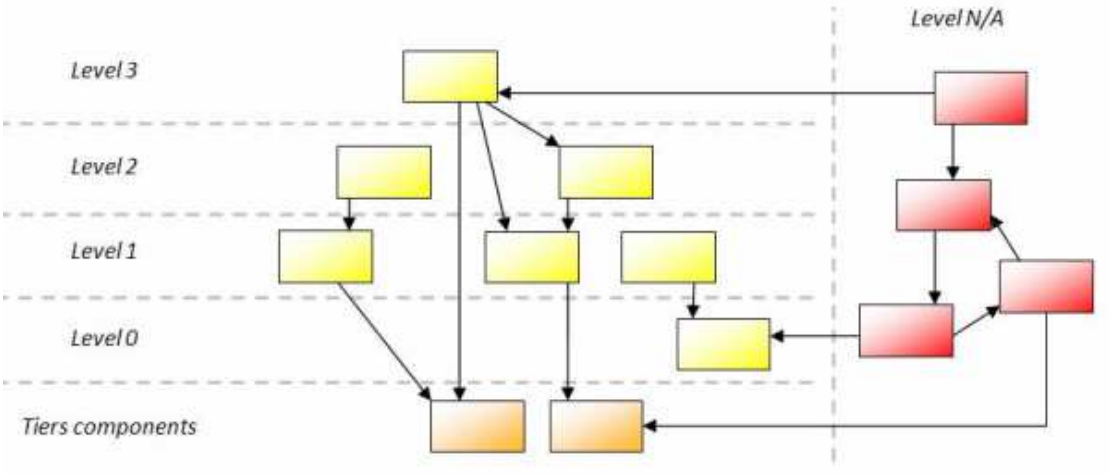


Figura 9-36: La métrica Level
Fuente: Patrick Smacchia (CodeBeter.com)

> Nuevamente un artículo muy interesante, llamado 'Layering, the Level Metric and the Discourse of Method'[106], de Patrick Smacchia, nos ayuda a sacar partido de esta métrica.

Como ya comentamos, muchas de estas métricas se pueden obtener a varios niveles: por ensamblado, espacio de nombres, tipos o métodos, lo que permite profundizar el estudio

Pero la gran potencia de NDepen estriba en poder integrar dichas métricas con sentencias de consulta LINQ sobre el código, lo que le da una plena libertad al usuario para crear las suyas propias. Hemos hecho uso de esta capacidad para crear métricas definidas por el usuario en NDepend Metrics Reporter.

Adquisicíon de las métricas NDepend

Quizá la funcionalidad que más interesante nos resultó de NDepend es la posibilidad de almacenar el histórico de todos los análisis realizados.

NDepend crea una carpeta, ubicada en el directorio la raíz de la solución, donde, en cada análisis, vuelca todo lo necesario para presentar un informe en HTML: snippets en javascript, hojas de estilo y plantillas personalizables, ficheros XML con los resultados de las métricas y otros valores… Cada vez que realizamos un nuevo análisis, si se ha realizado transcurrido desde el anterior un tiempo mínimo configurable, los últimos resultados existentes son almacenados un carpetas de históricos antes de actualizar los datos.

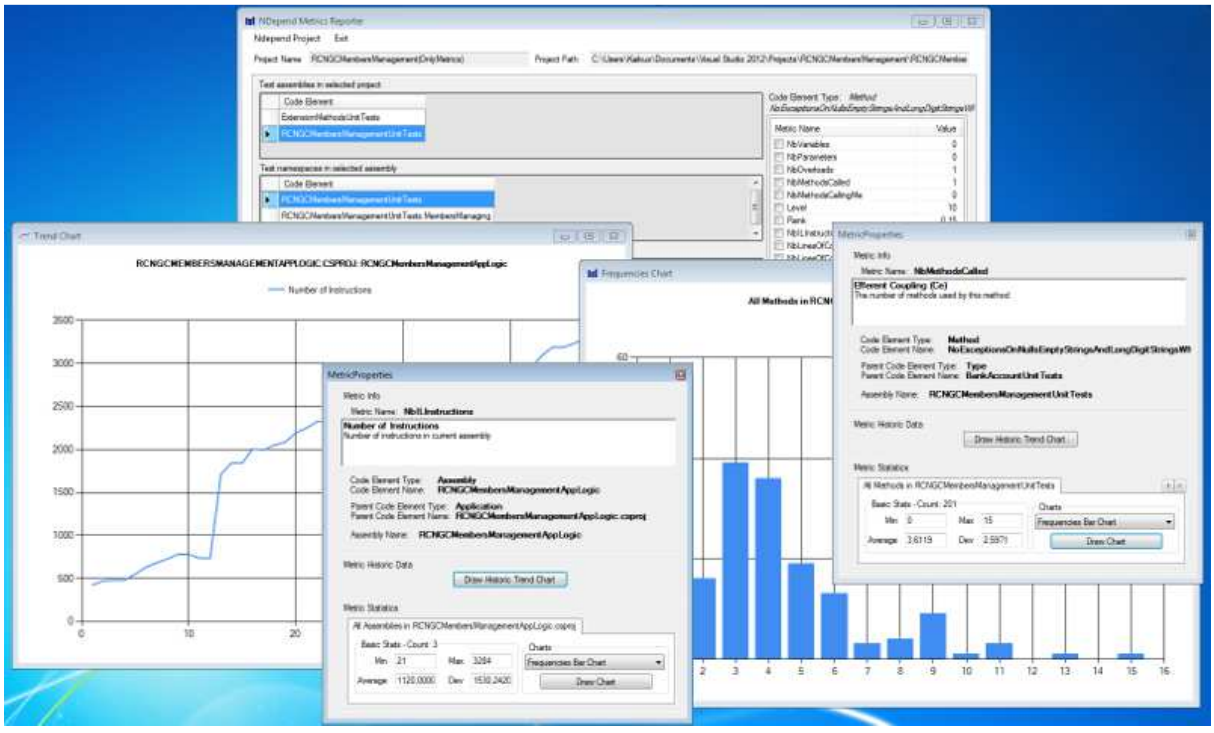


Figura 9-37: NDepend Metrics Reporter

En un principio el proyecto iba a hacer uso de dichos ficheros XML para alimentar una base de datos, que después podríamos consultar para obtener métricas y representar gráficas, pero finalmente, tras algunos e-mail intercambiados con los desarrolladores, se nos recomendó hacer uso de la API de NDepend para acceder directamente a dicha información, que se guarda en un fichero llamado 'VisualNDepend.bin' que se crea en cada análisis.

Como resultado de ello se creó la aplicación NDepend Visual Reporter (Figura 9-37), a través de la cual podemos ver directamente las métricas de código y representar resultados.

Desde esta aplicación tenemos acceso a todo el histórico de métricas recogidas durante el desarrollo del proyecto, pudiendo recuperar métricas que representamos tubularmente, así como gráficos históricos de líneas e histogramas de frecuencias.

10. RESULTADOS DE LA EVALUACION DEL CÓDIGO

10.1. Sobre la metodología de evaluación

Para exponer los resultados del estudio tomaremos como guía la cuestión que planteamos al principio de nuestra memoria: ¿Que es el Quality Code?

Nos centraremos, por tanto, en evaluar, uno a uno, los elementos que consideramos son el indicativo de la calidad del código.

- Conformidad con los requisitos
- Fiabilidad
- Legibilidad
- Flexibilidad y reusabilidad

Siempre que nos sea posible, utilizaremos las métricas obtenidas a través de *NDepend Metrics Reporter* como base y soporte de la argumentación. En ocasiones utilizaremos otros datos como apoyo.

Los gráficos de NDepend Metrics Reporter

La herramienta diseñada para recoger y exponer las métricas ha resultado de inestimable ayuda para el estudio, especialmente gracias a la posibilidad de crear métricas personalizadas, que hemos usado ampliamente para obtener evoluciones históricas sobre valores estadísticos de métricas base. Los dos tipos de gráfico que se ha definido son:

Histogramas de frecuencias

El valor puntual de una métrica para un solo elemento de código no es demasiado representativo. Por ejemplo, ¿de qué nos sirve saber exactamente cual es el LCOM de la clase *'Invoice'*?. ¿Es suficiente una media de la métrica de todas las clases? Es útil, sí, especialmente si disponemos de máximos, mínimos y desviaciones. Pero, lo verdaderamente interesante es ver los LCOM de todos los tipos del ensamblado. Una tabla de valores nos presenta toda la información, pero sería demasiado extensa y farragosa de manejar.

Para poder analizar comparativamente todos los resultados de una métrica particular hemos decidido usar un histograma de frecuencias para cada métrica, en los que:

- ***Eje X:*** *El valor de la métrica*

- ***Eje Y:*** La frecuencia con que dicho valor se repite. En nuestro caso dicha frecuencia nos indica cuantos elementos de código, para la métrica en estudio, tienen ese valor.

Así, con un solo vistazo, podemos evaluar globalmente una métrica de código: cual es el valor más repetido (moda), máximos y mínimos, en qué rango se agrupan, si hay dispersión.

Veamos, por ejemplo, algo tan simple como el número de líneas de todos los métodos del ensamblado.

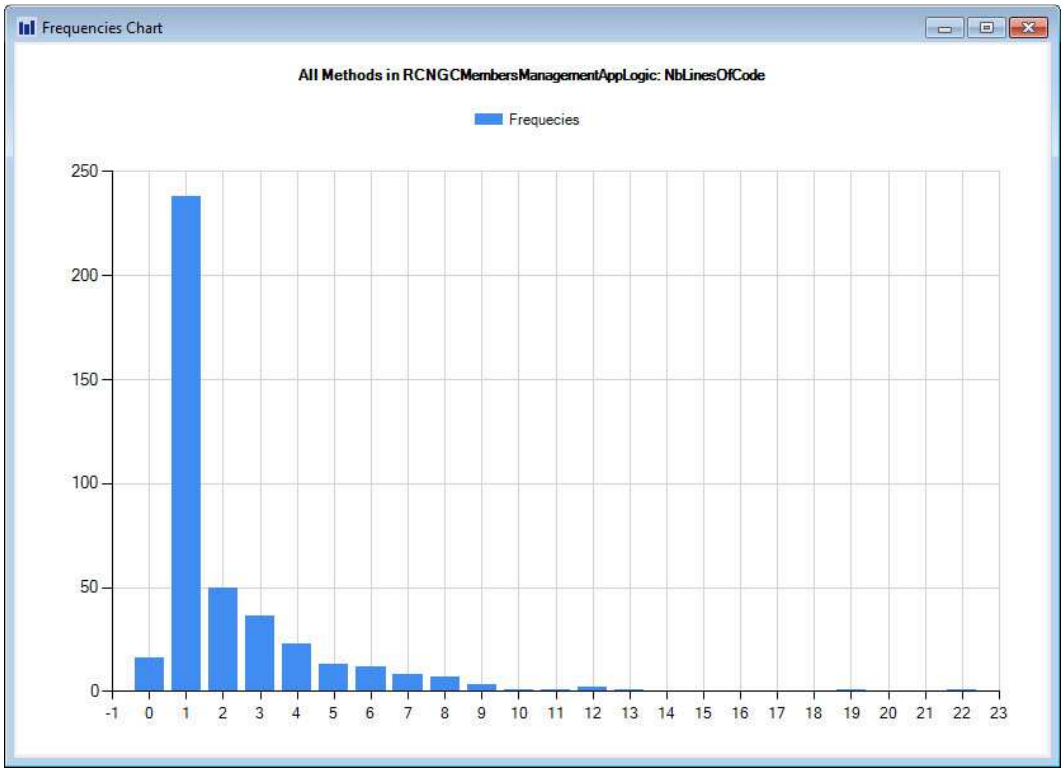


Figura 10-1: Histograma de frecuencias de líneas de código de todos los métodos en el ensamblado

Como podemos observar, con diferencia el valor que más aparece es 1. Es decir, que los métodos tienen, en su gran mayoría (más de 240 de ellos), una sola instrucción. Mas tarde discutiremos el por qué de la enorme prevalencia de este valor. Pero, lo que si podemos apreciar, de un solo vistazo, es que el resultado ha sido que los métodos son principalmente muy pequeños.

Evolución histórica

Con la anterior gráfica podemos analizar un momento puntual en el estado del código, en particular lo usaremos para ver el resultado final. Pero dicho valor puede ser producto de una serie de refactorizaciones finales.

Esto es lo que normalmente ocurre en desarrollos que no son TDD: el código va degradándose hasta que se hace tan imposible de mantener que requiere una gran refactorización. Una gráfica de 'calidad de código' mostraría una serie de picos a lo largo de su desarrollo.

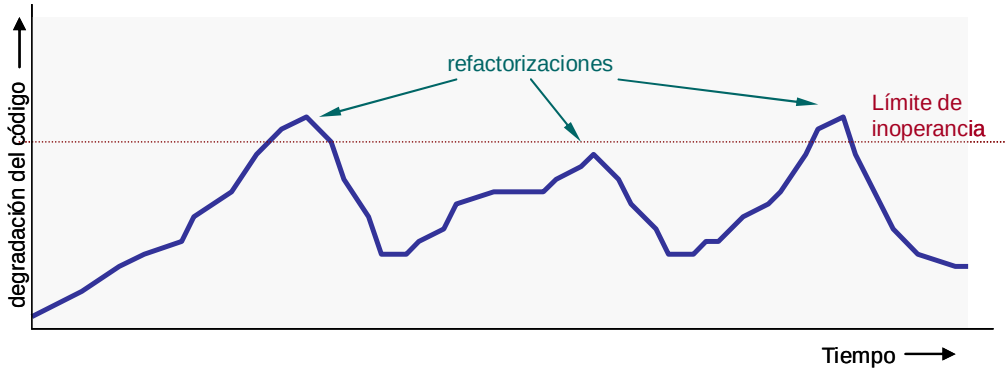


Figura 10-2: Refactorizaciones en un desarrollo clásico

Sin embargo, el TDD, al estar el código en permanente refactorización, nunca deberán llegar a formarse dichos picos. Pasados los inicios del desarrollo, en los que la escasa cantidad de código produce grandes variaciones en la calidad con cada ciclo de test, se debería producir una estabilización con pequeños dientes de sierra y ligeras tendencias.

Esto último es lo que buscaremos en los gráficos de tendencias

Sin embargo, hemos de hacer una consideración final. En el actual desarrollo las últimas aportaciones fueron las tipos que conforman la definición del esquema para los envíos XML del SEPA, muy complejos y con un diseño muy específico (sin métodos, con muchos parámetros y muy extensos) que desvirtuaban los resultados de las métricas. Finalmente se tomó la decisión de separar dichas clases a un ensamblado externo, una biblioteca a la que hacer referencia. Las medidas así obtenidas son más reales, pero en la evolución histórica de ciertas métricas se puede una variación final importante.

10.2. Resultados

10.2.1. Conformidad con los requisitos

Durante la fase de análisis, establecimos los requisitos en forma de historias de usuarios. Con la ayuda del BDD y a herramienta SPecFlow convertimos dichas historias de usuario en escenarios, y para cada uno de ellos se definió un test de aceptación.

El hecho de que todos estos test de aceptación, al final del proyecto, se encuentren en verde, implica que todos los requisitos de usuario recogidos en el mismo han sido perfectamente cubiertos.

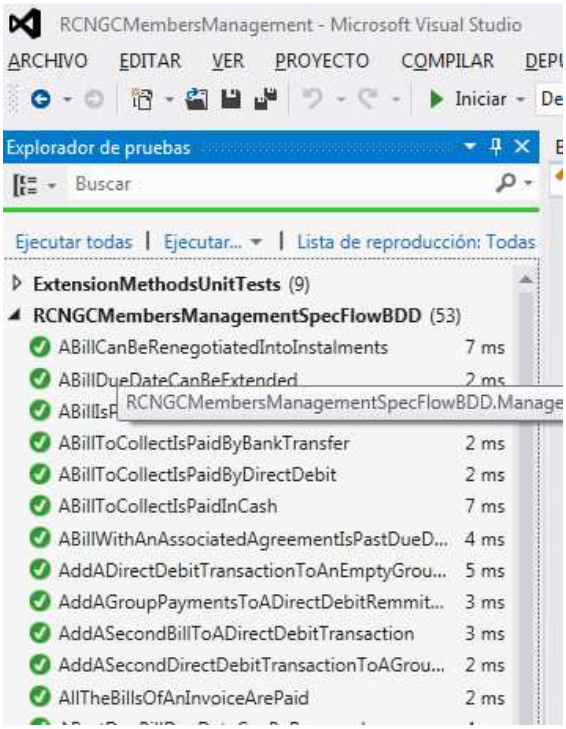


Figura 10-3: Todos los test de aceptación de los escenarios BDD 'en verde'

Únicamente en el caso en que dichas historias de usuario y escenarios no hubieran sido correctamente recogidos, podríamos argumentar no conformidad con los requisitos, pero:

- Dichas historias de usuario y escenarios fueron hechos en colaboración con el cliente, y será él mismo, con cada entrega de una iteración, quien podrá comprobar que se hizo exactamente lo que con él se acordó.

- Si, a posteriori, el cliente considera que faltaba algo en la historia de usuario, es fácil de añadir.

- En cualquier caso lo que estamos estudiando es si TDD produce mejor conformidad con los requisitos. Y, ciertamente, TDD garantiza el cumplimento de lo que aparece en los test de aceptación, independientemente de si, en el análisis, las historias han sido incompletas

10.3. Fiabilidad (Tested Code)

El código en TDD está 100% testado, simple y llanamente.

En otras metodologías resulta tremendamente complejo llegar a este porcentaje de cobertura. Debemos, en cualquier caso tener en cuenta que 100% de cobertura no implica 100 falta de fallos. Puede que existan escenarios no previstos. Pero el desarrollo guiado por test es también llamado 'desarrollo guiado por ejemplos'. Es decir,

codificamos en base a ejemplos, solo lo que los ejemplos nos exige. Si hemos sido suficientemente

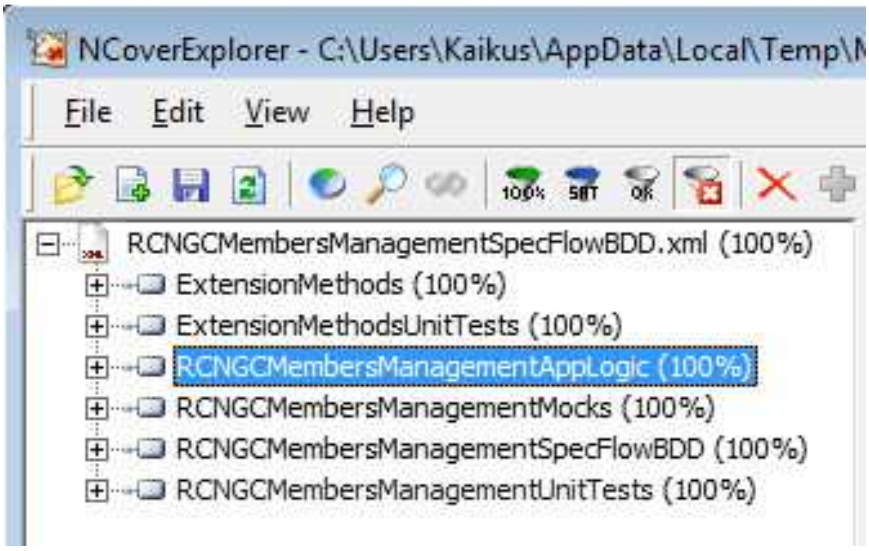


Figura 10-4: Código de la aplicación 100% cubierto por tests

En otras metodologías resulta tremendamente complejo llegar a este porcentaje de cobertura. Debemos, en cualquier caso tener en cuenta que 100% de cobertura no implica 100% falta de fallos. Puede que existan escenarios no previstos. Pero el desarrollo guiado por test es también llamado 'desarrollo guiado por ejemplos'. Es decir, codificamos en base a ejemplos, solo lo que los ejemplos nos exigen.

