Universidad Rey Juan Carlos

Escuela Técnica Superior en Ingeniería Informática

Diseño de un Bus de Eventos

Memoria del Trabajo Fin de Grado

en Ingeniería del Software

Autor: Serghei Sergheev

Tutor: María Teresa González de Lena Alonso

Junio 2022

Agradecimientos

En primer lugar, quiero agradecer a mi madre (Lilia Botnari) por sus conocimientos, ideas, consejos, y su constante apoyo durante todos estos años. He de comentar que sin ella dudo que llegase a ser quien soy hoy en día.

Quiero agradecer a todos los profesores de la universidad su dedicación, paciencia, comprensión y empeño por enseñar a los jóvenes de hoy, que serán quienes construyan un futuro mejor el día de mañana.

Finalmente, quiero agradecer a mi tutora Mayte su dedicación, consejos y tiempo. He de mencionar que ella es tanto una gran profesional en su campo, como una gran persona.

Resumen

Hoy en día, el desarrollo de software profesional es dependiente de librerías de terceros. Una dependencia de terceros es un código externo que abstrae una funcionalidad, y su uso en un proyecto, agiliza el proceso de desarrollo. Pero, por otro lado, no se tiene garantía alguna del mantenimiento ni perduración a lo largo del tiempo de este tipo de dependencias. El objetivo principal de este proyecto es desarrollar una librería que minimice las dependencias. En concreto, se propone la creación de un bus de eventos, basado en la arquitectura orientada a eventos, que promueva el desacoplamiento. La idea es llegar a crear una dependencia de terceros cuya adopción por parte de un equipo ajeno sea tán rápida que pueda llegar a convertirse en una dependencia de primeros con facilidad.

Índice general

Índice de figuras

Capítulo 1

Introducción

En el desarrollo de aplicaciones complejas es necesario emplear partes de código ya realizadas por otros. Esto se conoce como dependencias de terceros. En este primer capítulo se va a presentar la motivación de este trabajo, se comenzará introduciendo los conceptos básicos, para finalmente presentar el problema que existe con las dependencias de terceros.

1.1. Motivación

El diseño e implementación de software pueden parecer muy recientes, pero el primer programa fué creado y ejecutado en el año 1948 [1], es decir, la humanidad lleva casi un siglo desarrollando software.

En ciencia de la computación, disponemos de dos categorías esenciales que se complementan una a la otra, estas son, el hardware, la parte de microchips o circuitos, y el software, que gestiona o maneja dichos circuitos.

En cuanto al hardware, se sabe que este, se encuentra en constante evolución [2], podemos validarlo con el persistente crecimiento anual en la velocidad de los microprocesadores, por tanto, podemos afirmar que ”la Ley de Moore se resiste a perecer”[3].

En cambio, los principios de diseño de software permanecen intactos, continuamos partiendo de los mismos principios [4] y bloques de construcción básicos para diseñar los complejos sistemas de información de hoy en día.

La complejidad en el software y su desarrollo, se debe a diferentes motivos como: la complejidad del dominio del problema, las dificultades del proceso, la flexibilidad de las herramientas y el comportamiento impredecible [5].

Este proyecto, se va a enfocar en cómo se puede reducir esa creación de complejidad mediante la adopción de disciplinas o principios adecuados de diseño de software, centrándonos sobre todo, en promover la adopción de la arquitectura orientada a eventos.

1.1.1. Acoplamiento

Al hablar de creación de complejidad debido a la falta de uso de principios de diseño de software, una de las causas más comunes es no disponer de un balance de acoplamiento

entre módulos [6]. Un balance de acoplamiento, se refiere a la limitación de las interacciones entre módulos – que indica quién conoce a quién – y del flujo de dichas interacciones. Estableciendo una limitación, se consigue cierto grado de reutilización, y por tanto, una mejor mantenibilidad de los módulos, aunque es recomendable encontrar un balance [7], ya que un desacople total también es causa de creación de complejidad [8].

Es posible aplicar esta limitación tanto a módulos de alto nivel, con interacciones entre servicios, como a módulos de bajo nivel, con interacciones entre los diferentes objetos, es decir, a nivel de código.

Para comprender qué representa el acoplamiento, debemos saber cómo se genera.

1.1.2. Dependencias

El acoplamiento se genera cuando se crea una dependencia, por tanto, se deben gestionar las dependencias de manera efectiva para evitar la creación de acoplamiento.

Se puede definir una dependencia como una relación entre dos piezas o módulos independientes, en la cual, una no puede realizar su función o incluso existir sin la otra [9]. Normalmente cuando se menciona que un módulo A depende de otro B, significa que A no puede existir sin B, porque A referencia a B, y es habitual denominar a A cliente de B, y a B servicio de A. Nótese que B no depende de A.