Por tanto, si el programador ha sido metódico, siguiendo las pautas que comentamos en la disciplina y requerimientos a la hora de desarrollar TDD, podemos afirmar que nos encontramos ante código casi 100% robusto.

10.4. Legibilidad (Clean Code)

La legibilidad es hasta cierto punto un elemento subjetivo, difícilmente medible. Algunas cosas son simplemente observables como el uso de nombres con significado, o métodos concisos y claros:

En el ejemplo de la Figura 10-5 marcamos una factura como pendiente de pago 'ToBePaid' si le quedan recibos al cobro y no le queda ninguno impagado. Es muy sencillo de seguir, pues usamos nombres con significado, y los detalles de implementación de cada una de las tareas de la inicialización las realiza un método diferente (*Single Responsability Principle*).

```
public class Invoice: BaseInvoice
{
    ……

    public void SetInvoiceToBePaidIfHasNoUnpaidBills()
    {
    if (InvoiceHasBillsToCollect() && InvoiceHasNoUnpaidBills())
        this.invoiceState = InvoicePaymentState.ToBePaid;
    }

    private bool InvoiceHasNoUnpaidBills()
    {
        Dictionary<string, Bill> billsCollection = this.invoiceBills;
        var unpaidBills = billsCollection
            .Where(bill => bill.Value.PaymentResult == Bill.BillPaymentResult.Unpaid);
        return (unpaidBills.Count() == 0);
    }

    private bool InvoiceHasBillsToCollect()
    {
        Dictionary<string, Bill> billsCollection = this.invoiceBills;
        var toCollectBills = billsCollection
            .Where(bill => bill.Value.PaymentResult == Bill.BillPaymentResult.ToCollect);
        return (toCollectBills.Count() != 0);
    }
    ……
}
```

Figura 10-5: Ejemplo de 'Clean Code' en la aplicación desarrollada

En algunos casos hay que llegar al compromiso de decidir si resulta más claro una instrucción no demasiado compleja o extraerla a un método externo, puesto que continuas llamadas hacen el código a veces demasiado disperso. A veces mirar desde otras ópticas más allá del Clean Code ayuda. P.E.: ¿me conviene extraer el método por cuestiones de refactorización (principios SRP, exceso de acoplamiento…)

Dicho todo esto, ¿qué métricas pueden ayudarnos a comprobar la claridad el código? Vamos algunas.

Tamaño de los métodos

Mirando este histograma de la Figura 10-6 podemos observar que como media, vemos que los métodos tienen muy pocas líneas de código, según las estadísticas, una media de 2,2 lineas, con una desviación de 2,4. En cualquier caso hemos de prestar atención a algunos datos que nos muestra el histograma:

- Hay una enorme prevalencia de métodos con una sola línea de código. Si bien hay muchos métodos con estas características, hay que tener en cuenta que, entre ellos estan los 'getters' y 'setters' de las propiedades.

- Hay dos métodos, de 19 y 22 líneas, que desvirtúan el gráfico. Son dos métodos relacionados con la creación del fichero SEPA.

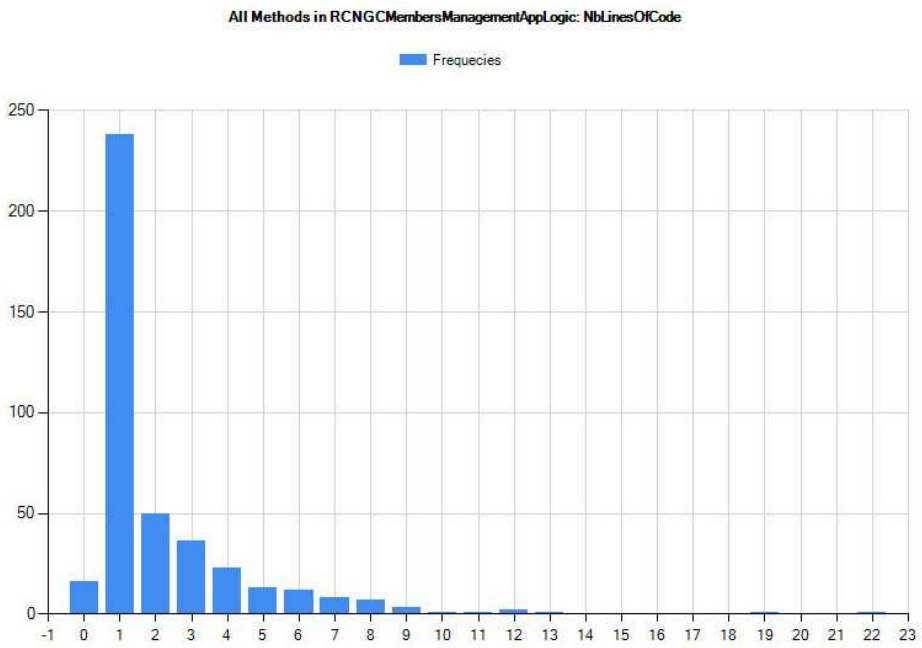


Figura 10-6: Histograma de frecuencias del número de líneas de código para todos los métodos del ensamblado

Por otra parte, en la evolución histórica de la media de número de líneas por método, (Figura 10-7) ésta se ha mantenido relativamente estable, con una tendencia de bajada final debida a la anteriormente nombrada implementación del SEPA, cuyas clases no incluían métodos, sino tan solo campos y propiedades (LOC=1). Después de traspasar dichas clases a otro ensamblado, vemos que el resultado final vuelve al estabilizado 2,2.

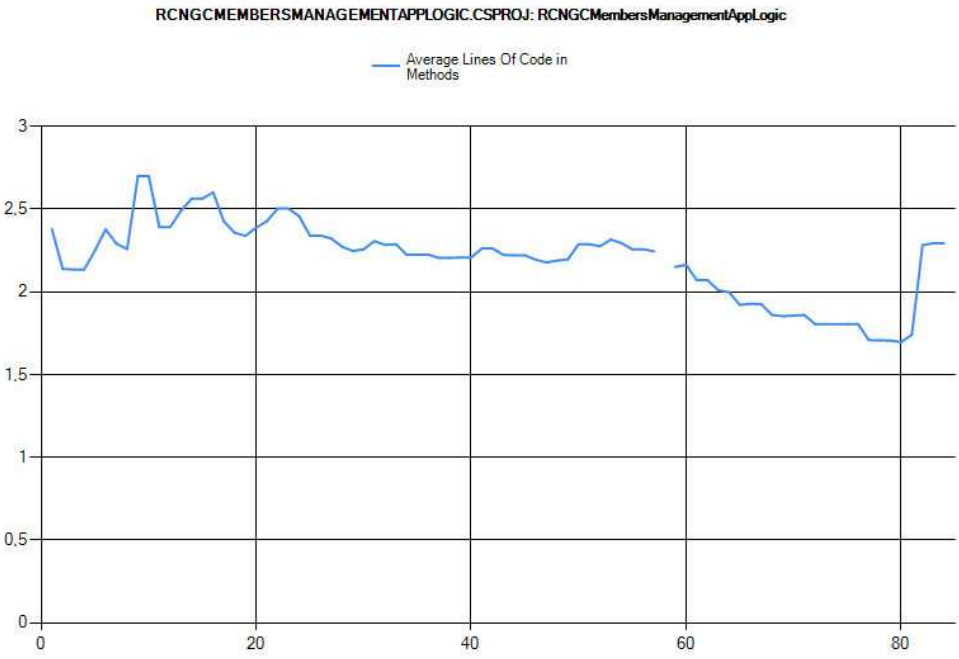


Figura 10-7: Evolución histórica de la media de la métrica LOC para todos los métodos del ensamblado

Complejidad Ciclomática

¿Cuántas decisiones anidadas deben de tomarse en los métodos? En el Clean Code se intenta evitar al máximo el uso de sentencias condicionales (if/else – switch/case), factorizándose siempre que sea posible en base a polimorfismo. En el histograma de la podemos observar que, el código lineal, sin una sola decisión, sin un solo 'if', es decir de complejidad = 1, es lo común. En algunos métodos recurrimos al condicional para alguna comprobación básica como la del código del ejemplo, y solo en contadísimas ocasiones debemos hacer uso de sentencias swith/case que disparan esta métrica.

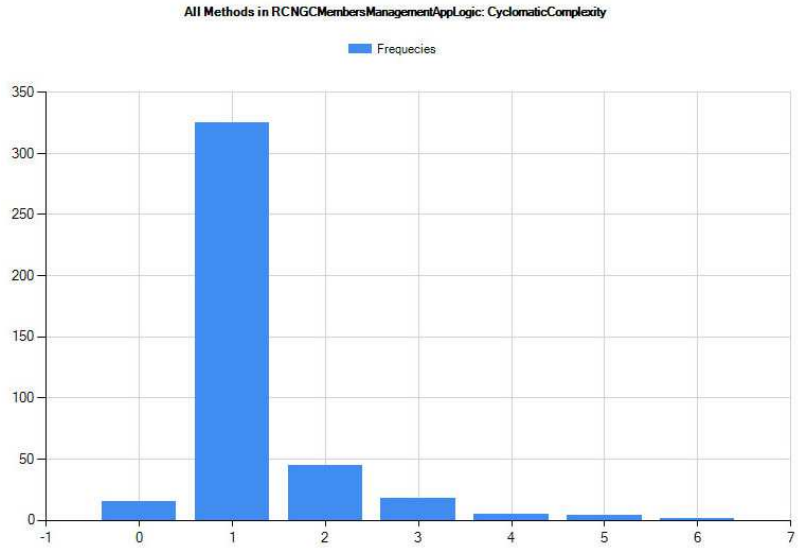


Figura 10-8: Histograma de frecuencias de la Complejidad Ciclomática

Nuevamente, en el historico, observamos un comportamiento similar al anterior

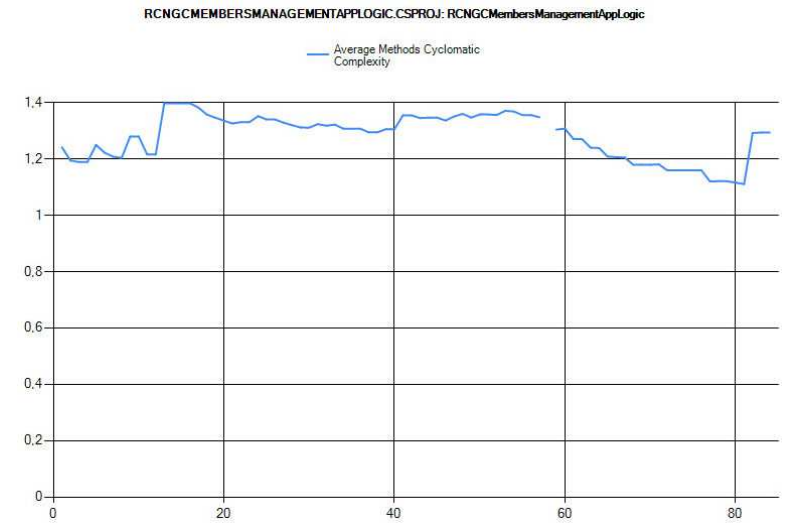


Figura 10-9: Evolución histórica de la media de la métrica Complejidad Ciclomática para todos los métodos del ensamblado

Profundidad de anidamiento

¿Tiene nuestro código demasiados ámbitos anidados, por ejemplo, como resultado de bucles o sentencias de decisión?. Realmente no. En la inmensa mayoría de los métodos ni siquiera existen.

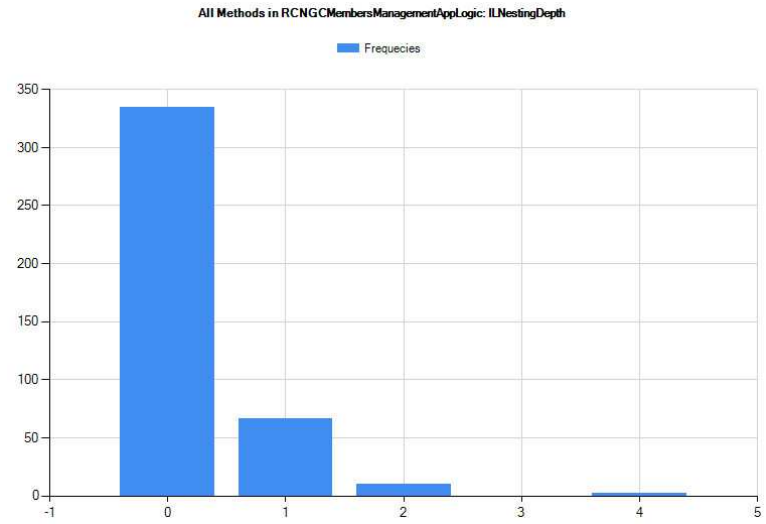


Figura 10-10: Histograma de frecuencias de la Profundidad de Anidamiento

Uno de los factores que ha ayudado ha obtener esto resultado es la inclusión de las sentencias LINQ en Visual Studio 2008 y .Net Framework 3.5. Gracias a LINQ las colecciones implementan, entre otras funciones más complejas, un método .Select que permite recorrerlas sin necesidad de un bucle 'foreach'.

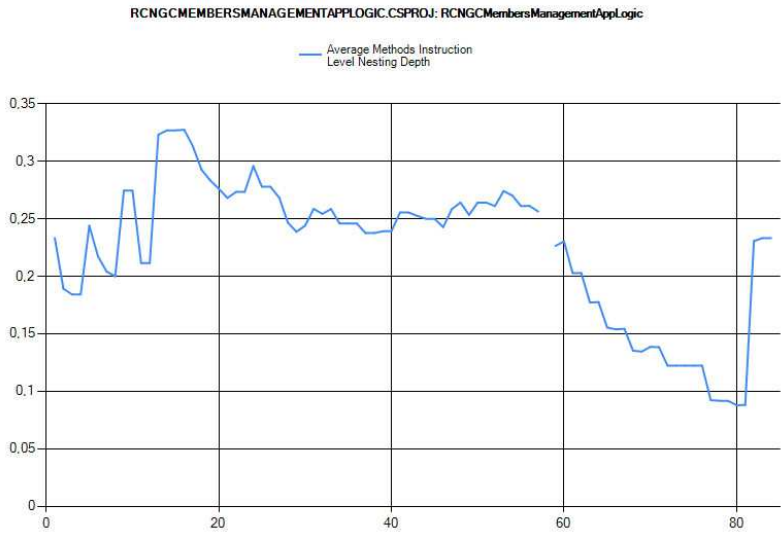


Figura 10-11: Evolución histórica de la media de la métrica Profundidad de Anidamiento para todos los métodos del ensamblado

En el histórico de la Figura 10-11 volvemos a observar el mismo comportamiento: un inicio irregular, una estabilización y luego una tendencia (en este caso de fuerte bajada) al implementar las clases de SEPA, que, como solo incluyenmétodos 'getter' y 'setters', bajan el valor de la media. Tras su escisión, volvemos al valor estable que finalmente vemos desglosado en el histograma.

10.5. Flexibilidad y Reusabilidad (Principios OOD)

La flexibilida y la reusabilidad se consuguen siguiendo los principios SOLID del OOD, que recordemos, son:

- **(S) SRP - Single Responsability Principle[107]:** Una clase debe tener una UNICA responsabilidad.
- **(O) OCP – Open Closed Principle[108]:** Una clase debería estar abierta a su extensión, pero cerrada a su modificación
- **(L) LSP – Liskov Substitution Principle[109]:** Una clase derivada debería poder sustituir a un su clase base.
- **(I) ISP – Interface Segregation Principle[110]:** Realizar un interfaz específico para cada cliente es mejor que no solo genérico.
- **(D) DIP – Dependency Inversion Principle[111]:** Las clases deberían depender de abstracciones, no de implementaciones.

Es muy difícil estudiar cada principio en base a métricas, pero hay conceptos que las relacionan, de los que ya hemos hablado extensamente cuando expusimos las métricas.

- Cohesión
- Acoplamiento
- Abstración

10.5.1. Cohesión

Tremendamente relacionados con el principio de responsabilidad única, vemos que los métodos deben ser fuertemente cohesionados dentro de una clase. Ya vimos que el tamaño de los métodos es normalmente pequeño, lo que resulta un indicio de cohesión. Pero tenemos métricas específicas, como la de *'Lack Of Cohesion Of Methods' (**LCOM**).* Sin embargo, una excesiva cohesión no es siempre buena, pues puede indicar sobre-acoplamiento en las clases. Para ello podemos estudiar la métrica 'Relational Cohesion'***(H).***

Lack of Cohesion Of Methods (LCOM)

En nuestro caso estudiaremos una métrica derivada, LCOM-HS (de Henderson-Sellers), que fluctúa entre los valores [0-2]. Un valor de 0 indica perfecta cohesión, y un valor superior a 1 indica que podemos tener problemas con el método.

El algoritmo utilizado para el cálculo de la métrica es:

- LCOM = 1 – (sum(MF)/M*F)
- LCOM HS = (M – sum(MF)/F)(M-1)

Donde:

- M es el número de métodos en la clase (estáticos o de instancia, incluidos constructores, 'getters' y 'setters', eventos, y métodos 'dispose')
- F es el número de campos de instancia en la clase
- MF es el número de métodos de clase que acceden a un campo de instancia en particular
- Sum(MF) es la suma de todos los MF de todos los campos de instancia

La idea es que una clase es completamente cohesiva si todos los métodos usan todos los campos de instancia, con lo que sum(MF)=M*F y, por tanto, LCOM=0 y LCOMHS = 0

Este algoritmo tiene algunos problemas con las transitividades. En cualquier caso veremos los resultados de LCOM-HS *(Henderson-Sellers, 1995)[9]*, que fluctúa entre los valores [0-2]. Un valor de 0 indica perfecta cohesión, y un valor superior a 1 indica que podemos tener problemas con el método.

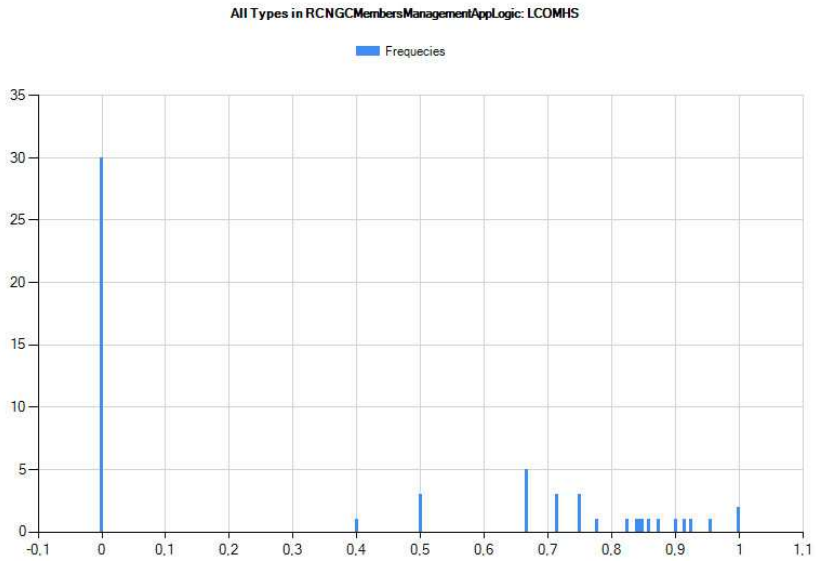


Figura 10-12: Histograma de frecuencia de LCOM-HS

Como podemos observar en la Figura 10-12, la inmensa mayoría de los métodos tienen una perfecta cohesión, y ninguno supera el umbral de peligro que es el valor 1.

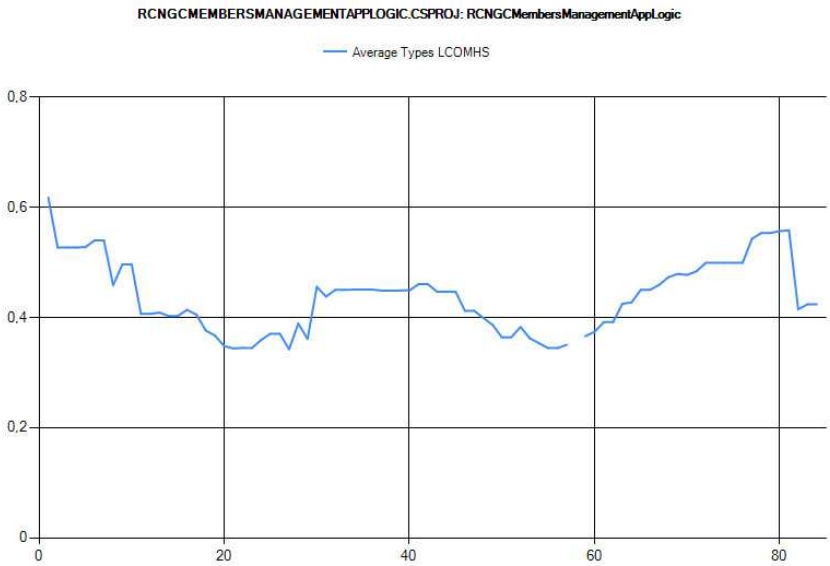


Figura 10-13: Evolución histórica de la media de la métrica LCOMHS para todas las clases del ensamblado

Se repite en la Figura 10-13 el esquema de históricos anteriores: inestabilidad inicial, estabilización alrededor del valor 0.4, y degradación que se corrige al extraer los tipos del SEPA.

Relational Cohesion (H)

La cohesión relacional mide el número medio de relaciones internas en un tipo

- Sea R el número de relaciones entre tipos que son internas al ensamblado (es decir, que no conectan con tipos fuera del ensamblado).
- Sea N el número de tipos en el ensamblado

$$H= (R+1)/N$$

(el +1 evita que H=0 cuando N=1)

Assemblies analyzed in selected project

Code Element	RelationalCohesion
ExtensionMethods	1
RCNGCMembersManagementAppLogic	2,137931
RCNGCMembersManagementMocks	0,5

Figura 10-14: Valor de la Cohesión Relacional para el ensamblado principal

Esta métrica es un indicativo de la relación que tiene el ensamblado con sus tipos, y nos ayuda, como antes comentamos, a detectar acoplamientos excesivos, se considera que un valor correcto es el que oscila entre[1.5, 4], por lo que nuestro ensamblado goza de un saludable 2.13 a este nivel.

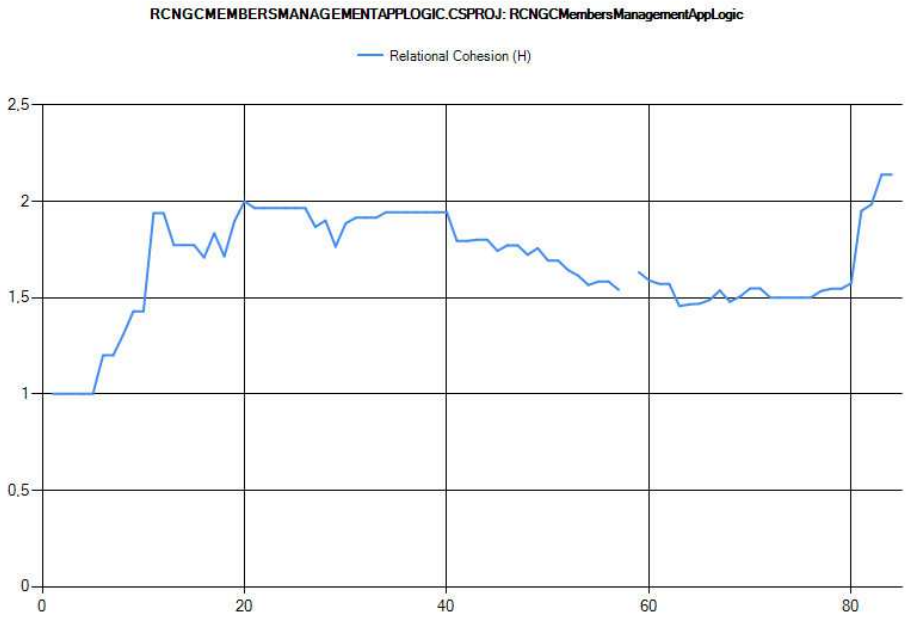


Figura 10-15: Evolución histórica de la métrica Cohesión Relacional

Como era de esperar, igual evolución que en las anteriores

10.5.2. Acoplamiento

Las métricas básicas de acoplamiento son el Acoplamiento Aferente y el Acoplamiento Eferente

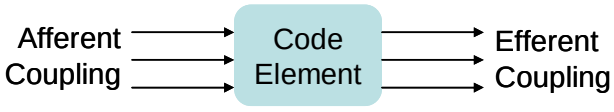


Figura 10-16: Acoplamiento Aferente y Eferente

En nuestro estudio revisaremos las métricas de acoplamiento a nivel de clases y de métodos.

Las medidas de acoplamiento nos permiten localizar algunos métodos o clases que tengan demasiadas responsabilidades, ya sea porque hacen demasiadas llamadas al exterior o porque son invocadas por un número excesivo de otros métodos o clases.

También nos ayudan a descubrir si existe gran dependencia a nivel de detalles de implementación entre dos clases. Cuando existe acoplamiento a este nivel termina por incumplirse el Open/Close Principle (OCP).

Por último, valores de Aferent Coupling=0 indican la existencia del llamado 'Dead Code', o código inaccesible, que puede surgir tras realizar las refactorizaciones.

Acoplamiento Aferente (Afferent Coupling)

A nivel de métodos

Llegados a este punto, hemos de decir que las métricas que cuentan el número de veces en que un método es llamado (Acoplamiento Aferente y Type Rank) presentan un grave problema para nuestro estudio:

- Las métricas que se han ido adquiriendo a lo largo del desarrollo han incluido siempre a los ensamblados de test y de BDD. Esto implica que las veces en que un método es llamado incluyen el número de métodos de test que lo usan, desvirtuando completamente el resultado, ya que no da información sobre el acoplamiento real. Por tanto los gráficos históricos son irrelevantes.

Por tanto se ha realizado al final un análisis descartando dichos ensamblados de test. Sin embargo el resultado arroja un valor desconcertante: casi 150 métodos que 'no son nunca llamados'.

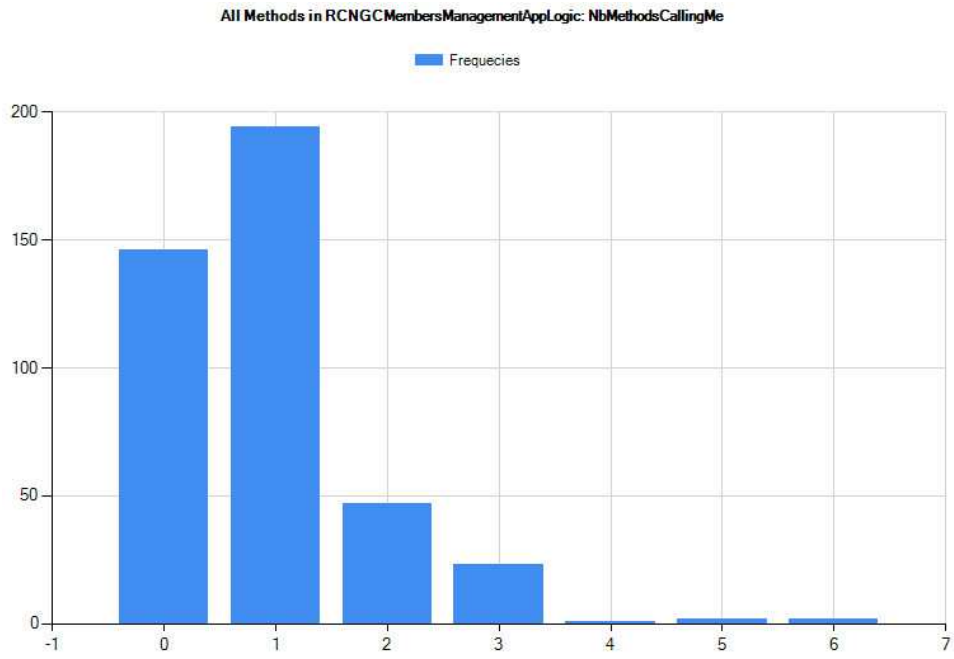


Figura 10-17: Acoplamiento Aferente en los métodos del ensamblado

¿Significa esto que nos encontramos ante una gran cantidad de código inaccesible, de 'Dead Code'?

¡No! Debemos recordar que estamos desarrollando una librería de clases, y además, parcial, por lo que muchos métodos no son llamados en ningún momento, sino que han

sido implementados para ser llamados desde otras librerías o desde un programa principal que haga uso de ésta.

Una vez aclarado este punto, podemos observar que salvo algunos métodos de cierta importancia, en la mayoría de los casos el Acoplamiento Aferente es mínimo, una buena señal de independencia y SRP.

A nivel de clases

Con las clases tenemos el mismo problema que con los métodos a la hora de estudiar históricos, pues las llamadas desde las clases del test desvirtúan los resultados. Nuevamente nos centramos en el análisis puntual que las descarta.

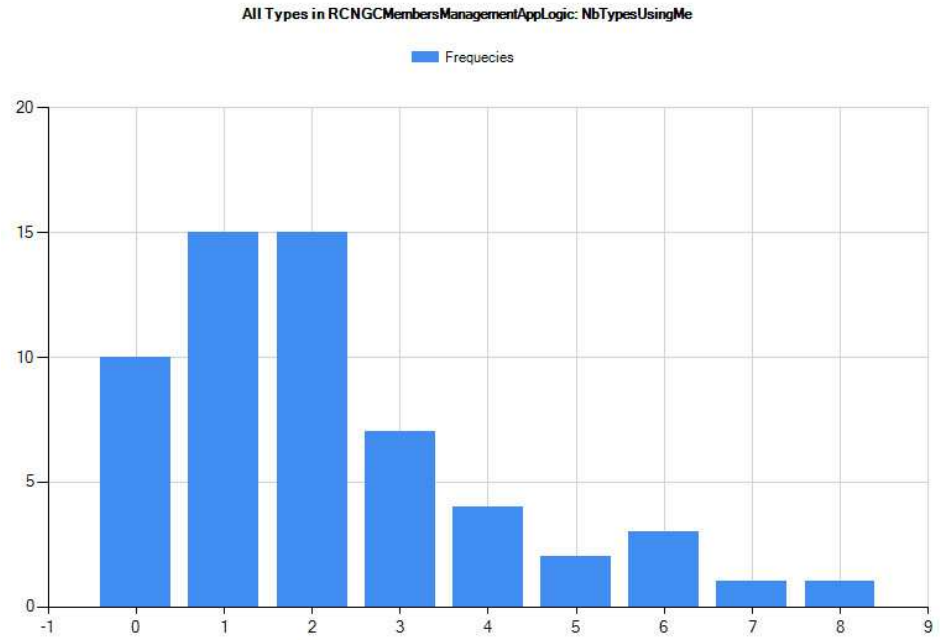


Figura 10-18: Acoplamiento Aferente en los tipos del ensamblado

Nos encontramos nuevamente con una nada despreciable cantidad de clases 'no llamadas', y la razón es la misma de antes. Ocurre en las 'Clases Controladoras', tales como *'BillsManager'*, *'InvoicesManager'*, 'DirectDebitRemmitancesManager' *o 'SEPA Manager'*, que están en lo alto del grafo de dependencias, o como el *'XMLValidator'*, clase de ayuda para la validación de los esquemas XSD, que finalmente solo hemos usado desde los test.

En el otro extremo tenemos un par de clases, como *'BankAccount'* o *'Bill'*, que obviamente son muy referenciadas en una biblioteca sobre facturación y adeudos bancarios.

Por lo demás, podemos concluir que el acoplamiento aferente está bastante controlado, y nuestras clases no están demasiado acopladas a este nivel.

Acoplamiento Eferente (Efferent Couplig)

A nivel de métodos

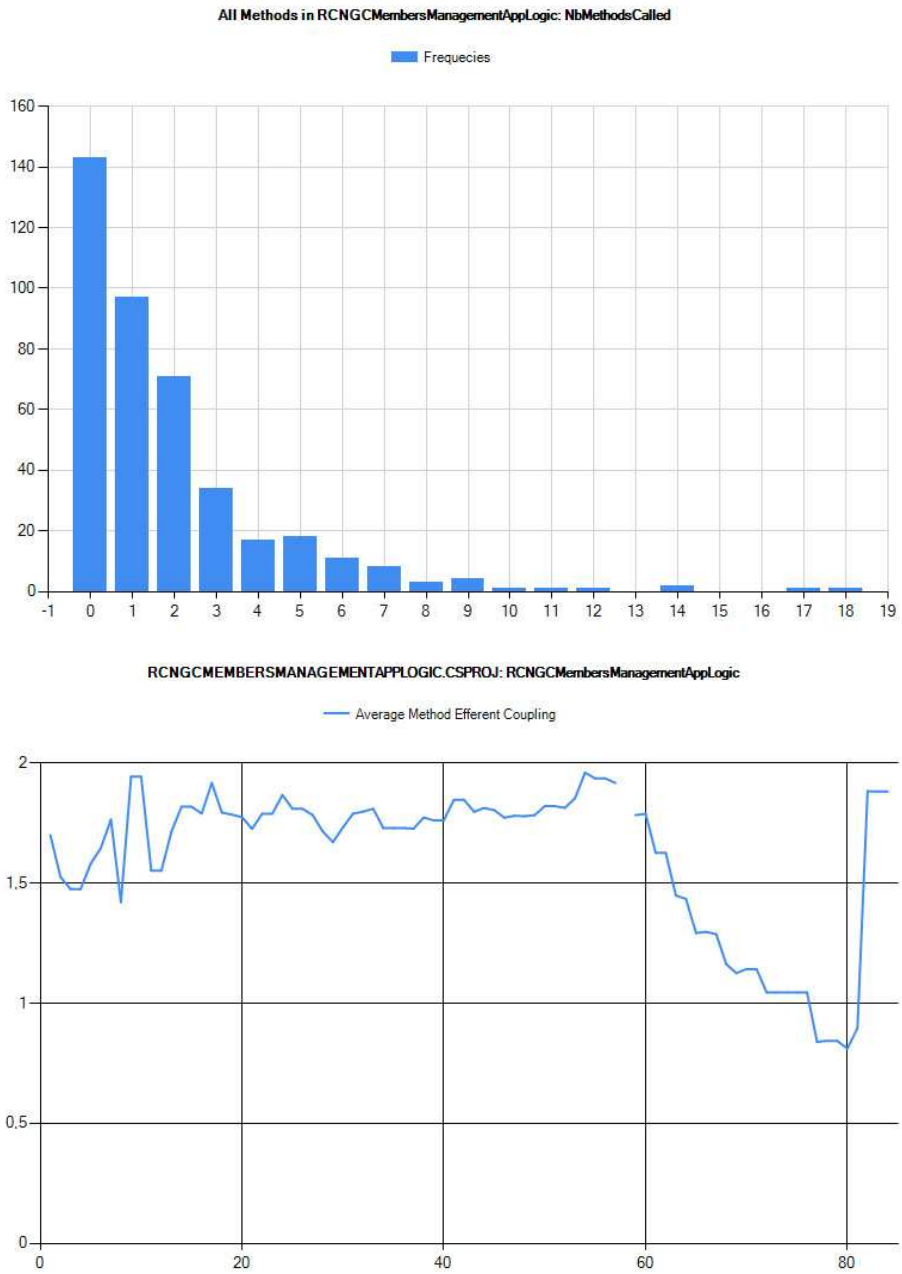


Figura 10-19: Acoplamiento Eferente en los Métodos. Histograma y evolución de la media

Con respecto al acoplamiento aferente, podemos ver que se mantiene controlado. Podemos observar la misma desviación anteriormente comentada en el histórico, corregida al final. Casi todos los métodos cumplen el Principio de Responsabilidad única, al llamar a muy pocos otros métodos. Solo en algunos casos puntuales se al acoplamiento aferente se eleva, y una vez revisados, no resultan alarmantes.