Cada vez que un módulo referencie a otro, se está creando una dependencia, y por tanto, se crea acoplamiento. En caso de querer aplicar el término a nivel de código, el módulo se consideraría un objeto (aplicando el paradigma orientado a objetos [10]).

1.1.3. Cadena de responsabilidades

Una definición comprensible sobre qué es un patrón de diseño es: un patrón de diseño es una solución, es decir, un conjunto de normas aplicables a un problema en un contexto o situación específicos [11].

Desde el punto de vista de este proyecto, se definiría como: una coreografía concreta entre las dependencias que consigue un balance de acoplamiento, es decir, una manera en la que organizar las interacciones entre módulos que se ajusta a un contexto específico.

Nótese que el concepto de balance no es lo mismo que el de equilibrio, ya que el balance no restringe a la misma proporción de las características y depende del contexto. Por tanto, en ciertas situaciones una estructura con mayor proporción de rigidez logra un balance y en otras, una estructura más flexible es la mejor opción.

Una cadena de responsabilidad es un patrón de diseño, cuyo objetivo es evitar el acoplamiento entre la petición y el responsable de procesar dicha petición [12]. Para lograrlo, se provee la petición a cada uno de los manejadores. Cada manejador dispone de una lógica de procesamiento decidiéndo qué hacer al recibir la petición. Se puede definir un manejador como un elemento que puede consumir una petición concreta.

1.1.4. Arquitectura orientada a eventos

La arquitectura orientada a eventos es un mecanismo que nos provée de la habilidad para detectar eventos o sucesos, y nos permite reaccionar a ellos de manera inteligente [13].

Dos conceptos fundamentales que son la base de esta arquitectura son: los eventos, que transportan información de los sucesos que ocurren, y los manejadores, que son quienes consumen dichos eventos (la idea es similar al patrón de la cadena de responsabilidades).

Aparte de promover el bajo acoplamiento, este tipo de arquitectura, nos permite reaccionar de manera dinámica a sucesos que ocurren en un contexto determinado [14]. Por tanto, también nos proporciona la habilidad de sobreescribir las consecuencias de dichos eventos o sucesos.

En la mayoría de casos, habrá una alta dificultad a la hora de alterar, agregar, sobreescribir, o extender el comportamiento original, si el software no adopta una arquitectura orientada a eventos.

1.1.5. Librerías de terceros

La solución habitual para aplicaciones empresariales (o en general) es incorporar al proyecto una librería o dependencia de terceros que proporcione dicha funcionalidad de publicación y manejo de eventos.

Una dependencia de terceros [15] es una dependencia externa que no pertenece a la misma solución, es decir, no estará siempre accesible ya que no podemos asegurar su mantenimiento a lo largo del tiempo, porque este no depende de nosotros.

En caso de tomar la decisión de usar una librería de terceros, aparte de acoplar nuestro código a un código externo, es muy posible que dicha librería también tenga dependencias de otras librerías, es decir, ahora dependemos de manera transitiva de otras piezas de software y no de solo una.

1.2. Estado del arte

Para poder adoptar una arquitectura orientada a eventos a nivel de código, es necesario incorporar una dependencia al proyecto. Dicha dependencia puede ser, una dependencia de primeros o de terceros, aunque realmente, ambas soluciones tienen desventajas.

1.2.1. Dependencias de primeros

En caso de querer incorporar una dependencia de primeros [16], se debe disponer de un equipo profesional de desarrollo que diseñe, implemente y se asegure, tanto de la mantenibilidad, como de la transmisión del conocimiento de dicha librería.

Esta opción puede ser la más costosa, y si el equipo no dispone del suficiente conocimiento sobre el dominio, el diseño puede verse perjudicado, y por tanto, también la mantenibilidad, la facilidad de cambio y la transmisión del conocimiento.

1.2.2. Dependencias de terceros

En caso de optar por la opción de incorporar una librería de terceros al proyecto que la necesita, estaremos acoplando nuestro código (el código del proyecto) a una dependencia

de terceros, de la que no podemos asegurar su mantenimiento a lo largo del tiempo, aunque esta sea la opción más común.

Las implementaciones más reconocidas de la arquitectura orientada a eventos tienden a estar acopladas a un *framework* (conjunto de soluciones). Por ejemplo, la librería Spring Events [1], la cual necesita de otras dependencias de terceros – el *core* de Spring [2] – para poder funcionar.