A nivel de clases

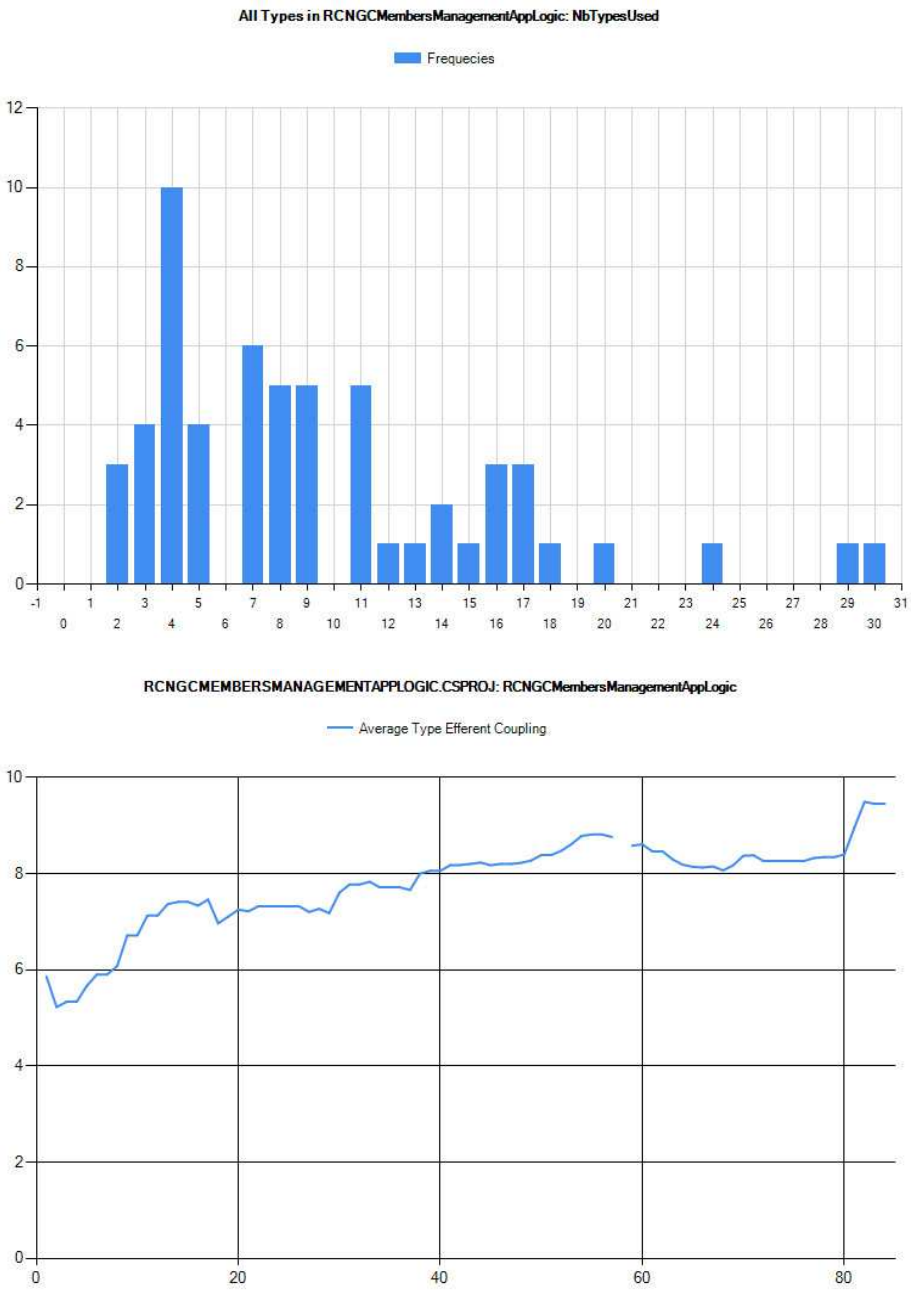


Figura 10-20: Acoplamiento eferente en los tipos. Histograma y evolución de la media

Vemos que esta métrica crece de forma natural, con una tendencia fija, a medida que el código evoluciona, pues van creciendo el número de clases implementadas y la relación entre ellas, especialmente teniendo en cuanta que todas ellas forman parte de un mismo dominio: la facturación. La media se mantiene en niveles relativamente bajos, con una moda de 4, por lo que podemos concluir que la calidad del código es correcto según esta métrica.

Cabe destacar la existencia de cuatro valores altos, que se corresponden con las clases 'controladoras' que figuran en lo alto del diagrama de dependencias, las mismas que

tenían un valor de 0 en la métrica aferente. Observamos, por tanto, una característica de las clases 'controladoras': bajo acoplamiento aferente y alto acoplamiento eferente. Un comportamiento esperado y deseable

Asociación entre Clases

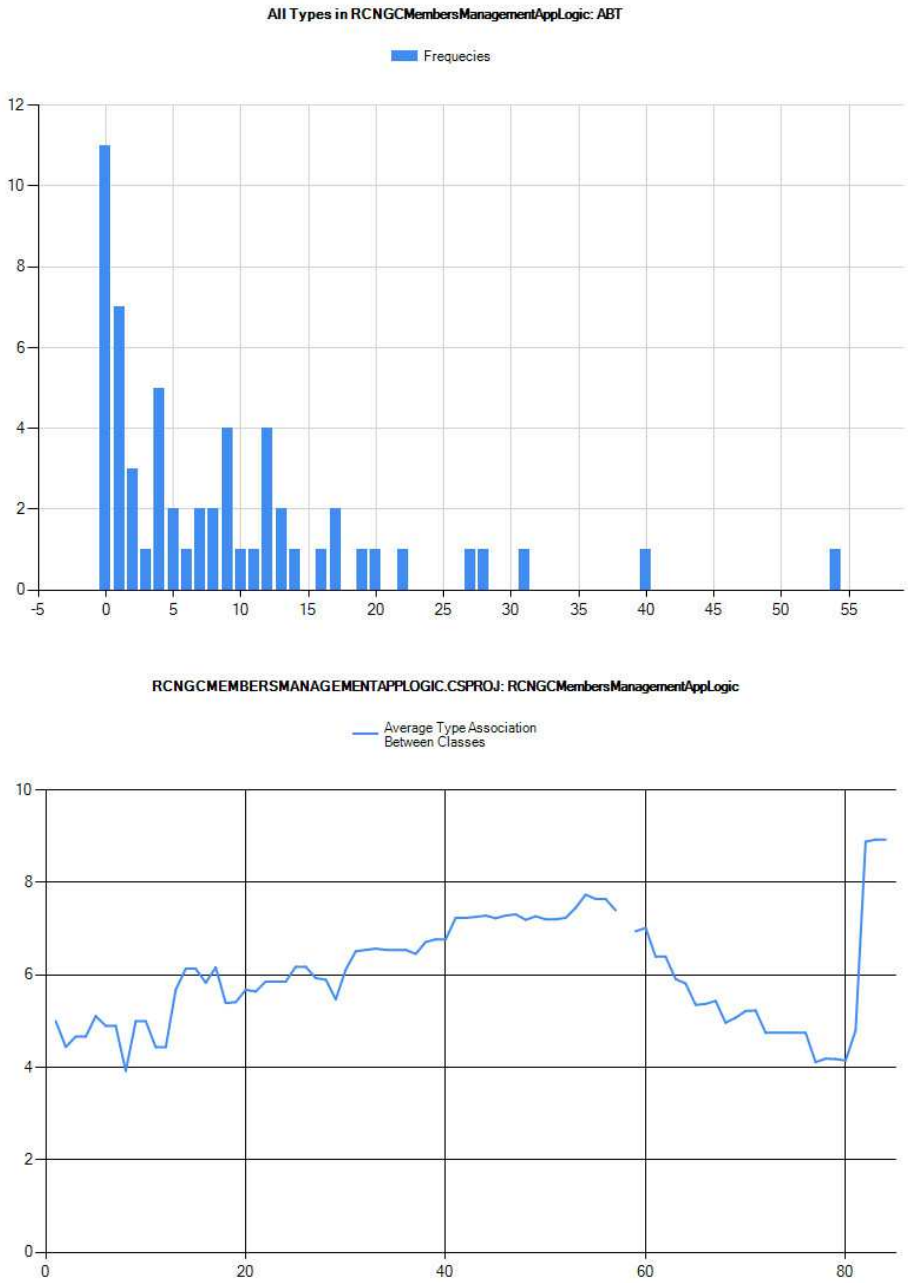


Figura 10-21: Asociación entre Clases. Histograma y evolución de la media

La asociación entre clases, para un tipo dado, mide el número de miembros de otras clases que usa directamente en el cuerpo de sus métodos. Es similar al acoplamiento aferente, pero, con una diferencia, cuenta los miembros usados.

Pongamos un ejemplo: Los métodos de la clase A usan a 5 métodos de la clase B. Por tanto, A usa B (acoplamiento aferente =1), y usa 5 métodos (ABC=5). Por tanto, la métrica ABC produce valores más altos. Si esta diferencia es muy grande, podría ser un indicativo de que hay un alto acoplamiento debido a que la clase A está entrando en demasiados detalles de implementación de la clase B.

En nuestro caso no se aprecia tal excesivo acoplamiento, aunque durante la implementación de las clases SEPA se produjo una mejora importante (aunque no real) en la métrica.

10.5.3. Abstracción e Inestabilidad

Hay dos métricas que se relacionan en el código:

- Abstracción: ¿Cuan abstracto es mi ensamblado? Dicha métrica se computa como el ratio entre clases abstractas sobre el total de clases

  Abstracción = Total clases abstractas / Total de clases

- Inestabilidad: ¿Cuánta resistencia opone mi clase al cambio? Dicha métrica se computa como el ratio del acoplamiento eferente con respecto al acoplamiento total. Una clase que solo tenga acoplamiento eferente depende totalmente del exterior, es inestable, y no opondría demasiada 'resistencia', pues, al nadie depender de ella, podemos modificarla con libertad.

  I = Ce/(Ca+Ce)

Por si solas estas dos métricas no son demasiado relevantes, pero al combinarlas, podemos observar estados interesantes en los ensamblados:

- Un ensamblado muy inestable (que depende demasiado del exterior) no tiene problemas si es poco abstracto

- Un ensamblado muy abstracto debe ser todo lo estable e independiente posible, según el principio OCP.

Uniendo ambos extremos tenemos la llamada ´Línea de Secuencia Principal'

Línea de Secuencia Principal: A + I =1

Podemos ubicar cada ensamblado como un punto en este diagrama, usando su inestabilidad como coordenada X y su abstracción como coordenada Y. La distancia de este punto, calculada sobre dicha línea, será su indicativo de calidad.

En nuestro caso, estos los valores de abstracción, inestabilidad y distancia de la secuencia principal de nuestros ensamblados:

Ensamblado	A	I	Dist.
RCNGCMembersManagemmentApLogic	0.98	0.07	0.04
RCNGCISO20022CustomerDebitInitiation	0.9	0	0.07
RCNGCMembersManagementMocks	1	0	0
ExtensionMethods	0.92	0	0.06

Si los representamos en un gráfico, se ubicarán en la esquina inferior derecha, e 'zona verde'

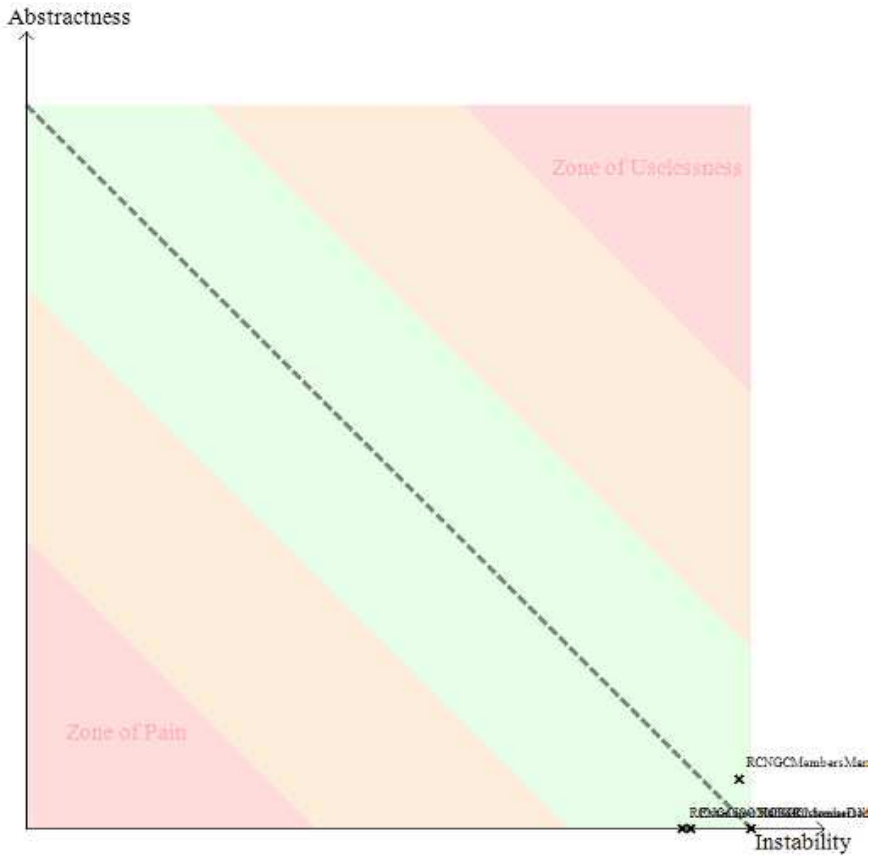


Figura 10-22: Abstracción e Inestabilidad

11. CONCLUSIONES

Llegado a este punto, cabe recapitular, y, quizá la mejor manera es reflexionar sobre si, en él, se han cumplido los objetivos propuestos.

Objetivo principal

Con respecto al objetivo principal del proyecto, que era evaluar la calidad del código en proyectos que usen TDD, la respuesta, en mi opinión, muy clara, es SI.

- Hemos argumentado que el código generado mediante TDD es de mayor calidad por pura y simple necesidad. Unas clases que se alejan del paradigma SOLID y que no siguen a pies juntillas el Clean Code se convierten en un verdadero tormento para el desarrollador que continuamente refactorizar un código en constante crecimiento.

- Hemos podido comprobar empíricamente en este proyecto la verdad de la anterior afirmación. Los resultados de todas las métricas han sido positivos, y se han mantenido así de principio a fin, durante todo el desarrollo.

- En los desarrollos con metodologías no ágiles, donde la arquitectura viene predefinida y el implementador recibe una 'caja negra' de la que solo le exigen que cumpla con cierta interfaz pública, estática e inmutable, éste se puede permitir cualquier licencia, siempre que su código esté debidamente comentado, claro está. Mientras cumpla con unos mínimos de eficiencia, su código puede permitirse ser de cuestionable calidad. Hasta que tenga que hacer cambios…

Tan solo una puntualización. Me ha quedado también claro el primer valor del manifiesto ágil, que insiste en la necesitad de disponer de programadores altamente cualificados en el agilismo. Sin ellos, o, al menos, un par de ellos en el equipo que echen una mano al resto (la figura del 'facilitador' en SCRUM), el progreso se hace terriblemente lento, al enfrentarse a cada paso, en cada ciclo TDD, a decisiones importantes difíciles de tomar para una persona de baja o mediana competencia. Es necesario un gran dominio del lenguaje de programación en cuestión, así como de refactorización y diseño orientado a objetos. Para un servidor, bastante oxidado en el desarrollo, el desarrollo ha resultado tremendamente laborioso, aunque he de confesar que ha resultado muy grato comprobar a través de las métricas que la calidad final del mismo era bastante aceptable.

Objetivos Especificos

Hablábamos al comienzo del proyecto, en su enfoque, de los objetivos específicos, y quiero recordarlos, porque, para mí, son la verdadera esencia del proyecto. En particular el primero de ellos

- **Formar:**

  - **Es un proceso de aprendizaje:** Definitivamente, sí. En el prólogo comenté cómo mi tutor me embarcó en este viaje, como me convención de que iba a aprender un montón de cosas. Que iba a descubrir una forma nueva, radicalmente diferente de programar, y que al final se lo agradecería. Y he aprendido. Mucho. Y se lo agradezco infinitamente.

  - **Es un estudio didáctico:** A lo largo del proyecto he descubierto cosas que me han asombrado y me han ilusionado tanto como a mi tutor. Así que aquí me tienen haciendo proselitismo ágil ☺. Con este proyecto, en esta memoria, intento transmitir a todos las cosas que he aprendido, que como antes dije, son muchas. Espero que sirva de ayuda, de guía en los primeros pasos a aquellos que quieran empezar a adentrarse en las aguas del agilismo. Y, para los que estén mirando desde la orilla, indecisos, les entren ganas de echarse un buen chapuzón.

- **Implementar:** Quizá la parte más descuidada, pues se centra mucho más en los dos anteriores objetivos. Pero he de decir que me siento muy orgulloso de mi pequeño NDepend Metrics Reporter, de lo útil y potente que ha resultado ser. Por otra parte, a mi me ha venido de perlas el desarrollar un módulo en C# para enviar los ficheros de adeudos directos según las especificaciones del SEPA, que en mi empresa tengo que dejar lista toda la migración para el 1 de Febrero del año que viene. ¿Alguien más con el mismo problema?

  Tienen libre acceso al código en http://github.com/kikogomez ☺

Anexo 1

Historias de Usuario

Diagramas de Clases

Diagramas de Dependencias

RCNGC MEMBERS MANAGER

Historias de Usuario

Para la adquisición de requisitos, y como intermediario entre el 'poduct owner' y los desarrolladores finales, se han usado las historias de usuario de BDD, con la herramienta de SpecFlow

Añadir miembros de Club

Como dijimos, es muy simple. Tan solo vamos a añadir miembros básicos. La información de facturación se incluye posteriormente.

```
Feature: Add members
       In order to manage the club members
       As a administrative assistant
       I want to add new members to the system

Scenario Outline: Second surname is optional but given name and first
surname are mandatory
       Given These names <GivenName>, <FirstSurname>, <SecondSurname>
       When I process the names
       Then The name is considered <valid>

Scenarios:
| GivenName   | FirstSurname     | SecondSurname | valid     |
| "Francisco" | "Gomez-Caldito" | "Viseas"      | "valid"   |
| "Francisco" | "Gomez-Caldito" | ""            | "valid"   |
| "Francisco" | ""              | "Viseas"      | "invalid" |
| ""          | "Gomez-Caldito" | "Viseas"      | "invalid" |
| "Francisco" | ""              | ""            | "invalid" |
| ""          | "Gomez-Caldito" | ""            | "invalid" |
| ""          | ""              | "Viseas"      | "invalid" |
| ""          | ""              | ""            | "invalid" |

Scenario: The members ID are consecutive
       Given The current memberID sequence number is 2
       When I add a new member
       Then The current memberID sequence number is 3

Scenario: Up to 99999 members
       Given The current memberID sequence number is 100000
       When I add a new member
       Then The new member is not created
```

Anexo Figura 1: Historia de usuario: Añadir miembros del club

Generar Facturas

Se facturan a los socios los productos comprados y los servicios usados. Cada factura puede referirse a uno o más conceptos, y se contempla los impuestos (que pueden ser diferentes para cada concepto) y posibles descuentos.

```
Feature: Generating invoices
       In order to bill the club members
       As an administrtative assistant
       I want to generate invoices

Background:
       Given Last generated InvoiceID is "INV2013000023"

       Given A Club Member
       | MemberID | Name      | FirstSurname   | SecondSurname |
       | 00001    | Francisco | Gomez-Caldito | Viseas        |

       Given This set of taxes
       | Tax Type            | Tax Value |
       | No IGIC             | 0         |
       | IGIC Reducido 1     | 2.75      |
       | IGIC Reducido 2     | 3.00      |
       | IGIC General        | 7.00      |
       | IGIC Incrementado 1 | 9.50      |
       | IGIC Incrementado 2 | 13.50     |
       | IGIC Especial       | 20.00     |

       Given These services
       | Service Name                | Default Cost | Default Tax  |
       | Rent a kajak                | 50.00        | IGIC General |
       | Rent a katamaran            | 100.55       | IGIC General |
       | Rent a mouring              | 150.00       | IGIC General |
       | Full Membership Monthly Fee | 79.00        | No IGIC      |

       Given These products
       | Product Name    | Default Cost | Default Tax  |
       | Pennant         | 10.00        | IGIC General |
       | Cup             | 15.00        | IGIC General |
       | Member ID Card | 1.50         | No IGIC      |

Scenario: Generate an invoice for a service charge
       Given The member uses the club service "Rent a kajak"
       When I generate an invoice for the service
       Then An invoice is created for the cost of the service: 53.50
       And The invoice state is "To be paid"
       And The invoice is assigned to the Club Member

Scenario: Generate an invoice for a sale
       Given The member buys a "Pennant"
       When I generate an invoice for the sale
       Then An invoice is created for the cost of the sale: 10.70
       And The invoice state is "To be paid"
       And The invoice is assigned to the Club Member
```

Anexo Figura 2: Historia de usuario: Añadir Facturas (Parte 1)

```
Scenario: The invoices ID must allways be consecutive
       Given The member uses the club service "Rent a kajak"
       When I generate an invoice for the service
       Then The generated Invoice ID should be "INV2013000023"
       Then The next invoice sequence number should be 24

Scenario: Up to 999999 invoices in a year
       Given Last generated InvoiceID is "INV2013999999"
       Given The member uses the club service "Rent a mouring"
       When I try to generate an invoice for the service
       Then The application doesn't accept more than 999999 invoices in the
year

Scenario: Generate an invoice for multiple transactions with one tax type
       Given This set of service charge transactions
       | Units | Service Name   | Description            | Unit Cost | Tax         | Discount |
       | 1     | Rent a kajak   | Rent a kajak for one day | 50.00     | IGIC General | 0        |
       | 2     | Rent a mouring | Mouring May-June         | 150.00    | IGIC General | 0        |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 374.50
       And The invoice state is "To be paid"

Scenario: Generate an invoice for multiple transactions with different tax
type
       Given This set of service charge transactions
       | Units | Service Name          | Description      | Unit Cost | Tax         | Discount |
       | 1     | Membership Monthly Fee | Monthly Fee June | 79.00     | No IGIC      | 0        |
       | 2     | Rent a mouring         | Mouring May-June | 150.00    | IGIC General | 0        |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 400.00
       And The invoice state is "To be paid"

Scenario: Discounts on transactions must be applied before taxes
       Given This set of service charge transactions
       | Units | Service Name   | Description      | Unit Cost | Tax         | Discount |
       | 1     | Rent a mouring | Mouring May-June | 150.00    | IGIC General | 20       |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 128.40
       And The invoice state is "To be paid"

Scenario: Rounding: Round to two decimals Away From Zero
       Given This set of service charge transactions
       | Units | Service Name   | Description      | Unit Cost | Tax         | Discount |
       | 1     | Rent a mouring | Mouring May-June | 150.00    | IGIC General | 15       |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 136.43
       And The invoice state is "To be paid"

Scenario: Rounding: First calculate discount on unit, then round, then tax
unit, then round, then sum units
       Given This set of service charge transactions
       | Units | Service Name    | Description           | Unit Cost | Tax         | Discount |
       | 2     | Rent a katamaran | Rent a katamaran 2 days | 100.55    | IGIC General | 15       |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 182.90
       And The invoice state is "To be paid"
```

Anexo Figura 3: Historia de usuario: Añadir Facturas (Parte 2)

```
Scenario: Transactions can have differnt cost and tax than default service
ones
       Given This set of service charge transactions
       | Units | Service Name     | Description             | Unit Cost | Tax     | Discount |
       | 1     | Rent a katamaran | Renta a katamaran 2 days | 90        | No IGIC | 0        |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 90.00
       And The invoice state is "To be paid"

Scenario: We can mix services charges and sales in a single invoice
       Given This set of service charge transactions
       | Units | Service Name     | Description             | Unit Cost | Tax          | Discount |
       | 2     | Rent a katamaran | Renta a katamaran 2 days | 50        | IGIC General | 0        |
       | 2     | Rent a mouring   | Mouring May-June         | 150.00    | IGIC General | 20       |
       Given This set of sale transactions
       | Units | Product Name   | Description       | Unit Cost | Tax          | Discount |
       | 1     | Cup            | Blue Cup          | 10        | IGIC General | 0        |
       | 1     | Member ID Card | ID Card Reprinted | 1.50      | No IGIC      | 50       |
       When I generate an invoice for this/these transaction/s
       Then An invoice is created for the cost of the service: 375.25
       And The invoice state is "To be paid"
```

Anexo Figura 4: Historia de usuario: Añadir Facturas (Parte 3)

Generar facturas proforma

Se generan a veces facturas pro-forma (presupuestos) para algunos servicios

```
Feature: Generating pro forma invoices
       In order to give estimates to the club members
       As an administrtative assistant
       I want to generate pro forma invoices

Background:
       Given Last generated pro forma invoice ID is "PRF2013000023"

       Given A Club Member
       | MemberID | Name      | FirstSurname  | SecondSurname |
       | 00001    | Francisco | Gomez-Caldito | Viseas        |

       Given This set of taxes
       | Tax Type            | Tax Value |
       | No IGIC             | 0         |
       | IGIC Reducido 1     | 2.75      |
       | IGIC Reducido 2     | 3.00      |
       | IGIC General        | 7.00      |
       | IGIC Incrementado 1 | 9.50      |
       | IGIC Incrementado 2 | 13.50     |
       | IGIC Especial       | 20.00     |

       Given These services
       | Service Name               | Default Cost | Default Tax  |
       | Rent a kajak               | 50.00        | IGIC General |
       | Rent a katamaran           | 100.55       | IGIC General |
       | Rent a mouring             | 150.00       | IGIC General |
       | Full Membership Monthly Fee | 79.00        | No IGIC      |

       Given These products
       | Product Name    | Default Cost | Default Tax  |
       | Pennant         | 10.00        | IGIC General |
       | Cup             | 15.00        | IGIC General |
       | Member ID Card | 1.50         | No IGIC      |


Scenario: Generating a pro forma invoice for a set of service charges and
sales
       Given This set of service charge transactions
       | Units | Service Name     | Description             | Unit Cost | Tax          | Discount |
       | 2     | Rent a katamaran | Renta a katamaran 2 days | 50        | IGIC General | 0        |
       | 2     | Rent a mouring   | Mouring May-June         | 150.00    | IGIC General | 20       |
       Given This set of sale transactions
       | Units | Product Name    | Description       | Unit Cost | Tax          | Discount |
       | 1     | Cup             | Blue Cup          | 10        | IGIC General | 0        |
       | 1     | Member ID Card | ID Card Reprinted | 1.50      | No IGIC      | 50       |
       When I generate a pro forma invoice for this/these transaction/s
       Then A pro forma invoice is created for the cost of the service:
375.25
```

Anexo Figura 5: Generar Facturas Proforma

```
Scenario: A proforma invoice has no bill associated
       Given This set of service charge transactions
       | Units | Service Name     | Description             | Unit Cost | Tax          | Discount |
       | 2     | Rent a katamaran | Renta a katamaran 2 days | 50        | IGIC General | 0        |
       | 2     | Rent a mouring   | Mouring May-June         | 150.00    | IGIC General | 20       |
       When I generate a pro forma invoice for this/these transaction/s
       Then No bills are created for a pro forma invoice

Scenario: The invoice detail of a pro forma invoice can be edited
       Given I generate a pro forma invoice for this/these transaction/s
       | Units | Service Name   | Description      | Unit Cost | Tax          | Discount |
       | 2     | Rent a mouring | Mouring May-June | 150.00    | IGIC General | 0        |
       When I change the invoice detail to these values
       | Units | Service Name   | Description      | Unit Cost | Tax          | Discount |
       | 2     | Rent a mouring | Mouring May-June | 150.00    | IGIC General | 20       |
       Then The pro forma invoice is modified reflecting the new value:
256.80
```

Anexo Figura 6: Historia de usuario: Generar factura proforma (Parte 2)

Modificar Facturas

Una factura creada puede anularse mediante una factura rectificativa.

```
Feature: Manage Invoices
       In order to control the debt
       As a an administrtative assistant
       I want to manage the created invoices

Background:
       Given Last generated InvoiceID is "INV2013000023"

       Given A Club Member
       | MemberID | Name      | FirstSurname  | SecondSurname |
       | 00001    | Francisco | Gomez-Caldito | Viseas        |

       Given This set of taxes
       | Tax Type            | Tax Value |
       | No IGIC             | 0         |
       | IGIC Reducido 1     | 2.75      |
       | IGIC Reducido 2     | 3.00      |
       | IGIC General        | 7.00      |
       | IGIC Incrementado 1 | 9.50      |
       | IGIC Incrementado 2 | 13.50     |
       | IGIC Especial       | 20.00     |

       Given These services
       | Service Name               | Default Cost | Default Tax  |
       | Rent a kajak               | 50.00        | IGIC General |
       | Rent a katamaran           | 100.55       | IGIC General |
       | Rent a mouring             | 150.00       | IGIC General |
       | Full Membership Monthly Fee | 79.00        | No IGIC      |

       Given These products
       | Product Name   | Default Cost | Default Tax  |
       | Pennant        | 10.00        | IGIC General |
       | Cup            | 15.00        | IGIC General |
       | Member ID Card | 1.50         | No IGIC      |


Scenario: In some special cases, an invoice can be cancelled
       Given I have an invoice for the service "Rent a kajak"
       When I cancel the invoice
       Then The invoice state is "Cancelled"
       And All the pending bills are marked as Cancelled
       And The bill total amount to be paid is 0
       And An amending invoice is created for the negative value of the
original invoice: -53.50
       And The taxes devolution (-3.50) is separated from the base cost
devolution (-50)
       And The amending invoice ID is the same than the original invoice
with different prefix: "AMN2013000023"   Then The new member is not created
```

Anexo Figura 7: Historia de usuario: Modificar facturas

Gestionar recibos

Se generan recibos para cada factura. Estos pueden pagarse, quedar impagados al cumplir su vencimiento, cancelarse, o renegociarse a un nuevo vencimiento o en pagos aplazados.

```
Feature: Manage bills
       In order charge my invoices
       As an administrative assistant
       I want reate and manage bills for my invoices

Background:
       Given Last generated InvoiceID is "INV2013000023"

       Given A Club Member with a default Payment method
       | MemberID | Name      | FirstSurname  | SecondSurname | Default Payment method |
Spanish IBAN Bank Account          | Direct Debit Reference Number |
       | 00001    | Francisco | Gomez-Caldito | Viseas        | Direct Debit           |
IBAN ES68 1234 5678 0612 3456 7890 | 12345                                 |

       Given This set of taxes
       | Tax Type             | Tax Value |
       | No IGIC              | 0         |
       | IGIC Reducido 1      | 2.75      |
       | IGIC Reducido 2      | 3.00      |
       | IGIC General         | 7.00      |
       | IGIC Incrementado 1  | 9.50      |
       | IGIC Incrementado 2  | 13.50     |
       | IGIC Especial        | 20.00     |

       Given These services
       | Service Name                | Default Cost | Default Tax  |
       | Rent a kajak                | 50.00        | IGIC General |
       | Rent a katamaran            | 100.55       | IGIC General |
       | Rent a mouring              | 150.00       | IGIC General |
       | Full Membership Monthly Fee | 79.00        | No IGIC      |

       Given These products
       | Product Name    | Default Cost | Default Tax  |
       | Pennant         | 10.00        | IGIC General |
       | Cup             | 15.00        | IGIC General |
       | Member ID Card  | 1.50         | No IGIC      |

Scenario: A single bill is automatically created for a new invoice
       Given The member uses the club service "Rent a kajak"
       When I generate an invoice for the service
       Then An invoice is created for the cost of the service: 53.50
       And A single bill To Collect is generated for the total amount of the
invoice: 53.50
       And The bill ID is "INV2013000023/001"
       And By default no payment method is associated to bill
```

Anexo Figura 8: Historia de usuario: Gestionar recibos (Parte 1)

```
Scenario: No bills are created for a pro forma invoice
       Given The member uses the club service "Rent a kajak"
       When I generate an pro-forma invoice for the service
       Then A pro-forma invoice is created for the cost of the service:
53.50
       And No bills are created for a pro-forma invoice

Scenario: A bill can be renegotiated into instalments
       Given I have an invoice with cost 650 with a single bill with ID
"INV2013000023/001"
       When I renegotiate the bill "INV2013000023/001" into three
instalments: 200, 200, 250 to pay in 30, 60 and 90 days with agreement terms
"Payment Agtreement"
       Then The bill "INV2013000023/001" is marked as renegotiated
       And The renegotiated bill "INV2013000023/001" has associated the
agreement terms "Payment Agtreement" to it
       And A bill with ID "INV2013000023/002" and cost of 200 to be paid in
30 days is created
       And The new bill "INV2013000023/002" has associated the agreement
terms "Payment Agtreement" to it
       And A bill with ID "INV2013000023/003" and cost of 200 to be paid in
60 days is created
       And The new bill "INV2013000023/003" has associated the agreement
terms "Payment Agtreement" to it
       And A bill with ID "INV2013000023/004" and cost of 250 to be paid in
90 days is created
       And The new bill "INV2013000023/004" has associated the agreement
terms "Payment Agtreement" to it

Scenario: I can assign an specific expected payment method for a single bill
       Given I have an invoice with cost 650 with a single bill with ID
"INV2013000023/001"
       When I assign to be paid with a direct debit
       Then The new payment method is correctly assigned

Scenario: A bill to collect is paid in cash
       Given I have an invoice with some bills
       And I have a bill to collect in the invoice
       When The bill is paid in cash
       Then The bill state is set to "Paid"
       And The bill payment method is set to "Cash"
       And The bill payment date is stored
       And The bill amount is deduced form the invoice total amount

Scenario: A bill to collect is paid by bank transfer
       Given I have an invoice with some bills
       And I have a bill to collect in the invoice
       When The bill is paid by bank transfer
       Then The bill state is set to "Paid"
       And The bill payment method is set to "Bank Transfer"
       And The transferor account is stored
       And The transferee account is stored
       And The bill payment date is stored
       And The bill amount is deduced form the invoice total amount
```

Anexo Figura 9: Historia de usuario: Gestionar recibos (Parte 2)

```
Scenario: A bill to collect is paid by direct debit
       Given I have an invoice with some bills
       And I have a bill to collect in the invoice
       When The bill is paid by direct debit
       Then The bill state is set to "Paid"
       And The bill payment method is set to "Direct Debit"
       And The direct debit initiation ID is stored
       And The bill payment date is stored
       And The bill amount is deduced form the invoice total amount

Scenario: All the bills of an invoice are paid
       Given I have an invoice with some bills
       When All the bills are paid
       Then The invoice state is set as "Paid"

Scenario: A bill is past due date
       Given I have an invoice with some bills
       And I have a bill to collect in the invoice
       When The bill is past due date
       Then The bill is marked as "Unpaid"
       And The invoice containing the bill is marked as "Unpaid"

Scenario: A bill with an associated agreement is past due date
       Given I have an invoice with some bills with agreements
       And I have a bill to collect in the invoice with a payment agreement
       When The bill is past due date
       Then The bill is marked as "Unpaid"
       And The invoice containing the bill is marked as "Unpaid"
       And The associated payment agreement is set to "NotAcomplished" for
all bills involved on the agreement
       And The associated payment agreement is set to "NotAcomplished" for
the invoice

Scenario: A bill due date can be extended
       Given I have an invoice with some bills
       And I have a bill to collect in the invoice
       When I renew the due date
       Then The new due date is assigned to the bill

Scenario: A past due bill due date can be renewed
       Given I have an invoice with some bills
       And I have a bill to collect in the invoice
       And The bill is past due date
       When I renew the due date
       Then The new due date is assigned to the bill
       And The bill is marked as "ToCollect"
       And If there are no other bills marked as "Unpaid" the invoice is
marked "ToBePaid"
```

Anexo Figura 10: Historia de usuario: Gestionar recibos (Parte 3)

Gestionar la información de pago de los socios

Los socios pueden pagar en efectivo o mediante emisión de adeudo domiciliado. Para poder mandar un recibo han de firmar una orden de domiciliación. Un socio puede tener varias órdenes de domiciliación

```
Feature: Manage Members Billing Information
       In order be able to bill my club members
       As a administrative assistant
       I want to manage my members billing registered data

Background:

       Given A Club Member
       | MemberID | Name      | FirstSurname  | SecondSurname |
       | 00001    | Francisco | Gomez-Caldito | Viseas        |

       Given These Direct Debit Mandates
       | DirectDebitInternalReferenceNumber | RegisterDate | IBAN                       |
       | 2345                               | 2013/11/20   | ES6812345678061234567890 |
       | 2346                               | 2013/11/30   | ES3011112222003333333333 |

       Given These Account Numbers
       | IBAN                       |
       | ES6812345678061234567890 |
       | ES3011112222003333333333 |

Scenario: I can change the member default payment method
       Given I have a member
       And The member has associated cash as payment method
       When I set direct debit as new payment method
       Then The new payment method is correctly updated

Scenario: I can assign a new direct debit mandate to a member
       Given I have a member
       And The direct debit reference sequence number is 5000
       When I add a new direct debit mandate to the member
       Then The new direct debit mandate is correctly assigned
       And The new direct debit reference sequence number is 5001

Scenario: I can change the account number associated to a direct debit
       Given I have a member
       And I have a direct debit associated to the member
       When I change the account number of the direct debit
       Then The account number is correctly changed
       And The old account number is stored in the account numbers history
```

Anexo Figura 11: Historia de usuario: Gestionar la información de pago de los socios

Gestionar la información del Club como emisor de recibos

Como emisor de recibos el Club establece contratos de cedente de recibos con las entidades bancarias

```
Feature: Manage Creditor Info
       In order to charge the bills to my club members
       As a Club Manager
       I want to manage all my info as a direct debit creditor

Background:

       Given My creditor info is
       | NIF       | Name                              |
       | G35008770 | Real Club Náutico de Gran Canaria |

Scenario: Create a creditor agent
       Given I have a bank
       When I register the bank as a creditor agent
       Then The creditor agent is correctly created

Scenario: Register a direct debit initiation contract
       Given I have a creditor agent
       When I register a contract data
       Then The contract is correctly registered

Scenario: Register more than one direct debit initiation contract
       Given I have a direct debit initiation contract registered
       When I register a second contract data
       Then The contract is correctly registered

Scenario: I change the bank account for my contract
       Given I have a direct debit initiation contract
       When I change the creditor account to "ES8721002222002222222222"
       Then The contract account is correctly updated to
"ES8721002222002222222222"

Scenario: I change remove a direct debit initiation contract
       Given I have a direct debit initiation contract registered with
bussines code "333"
       When I remove the contract "333"
       Then The contract "333" is correctly removed
```