En caso de querer usar el bus de eventos incorporado en la librería Google Guava [3], se puede observar que el bus revela detalles que debería abstraer del cliente (*leaky abstractions* [17]), como por ejemplo, la habilidad de almacenar manejadores, que se permite directamente a través de la interfaz del bus.

Elegir una librería pública de código abierto que no sea muy conocida, tampoco es la mejor opción, ya que pueden surgir problemas como: el funcionamiento no es el esperado, hay una escasez de pruebas automatizadas o una falta de documentación.

A parte, el enfoque, diseño y pruebas no están centradas en la parte de eventos de dicha librería de terceros, sino en el *framework* en su conjunto. Normalmente, no se pone el empeño en esta parte concreta de emisión y manejo de eventos.

1.2.3. Lo ideal e inexistente

Un indicativo de haber construido una arquitectura de calidad – en este caso, la arquitectura de nuestra librería de eventos – es que ésta no depende de *frameworks*, herramientas o entornos [18]. Por ello, la idea es disponer de una librería de terceros de alta calidad con un buen diseño, pruebas y documentación, y que la adopción por parte de un equipo ajeno sea tan rápida que dicha librería pueda convertirse en una dependencia de primeros. En caso de ser así, esta nueva dependencia podría ser mantenida, persisitiendo en el tiempo a pesar de actualizaciones del lenguaje o cambios en el entorno, y modificada según las necesidades de negocio en cualquier momento.

1.3. Objetivos

El objetivo principal es desarrollar una librería que permita la minimización de las dependencias. En este caso, la creación de una librería de eventos, con un código fuente fácilmente mantenible y adoptable por el equipo de desarrollo.

Debido a los motivos ya mencionados, se propone la creación de una librería de eventos que deberá cumplir con los siguientes propósitos:

- Debe ser una dependencia mínima, la cuál, podemos definir como una librería que debe estar programada en un lenguaje en modo *vanilla* (es decir, en lenguaje puro) y sin dependencias de terceros, a excepción de las dependencias para pruebas unitarias y cobertura de código.

[1] Spring Events - librería de integración del manejo de eventos con el core de Spring
[2] Spring Framework - conjunto de librerías para el desarrollo de aplicaciones e inversión de control
[3] Google Guava - librería de utilidades comunes de código abierto para Java

- Se dispondrá de una documentación de alta calidad, la cual permita a un equipo ajeno adoptar la dependencia y asimilar su código fuente con facilidad para poder adaptarla a sus necesidades cuando sea necesario [19].
- Es deseable disponer de un alto porcentaje de cobertura de código, para ello, se realizarán pruebas que verifiquen el comportamiento esperado. También se deberá disponer de un proceso determinista que permita verificar dicha cobertura de código.

1.4. Estructura de la memoria

El resto del documento se divide en cuatro capítulos. En el siguiente se describe la metodología de desarrollo. El capítulo tres trata sobre el diseño, desarrollo y despliegue de la solución. En el cuatro se describen los diferentes experimentos, pruebas y métricas. Finalmente, en el quinto se presentan las conclusiones de este documento.

Capítulo 2

Metodología

En este capítulo se van a presentar diversas metodologías de software existentes y cómo se han adaptado para el desarrollo de este proyecto.

2.1. Metodologías de desarrollo

Antes de la realización de cualquier proyecto software, es recomendable tener establecida una metodología de desarrollo, es decir, tener estandarizado el cómo se va a estructurar, planificar y revisar el progreso a lo largo del tiempo. En este caso, no disponemos del conocimiento sobre el dominio que modelamos, es decir, no conocemos cuáles son los elementos que intevienen en el proceso de transmisión y recepción de eventos para poder modelar con certeza el dominio a nivel de código. Por ello, se analizarán diferentes metodologías para ver cual se ajusta mejor a este tipo de proyecto.

2.2. Metodología en cascada

La metodología en cascada es originaria de las industrias de manufactura y construcción. El proceso se basa en dividir las actividades de un proyecto en fases secuenciales, donde cada fase depende de la anterior. La primera definición formal de esta metodología se publica en un artículo en 1970 por Winston W. Royce [1] [20]. Esta metodología afirma que el cambio es costoso, indeseable y evitable [21]. Siendo dichas afirmaciones decisivas para no adoptar esta metodología para el proyecto actual, el cual, se basa en el cambio contínuo y la investigación del desconocido dominio a modelar.

2.3. Metodologías agiles

Las metodologías ágiles apoyan la idea de que los requisitos sean cambiantes a medida que se progresa en un proyecto [22]. Este tipo de metodologías surgen debido al fracaso las prácticas tradiccionales, caracterizadas por sus rígidas y estructuradas fases.