Anexo Figura 12: Historia de usuario: Gestionar la información del club como emisor de recibos

Generar remesas de recibos domiciliados

Una vez elegido un contrat, se genera una remesa, a la que se van añadiendo los reibos, para, finalmente crear el mensaje SEPA ISO20022 pain.008.001.02

```
Feature: Generating Direct Debit Remmitances
        In order to charge the bills to my club members
        As an administrative assistant
        I want to generate direct debit remitances to bank

Background:

        Given My Direct Debit Initiation Contract is
        | NIF       | Name                              | BIC         | CreditorAgentName
| LocalBankCode | CreditorBussinesCode | CreditorAccount          |
        | G35008770 | Real Club Náutico de Gran Canaria | CAIXESBBXXX | CAIXABANK
| 2100          | 777                  | ES5621001111301111111111 |

        Given These Club Members
        | MemberID | Name      | FirstSurname  | SecondSurname | Reference | Account             |
BIC         |
        | 00001    | Francisco | Gomez-Caldito | Viseas        | 1234      | 01821111601111111111 |
BBVAESMMXXX |
        | 00002    | Pedro     | Perez         | Gomez         | 1235      | 21001111301111111111 |
CAIXESBBXXX |

        Given These bills
        | MemberID | TransactionConcept            | Amount |
        | 00001    | Cuota Mensual Octubre 2013    | 79     |
        | 00002    | Cuota Mensual Octubre 2013    | 79     |
        | 00002    | Cuota Mensual Noviembre 2013 | 79     |

Scenario: Create a new direct debit remmitance
        Given I have a I have a direct debit initiation contract
        When I generate a new direct debit remmitance
        Then An empty direct debit remmitance is created

Scenario: Create an empty group of direct debit payments
        Given I have a will send the payments using "COR1" local instrument
        When I generate an empty group of direct debit payments
        Then An empty group of direct debit payments using "COR1" is
generated

Scenario: Create a Direct Debit Transaction from a bill as specified by a
member direct debit mandate
        Given I have a member
        And The member has a bill
        And The member has a Direct Debit Mandate
        When I generate Direct Debit Transaction
        Then The direct debit transaction is correctly created

Scenario: Add a second bill to a direct debit transaction
        Given I have a direct debit with 1 bill and amount of 79
        When I add a new bill with amount of 79
        Then The direct debit transaction is updated with 2 bills and amount
of 158
```

Anexo Figura 13: Historia de usuario: generar remesas de recibos domiciliados

```
Scenario: Add a direct debit transaction to an empty group of payments
       Given I have a direct debit with 1 bill and amount of 79
       And I have an empty group of payments
       When I add the direct debit transaction to the group of payments
       Then The group of payments is updated with 1 direct debit and total
amount of 79

Scenario: Add a second direct debit transaction to a group of payments
       Given I have a group of payments with 1 direct debit transaction and
amount of 79
       When I add a new direct debit transaction with amount of 79
       Then The group of payments is updated with 2 direct debit and total
amount of 158

Scenario: Add a group payments to a direct debit remmitance
       Given I have an empty direct debit remmitance
       And I have a group of payments with 1 direct debit transaction and
amount of 79
       When I add the group to the direct debit remittance
       Then The direct debit remittance is updated with 1 direct debit and
total amount of 79

Scenario: Generating SEPA ISO20022 XML CustomerDirectDebitInitiation Message
form a Direct Debit Remmitance
       Given I have a prepared Direct Debit Remmitance
       When I generate de SEPA ISO200022 XML CustomerDirectDebitInitiation
message
       Then The message is correctly created
```

Anexo Figura 14: Historia de usuario: Generar remesas de recibos domiciliados (Parte 2)

Gestionar los números de cuenta

Es muy importante la correcta adquisición de números de cuenta de la anteriores bases de datos. Se ha de permitir importar números incompletos, pero se han de validar todas las nuevas entradas. Los números antiguos pueden ser códigos de banco/sucursal/cuenta o CCC completos. Los nuevos pueden ser, además, IBAN.

```
Feature: Manage account numbers
       In order to create direct debits
       As an administrative assistant
       I want to process account numbers
       I want to store old incomplete bank account fields from previuos
database
       I want to accept only valid accounts if CCC or IBAN are provided

Scenario: When I provide a valid bank account it is stored and CCC and IBAN
is created
       Given This bank account "1234", "5678", "06", "1234567890"
       When I process the bank account
       Then the bank account is considered "valid"
        And the bank account is "stored"
        And The CCC "1234567806123456789O" is created
        And The spanish IBAN code "ES6812345678061234567890" is created

Scenario: When I provide an invalid bank account it is stored but no CCC nor
IBAN are created
       Given This bank account "1234", "5678", "05", "1234567890"
       When I process the bank account
       Then the bank account is considered "invalid"
        But the bank account is "stored"
        And The CCC "" is created
        And The spanish IBAN code "" is created

Scenario: When I provide an incomplete bank account it is stored but no CCC
nor IBAN are created
       Given This bank account "", "5678", "05", "1234567890"
       When I process the bank account
       Then the bank account is considered "invalid"
        But the bank account is "stored"
        And The CCC "" is created
        And The spanish IBAN code "" is created

Scenario: When I provide a too long bank account it is not stored
       Given This bank account "1234", "5678", "06", "12345678901111"
       When I process the bank account
       Then the bank account is considered "invalid"
        And the bank account is "not stored"

Scenario: When I provide a valid CCC it is stored, bank account fields are
created, and IBAN is created
       Given This CCC "1234567806123456789O"
       When I process the CCC
       Then the CCC is considered "valid"
        And the CCC is "stored"
        And the bank account "1234", "5678", "06", "1234567890" is created
        And The spanish IBAN code "ES6812345678061234567890" is created
```

Anexo Figura 15: Historia de usuario: Gestionar los números de cuenta

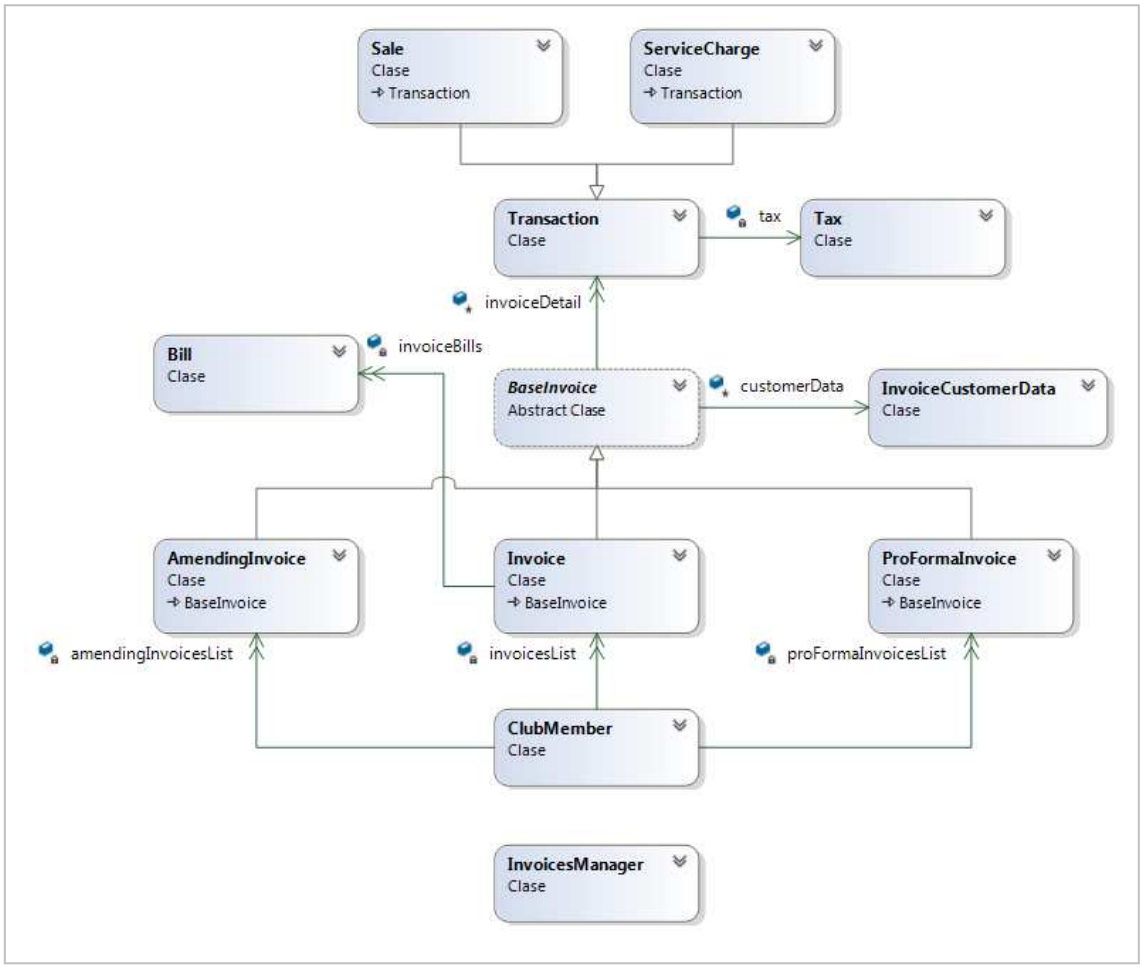
```
Scenario: When I provide a valid spanish IBAN it is stored, bank account
fields are created, and CCC is created
       Given This IBAN "ES6812345678061234567890"
       When I process the IBAN
       Then the IBAN is considered "valid"
        And the IBAN is "stored"
        And the bank account "1234", "5678", "06", "1234567890" is created
        And The CCC "12345678061234567890" is created

Scenario: When I provide an ivalid CCC no info is stored nor created
       Given This CCC "12345678051234567890"
       When I process the CCC
       Then the CCC is considered "invalid"
        And the CCC is "not stored"

Scenario: When I provide a invalid spanish IBAN no info is stored nor
created
       Given This IBAN "ES6812345678051234567890"
       When I process the IBAN
       Then the IBAN is considered "invalid"
        And the IBAN is "not stored"

Scenario Outline: Theese are the results when processing theese bank
accounts
       Given This bank account <Bank>, <Office>, <ControlDigit>,
<AccountNumber>
       When I process the bank account
       Then the bank account is considered <valid>
        And the bank account is <stored>
        And The CCC <CCC> is created
        And The spanish IBAN code <IBAN> is created

Scenarios:
| Bank   | Office | ControlDigit | AccountNumber  | valid     | stored
| CCC                    | IBAN                          |
| "1234" | "5678" | "06"         | "1234567890"   | "valid"   | "stored"
| "12345678061234567890" | "ES6812345678061234567890" |
| "1234" | "5678" | "05"         | "1234567890"   | "invalid" | "stored"
| ""                     | ""                            |
| ""     | ""     | "05"         | "1234567890"   | "invalid" | "stored"
| ""                     | ""                            |
| "1234" | "5678" | "06"         | "1234/56-0"    | "invalid" | "stored"
| ""                     | ""                            |
| "1234" | "5678" | "06"          | "123456789011" | "invalid" | "no stored"
| ""                     | ""                            |
```

Anexo Figura 16: Historia de usuario: Gestionar los números de cuenta (Parte 2)

Diagramas de Clases

Socios y facturación básica

- Cada socio incluye una lista de facturas que se les han emitido (normales, pro forma o rectificativas), derivadas de una clase base.
- Las facturas tienen un detalle de conceptos *('Transactions'*), que pueden ser ventas o cargos por servicios.
- Al mismo tiempo, cada factura puede pagarse de una vez o en recibos aplazados.
- Las facturas se gestionan desde una clase controladora *'InvoicesManager'*

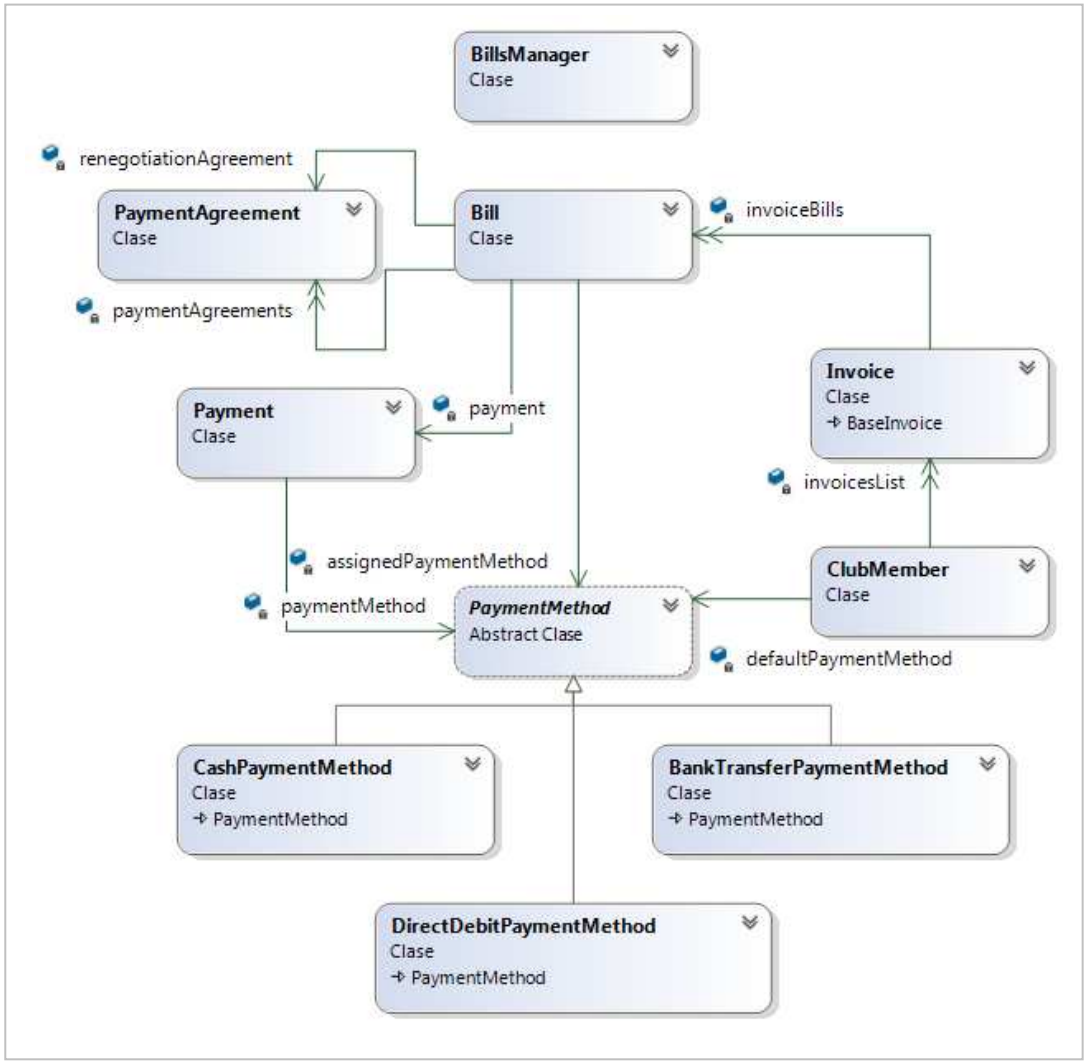


Anexo Figura 17: Diagramas de clases: Las facturas

Pago de recibos

En los recibos:

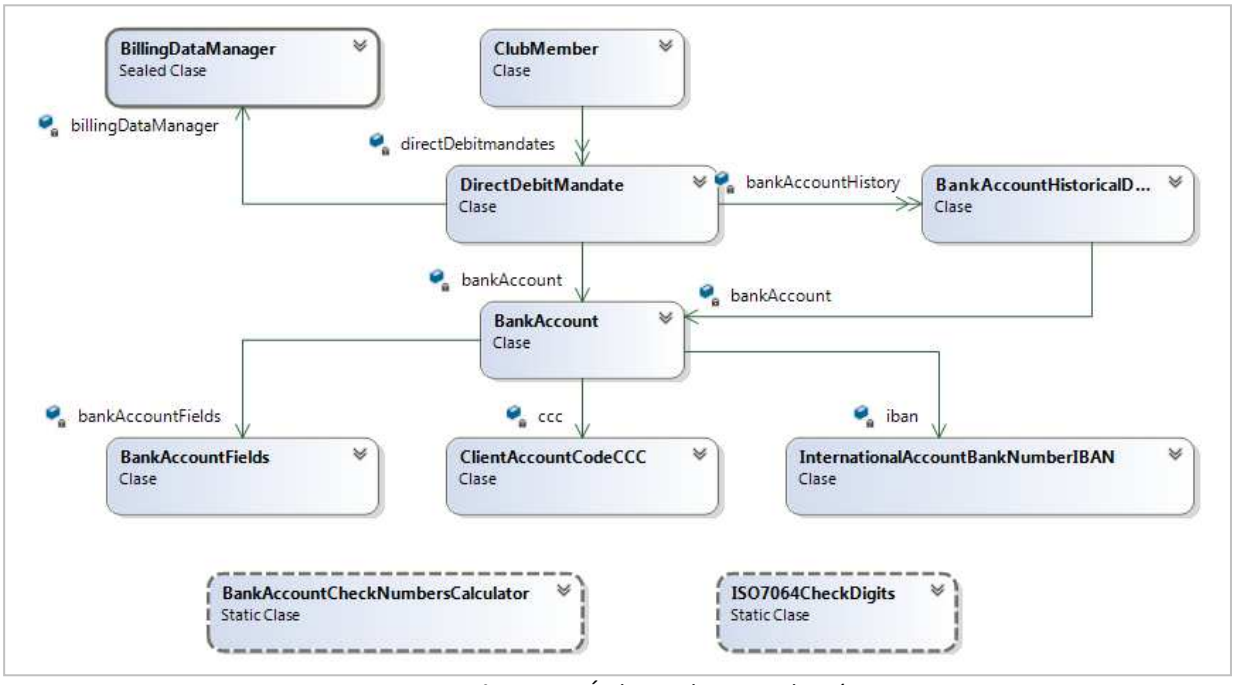
- Se le puede asociar una forma de pago por defecto
- Puede tener acuerdos de renegociación
- Cuando se paga, se le asocia un 'pago'.
- Se gestionan desde la clase controladora *'BillsManager'*.



Anexo Figura 18: Diagramas de clases: Las facturas

Órdenes de Domiciliación

Cada miembro puede tener una serie de órdenes de domiciliación, cada una con una cuenta asociada, que se puede expresar en varios formatos. Dicha cuenta puede cambiarse, pero se mantiene un histórico

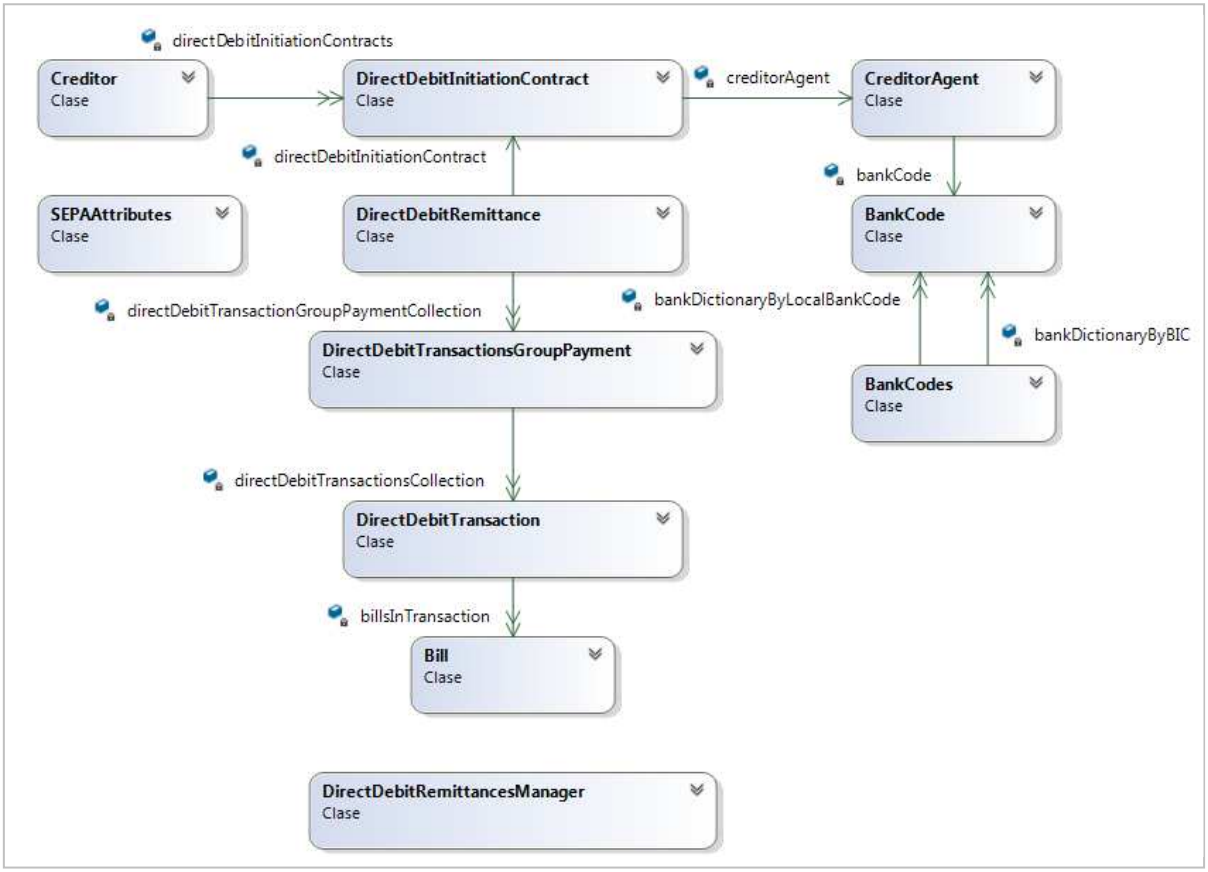


Anexo Figura 19: Órdenes de Domiciliación

Generación de Remesas al Banco

Se pueden generar remesas de recibos para enviar al banco.

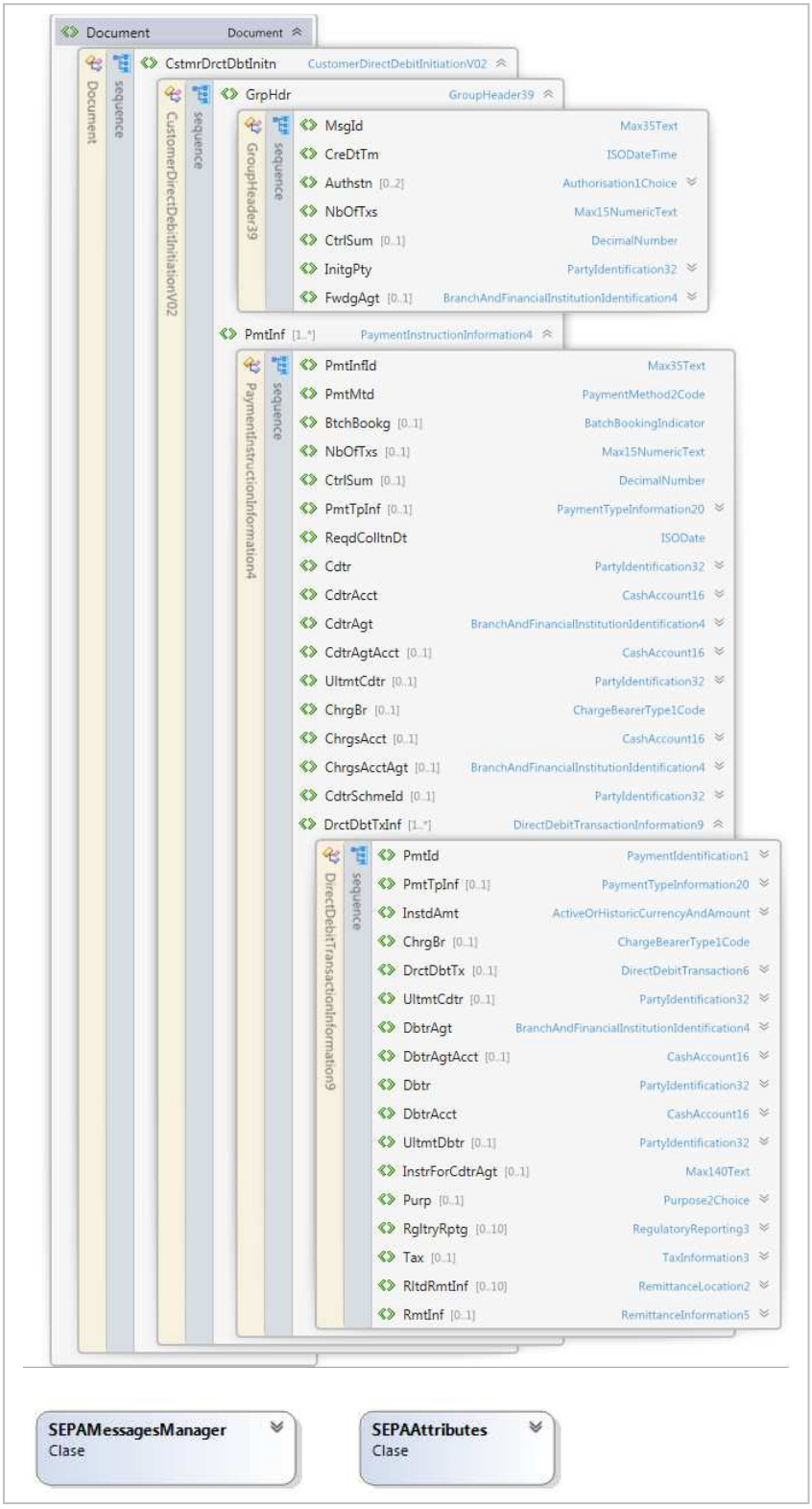
- Como emisor podemos firmar cuantos contratos queramos con nuestro banco
- Cada remesa que generemos estará asociada a uno de estos contratos
- Cada remesa esta formada por uno o más grupos de pagos, cada uno de los cuales puede tener uno o más adeudos directos, cada uno de los cuales puede ser el pago de uno o más recibos
- La generación de la remesa se realiza desde la clase controladora *'DirectDebitRemittancesManager'*.



Anexo Figura 20: Generación de remesas de recibos

Envío de remesas al banco

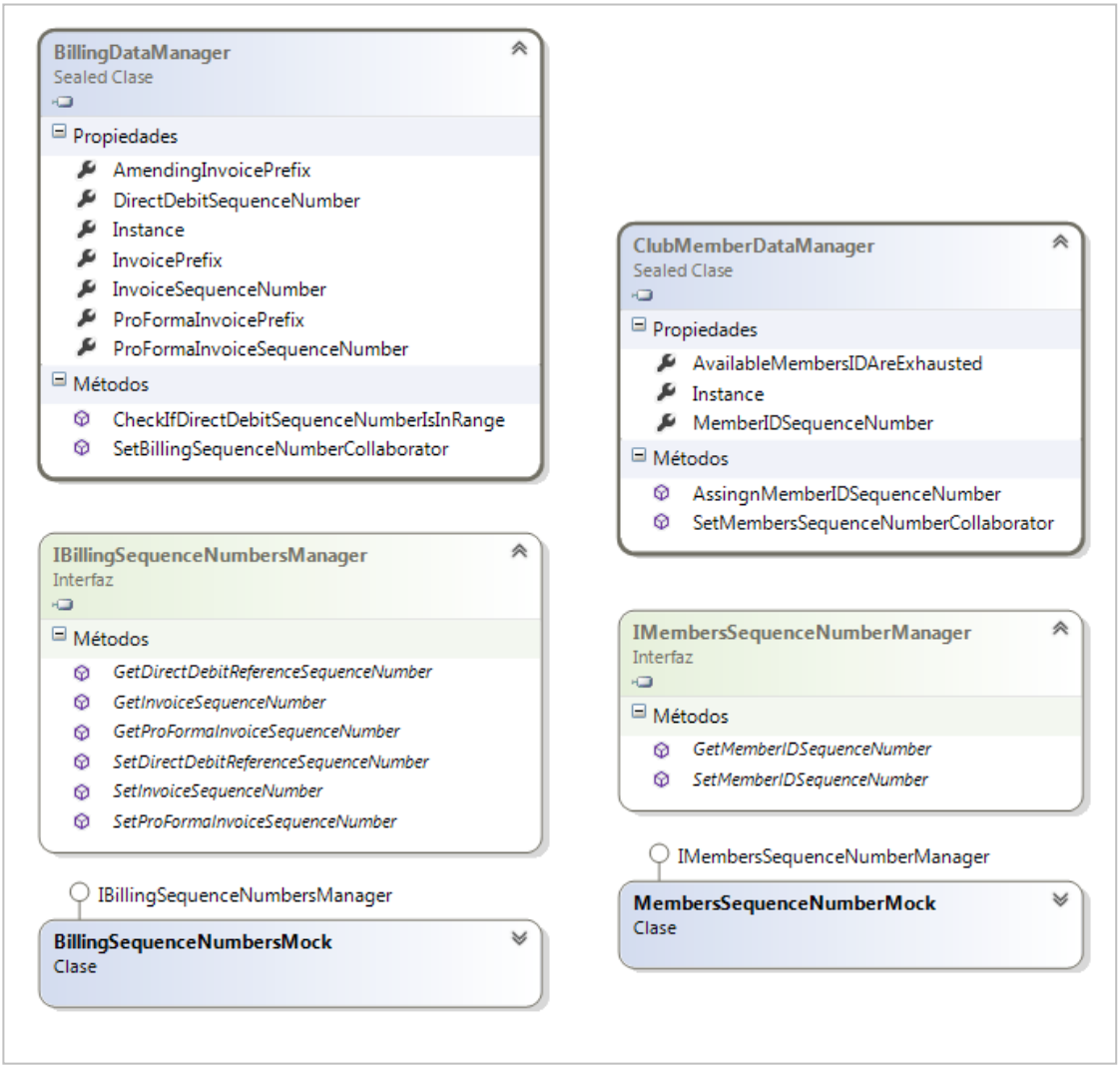
Una vez generada, la remesa se puede enviar al banco usando el formato establecido por el SEPA ISO20022 pain.008.001.02 mediante la clase *'SEPAMessagesManager'*. La especificación ISO es muy extensa (unas 60 clases). Mostramos el esquema más externo.



Anexo Figura 21: Envío de remesas al banco usando SEPA ISO20022 pain.008.001.02

Mocking

Para el presente proyecto se han implementado un par de clases para ayudar en el 'mocking' de los tests.

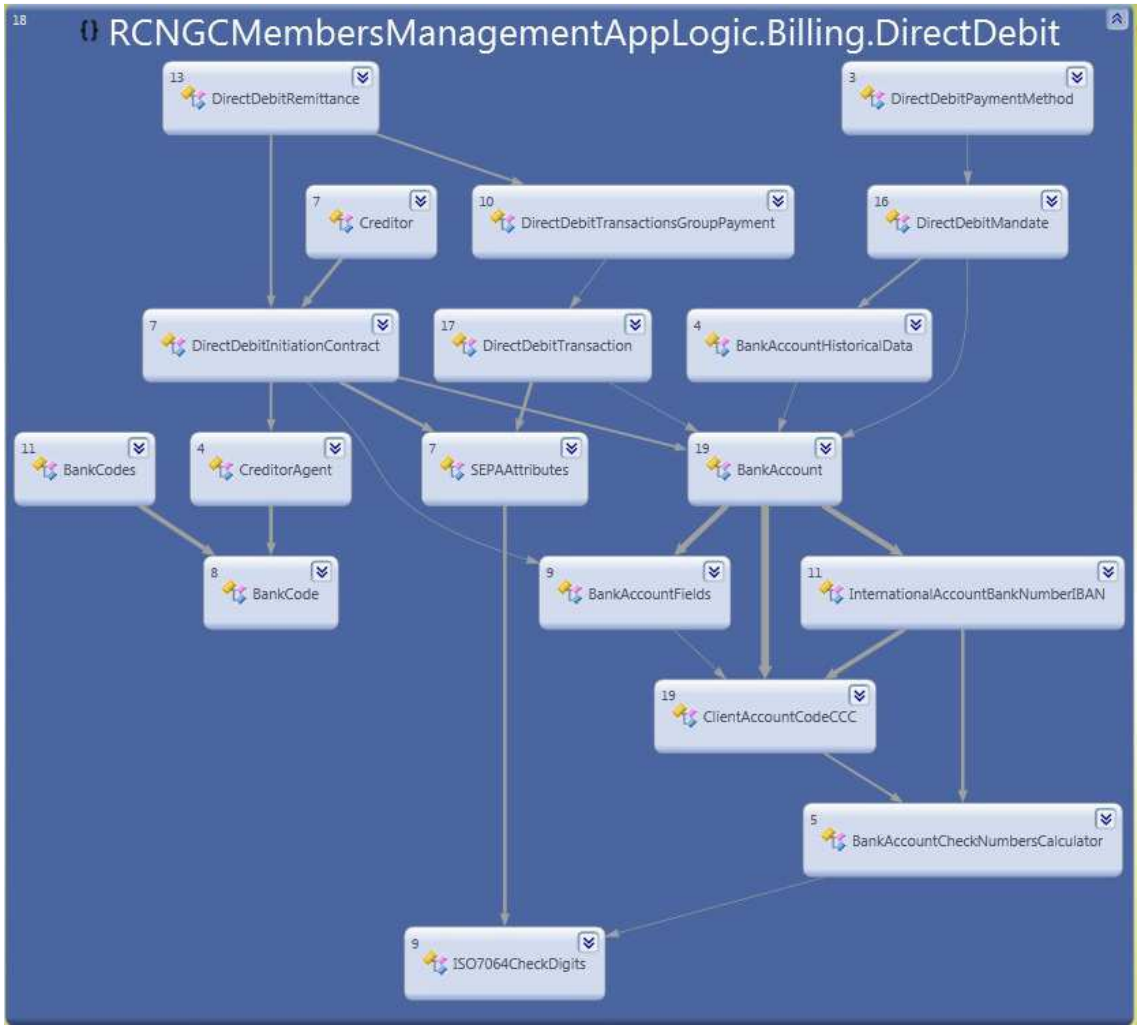


Anexo Figura 22: Diseño de 'Mocking'

Diagramas de dependencias

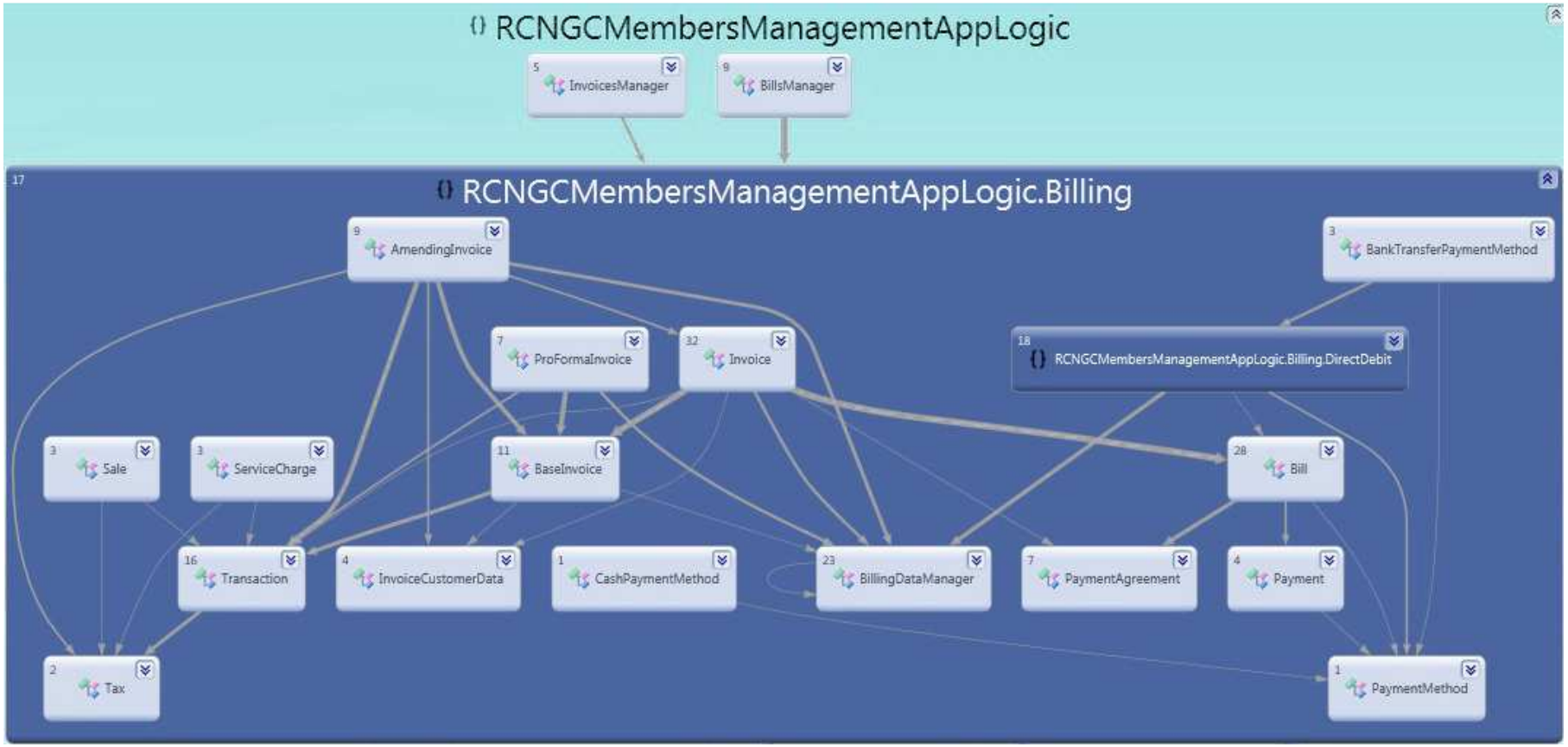
Como podemos observar en los siguientes diagramas, las clases están correctamente estructuradas y no se observan dependencias circulares en los grafos de dependencias.

Adeudos directos



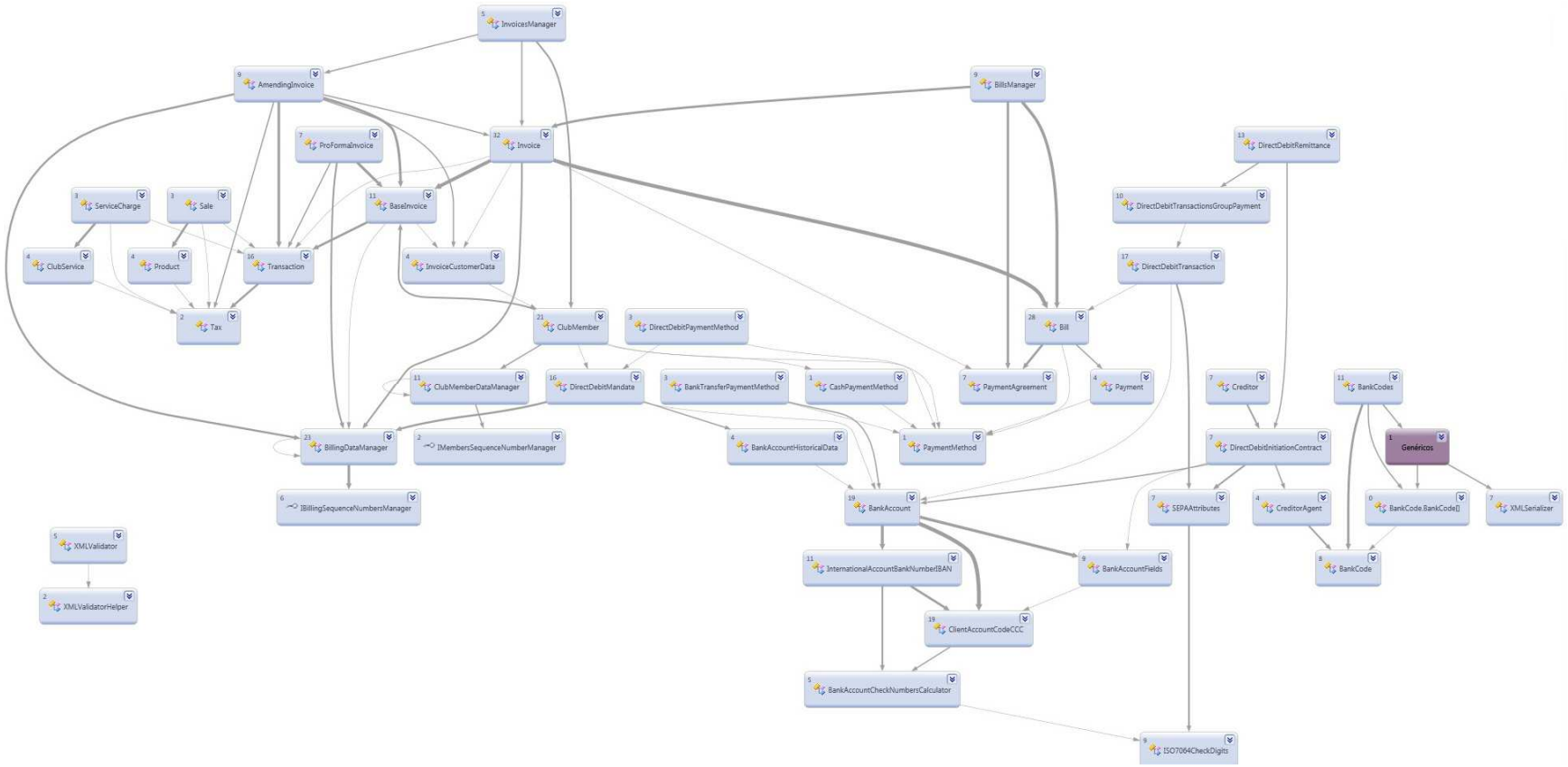
Anexo Figura 23: Dependencias en las clases de adeudo directo

Facturacion



Anexo Figura 24: Dependencias en las clases de facturación

Diagrama general de dependencias



Anexo Figura 25: Diagrama general de dependencias

BIBLIOGRAFÍA

[1] Abelson, Harold. Sussman, Gerald J. (1984) *Structure and Interpretation of Computer Programs.* First Edition – Mit Press

[2] Astels, David R. (2003). *Test-Driven Development: A Practical Guide.* Prentice Hall

[3] Beck, Kent (2000) *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional

[4] Beck, Kent (2003). Test-Driven Development: By Example. Addison-Wesley Professional

[5] Blé, Carlos. (2010). *Diseño Ágil con TDD*

[6] Chambers, Sean (2009) *31 Days Refactoring*

[7] Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional

[8] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John (1994) *Design Patterns: Elements of reusable Object-Oriented Software.* Addison-Wesley Professional

[9] Henderson-Sellers (1995) *A BOOK of Object-Oriented Knowledge (2nd. Edition)*. Prentice Hall.

[10] Martin, Robert C. (2008) *Design Principles and Design Patterns*. www.objectmentor.com

[11] Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall

[12] Meyer, Bertrand (1988) *Object-Oriented Software Construction.* Prentice hall

[13] North, Dan (2006) Introducing BDD . Dan North & Associated

[14] Takeuchi, Hirokata – Nonaka, Ikujiro (1986). *The New Product Development Game*

[15] Yourdon, E. (1982). *Análisis Estructurado Moderno*. Prentice Hall Iberoamericana

REFERENCIAS

[1] LOPD
http://www.agpd.es/portalwebAGPD/canaldocumentacion/legislacion/estatal/common/pdfs/LOPD_consolidada.pdf

[2] Artículo 10 del Convenio de Berna para la Protección de las Obras Literarias y Artísticas
http://www.wipo.int/treaties/es/ip/berne/trtdocs_wo001.html#P153_28927

[3] Directiva 2001/29/CE relativa a la armonización de ciertos derechos de autor
http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32001L0029:ES:HTML

[4] SEPA – Single Euro Payments Area

[5] European Council page on SEPA
http://ec.europa.eu/internal_market/payments/sepa/

[6] PSD – Directive on Payment Services
http://ec.europa.eu/internal_market/payments/framework/

[7] Ley de Servicios de Pago - Ley 16/2009, de 13 de noviembre
http://www.boe.es/boe/dias/2009/11/14/pdfs/BOE-A-2009-18118.pdf

[8] EPC – European Payments Council
http://www.europeanpaymentscouncil.eu/

[9] European Central Bank page on SEPA
http://www.ecb.europa.eu/paym/sepa/

[10] Portal sobre la SEPA en español
http://www.sepaesp.es

[11] ISO20022 - Universal financial industry message scheme
http://www.iso20022.org/

[12] ISO Technical Commitee TC68 Financial Services
http://www.iso.org/iso/iso_technical_committee?commid=49650

[13] Página del EPC sobre Adeudos Directos SEPA
http://www.europeanpaymentscouncil.eu/content.cfm?page=sepa_direct_debit_%28sdd%29

[14] Página del Banco de España sobre Adeudos Directos SEPA
http://www.sepaesp.es/sepa/es/secciones/instrumentos/adeudosdirectos/Adeudos_directos_basicos.html

[15] SEPA Core Direct Debit Scheme Rulebook (EPC016-06)
http://www.europeanpaymentscouncil.eu/knowledge_bank_detail.cfm?documents_id=553

[16] ISO 20022 Message Definition Report - Payments Maintenance 2009 - Edition September 2009
http://www.iso20022.org/documents/general/Payments_Maintenance_2009.zip

[17] ISO 20022 Customer-to-Bank Message Usage Guide - Customer Credit Transfer Initiation, Customer Direct Debit Initiation and Payment Status Report - Version 3.0

http://www.swift.com/assets/corporates/documents/our_solution/implementing_your_project_2009_iso20022_usage_guide.pdf

[18] SWIFT - Society for Worldwide Interbank Financial Telecommunication
http://www.swift.com

[19] Sepa Core Direct Debit Scheme Customer-to-Bank Implementation Guidelines (EPC130-08)
http://www.europeanpaymentscouncil.eu/knowledge_bank_detail.cfm?documents_id=544

[20] ISO20022 Message Archive Page
http://www.iso20022.org/message_archive.page

[21] ISO 20022 (pain) - Third version of the Payment Initiation messages
http://www.iso20022.org/message_archive.page#PaymentsInitiation3

[22] Ficha de Adeudos Sepa
http://www.sepaesp.es/f/websepa/secciones/Instrumentos/FICHA_DE_ADEUDOS_DIRECTOS_SEPA.pdf

[23] CSB-Cuaderno 19 – Marzo 2008
http://empresas.bankia.es/Ficheros/CMA/ficheros/Norma19Marzo2008.PDF
Nota: Este documento no aparece publicado en ninguna de las páginas de los organismos anteriores, sino que está disponible a través de las páginas para empresa de las diferentes entidades bancarias. El enlace provisto en esta memoria está sacado directamente a través de una búsqueda en Internet, perteneciendo en este caso a La Caixa.

[24] Migración a SEPA de los Adeudos Domiciliados Españoles
http://www.aebanca.es/cs/groups/public/documents/publicaciones/00-201300427.pdf

[25] AEB – Asociación Española de Banca
http://www.aebanca.es

[26] CECA – Confederación Española de Cajas de Ahorros
http://www.ceca.es

[27] UNACC – Unión Nacional de Cooperativas de Crédito
http://www.unacc.com/

[28] Órdenes en Formato ISO 20022 para la emisión de Adeudos Directos SEPA – Esquema Básico – Guía de Implantación – Noviembre 2012
http://empresa.lacaixa.es/deployedfiles/empresas/Estaticos/pdf/Transferenciasyficheros/Cuaderno_XML_Emision_Adeudos_SDD_Core_Noviembre_2012.pdf
Nota: Este documento no aparece publicado en ninguna de las páginas de los organismos anteriores, sino que está disponible a través de las páginas para empresa de las diferentes entidades bancarias. El enlace provisto en esta memoria está sacado directamente a través de una búsqueda en Internet, perteneciendo en este caso a La Caixa.

[29] Juego del Teléfono Roto
http://es.wikipedia.org/wiki/Tel%C3%A9fono_descompuesto

[30] Teoría de la Información
http://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_informaci%C3%B3n

[31] "Eat Poop You Cat" – Board Game Geek
http://boardgamegeek.com/boardgame/30618/eat-poop-you-cat

[32] Waterfall Model
http://en.wikipedia.org/wiki/Waterfall_model

[33] Métrica 3 – Ministerio de Hacienda y Administraciones Públicas de España
http://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html

[34] ¿Qué es el SCRUM?
http://www.proyectosagiles.org/que-es-scrum

[35] Manifiesto Ágil
http://agilemanifesto.org/iso/es/

[36] Software Craftmanship – WIKIPEDIA
http://en.wikipedia.org/wiki/Software_craftsmanship

[37] Los 12 principios del agilismo
http://agilemanifesto.org/iso/es/principles.html

[38] BDD – Behaviour Driven Development
http://en.wikipedia.org/wiki/Behavior-driven_development

[39] An Introduction to Aceptance Tests
http://agile.techwell.com/articles/weekly/introductory-acceptance-test

[40] SOLID – OOP Principles. Robert C. Martin "Uncle Bob"
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[41] Code Metrics – Ndepend
http://www.ndepend.com/metrics.aspx

[42] Agile Methodologies Survey Result (PDF) – Shine Thechnologies
http://www.shinetech.com/attachments/104_ShineTechAgileSurvey2003-01-17.pdf

[43] Ambler, Scott (3 August 2006) - Survey Says: Agile Works in Practice
http://www.drdobbs.com/architecture-and-design/survey-says-agile-works-in-practice/191800169

[44] Software Quality Management – WIKIPEDIA
http://en.wikipedia.org/wiki/Software_quality_management

[45] Hirota Takeuchi – Wikipedia
http://en.wikipedia.org/wiki/Hirotaka_Takeuchi

[46] Ikujiro Nonaka – Wikipedia
http://en.wikipedia.org/wiki/Ikujiro_Nonaka

[47] Ken SchWaber – Wikipedia
http://en.wikipedia.org/wiki/Ken_Schwaber

[48] Dan North & Associates
http://dannorth.net/

[49] Behaviour-Driven Development - Wikipedia
http://en.wikipedia.org/wiki/Behavior-driven_development

[50] What's in a story? Dan North & Associated
http://dannorth.net/whats-in-a-story/

[51] Mocks aren't stubs – Martin Fowler
http://martinfowler.com/articles/mocksArentStubs.html

[52] Code Coverage – WIKIPEDIA
http://en.wikipedia.org/wiki/Code_coverage

[53] Design Patterns – oodesign.com
http://www.oodesign.com/

[54] Design Principles and Design Patterns – Robert C. Martin
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[55] The Principles of OOD – Robert C. Martin
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[56] Single Responsability Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/srp.pdf

[57] The Open/Closed Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/ocp.pdf

[58] The Liskov Sustitution Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/lsp.pdf

[59] The Interface Segregation Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/isp.pdf

[60] The Dependency Inversion Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/dip.pdf

[61] Barbara Liskov – Wikipedia
http://en.wikipedia.org/wiki/Barbara_Liskov

[62] Inversion of Control Containers and the Dependency Injection Pattern – Martin Fowler
http://martinfowler.com/articles/injection.html

[63] A tutorial on Service Locator patter with implementation – Y. Sujan - The Code Project
http://www.codeproject.com/Articles/597787/A-tutorial-on-Service-locator-pattern-with-impleme

[64] Decoupling example in C# - Stack Exchange
http://stackoverflow.com/questions/226977/what-is-loose-coupling-please-provide-examples

[65] Structure and Interpretation of Computer Programs – Abelson & Sussman
http://mitpress.mit.edu/sicp/full-text/book/book.html

[66] Production of Large Computer Programs - Herbert D. Bennington
http://sunset.usc.edu/csse/TECHRPTS/1983/usccse83-501/usccse83-501.pdf

[67] Pair programming – WIKIPEDIA
http://en.wikipedia.org/wiki/Pair_programming

[68] The Repository Pattern Example in C# - Remondo Blog
http://www.remondo.net/repository-pattern-example-csharp/

[69] Microsoft DreamSpark
https://www.dreamspark.com/

[70] Sistema de Control de Versiones - WIKIPEDIA
http://es.wikipedia.org/wiki/Control_de_versiones

[71] Git-SCM
http://git-scm.com/

[72] GitHub
https://github.com/

[73] Git Graphic user Interfaces
http://git-scm.com/downloads/guis

[74] Ignorar ficheros en un repositorio - .gitIgnore
https://help.github.com/articles/ignoring-files

[75] Msysgit
http://code.google.com/p/msysgit/

[76] Twitter
https://twitter.com/

[77] SourceForge.Net
http://sourceforge.net/

[78] Cucumber
http://cukes.info/

[79] Lenguaje de programación Ruby
https://www.ruby-lang.org/es/

[80] SpecFlow
http://www.specflow.org/

[81] Gherkin
https://github.com/cucumber/cucumber/wiki/Gherkin

[82] SpecRun
http://specrun.com/

[83] SpecLog
http://www.speclog.net/

[84] NuGet
http://www.nuget.org/

[85] SpecFlow Configuration
http://www.specflow.org/documentation/Configuration/

[86] Gherkin languaje – Cucumber – GitHub
https://github.com/cucumber/cucumber/wiki/Gherkin

[87] SpecFlow – Sharing data between Bindings
http://www.specflow.org/documentation/Sharing-Data-between-Bindings/

[88] SpecFlow – POCO classes injection
http://www.specflow.org/documentation/Context-Injection/

[89] SpecFlow – ScenarioContext.Current
http://www.specflow.org/documentation/ScenarioContext.Current/

[90] SpecFlow – Scoped Bindings
http://www.specflow.org/documentation/Scoped-Bindings/

[91] Cucumber – Tags
https://github.com/cucumber/cucumber/wiki/Tags

[92] SpecFlow Documentation
http://www.specflow.org/documentation/

[93] TextContext Class – MSTest
http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.testcontext.aspx

[94] Assert Class – MSTest
http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert.aspx

[95] Using the Assert classes in MSTest
http://msdn.microsoft.com/en-us/library/ms182530.aspx

[96] TestDriven.Net
http://www.testdriven.net/

[97] NDepend
http://www.ndepend.com/

[98] NDepend Code Metrics
http://www.ndepend.com/Features.aspx#Metrics

[99] NDepend arquitecture and dependences Explorer
http://www.ndepend.com/Features.aspx#DependenciesView

[100] NDepend Code Query Linq
http://www.ndepend.com/Features.aspx#CQL

[101] NDepend Trend monitoring
http://www.ndepend.com/Features.aspx#Trend

[102] NDepend Diagrams
http://www.ndepend.com/Features.aspx#Diagrams

[103] NDepend Code Diff
http://www.ndepend.com/doc_vs_diff.aspx

[104] Google Page Rank Algorithm
http://www.sirgroane.net/google-page-rank/

[105] Code metrics on coupling, dead code, design flaws and re-ingeneering – Patrick Smacchia – CodeBetter.com
http://codebetter.com/patricksmacchia/2008/02/15/code-metrics-on-coupling-dead-code-design-flaws-and-re-engineering/

[106] Layering, the Level metric and the Discourse of Method – Patrick Smacchia – CodeBetter.Com
http://codebetter.com/patricksmacchia/2008/02/10/layering-the-level-metric-and-the-discourse-of-method/

[107] Single Responsability Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/srp.pdf

[108] The Open/Closed Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/ocp.pdf

[109] The Liskov Sustitution Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/lsp.pdf

[110] The Interface Segregation Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/isp.pdf

[111] The Dependency Inversion Principle – Robert C. Martin
http://www.objectmentor.com/resources/articles/dip.pdf