[1] Winston W. Royce - Computólogo estadounidense y pionero en el campo de la ingeniería de software

El manifiesto ágil [23] fue redactado por diecisiete de los ingenieros de software más influyentes de la época, que fueron convocados por Kent Beck [2]. Esta declaración ejemplifica el contraste entre las metodologías ágiles y las tradicionales.

Para el proyecto actual, de las metodologías ágiles se adoptará su típico ciclo de vida, caracterizado por ser: iterativo, poco a poco ir construyendo el todo, e incremental, cada entrega siendo un incremento respecto a la anterior, cada fase (requisitos, ánalisis, diseño, etc.) se realiza varias veces. Finalmente, la clave es, que cada iteración termina con toda seguridad en un incremento [24].

2.4. Framework Scrum

El término Scrum (que se traduce como melé) fue acuñado y definido por Ikujiro Nonaka [3] e Hirotaka Tekeuchi [4] en los años 80, época en la que las grandes empresas tecnológicas comenzaban a dominar el mercado y definir sus propias metodologías de trabajo. Ambos publicaron en 1986 en la Harvard Business Review [5] un artículo [25] dando inicio a esta nueva metodología ágil. En la actualidad, Scrum consta de tres elementos clave: los roles, los artefactos y los eventos [26].

- El primer elemento de esta metodología son los roles:
  - El *Development Team*, consta del personal técnico (autónomo y especializado) que desarrolla un incremento potencialmente entregable del producto al final cada *Sprint* (o iteración).
  - El *Product Owner*, es el responsable de maximizar el valor del trabajo realizado por el equipo de desarrollo. Su labor se basa en la gestión del producto (a nivel de negocio).
  - El *Scrum Master* se encarga de facilitar y asegurarse de la correcta aplicación del proceso de Scrum. Debe ser fiel a las prácticas y valores de esta métodología.
- El segundo elemento son los artefactos, estos representan la documentación, los diagramas, las métricas, y otros elementos que aportan valor al proyecto. Los artefactos específicos a esta metodología son:
  - El *Product Backlog* es una lista dinámica (o cambiante) de tareas priorizadas que indican los cambios (adiciones, modificaciones o eliminaciones) necesarios a realizar en el producto.
  - El *Sprint Backlog* se compone de una selección de tareas del *Product Backlog* subdivididas en tareas más técnicas que indican en lo que se va trabajar durante el *Sprint* (o iteración) actual.
  - Un incremento representa la compleción de una serie de tareas del *Sprint* que permiten estar un paso más cerca de alcanzar el producto deseado (o *Product Goal*).

[2] Kent Beck - Ingeniero de software estadounidense
[3] Ikujiro Nonaka - Profesor emérito de la universidad Hitotsubashi de Tokyo
[4] Hirotaka Takeuchi - Profesor en la unidad de estrategia de Harvard Business School
[5] Harvard Business Review - Revista de gestión, de periodicidad bimensual, publicada desde 1922

- El último de los elementos clave de Scrum son los eventos:
  - Un *Sprint* es un evento de longitud fija cuyo objetivo es crear un incremento del producto mediante la inspección y adaptación de los artefactos. Cada *Sprint* comienza inmediatamente tras la finalización del *Sprint* anterior.
  - El *Sprint Planning* se considera la actividad más relevante de una iteración (o *Sprint*). Se basa en dividir historias de usuario en tareas y el resultado esperado de esta reunión es una estimación [27].
  - La *Daily Scrum* es una reunión diaria (típicamente de 15 minutos) que se basa en la transparencia y la inspección del trabajo realizado para tener una oportunidad de adaptación para el día siguiente.
  - La *Sprint Review* se trata de una revisión del *Sprint*, el objetivo es inspeccionar el resultado del *Sprint* ya finalizado y determinar posibles futuras adaptaciones.
  - La *Sprint Retrospective* es un evento cuyo propósito es planificar formas de aumentar la calidad y la eficacia del proceso actual.

Los diferentes elementos usados y adaptados de esta metodología ágil para el desarrollo del proyecto actual son las siguientes:

- En cuanto a los roles, en este proyecto al tomar una sola persona todos los roles, carece de sentido, tanto división en roles, como la separación de responsabilidades.
- Respecto a los artefactos, por simplicidad se ha fusionado el *Product Backlog* y *Sprint Backlog* en un único *Backlog*. Este *Backlog* estará priorizado por la ordenación topológica [6] de las tareas asociadas a los conceptos emergentes (no se pueden completar totalmente tareas que impliquen un alto nivel de abstracción hasta que se hayan finalizado las de bajo nivel, es decir, se usa un enfoque de diseño *bottom-up*). También se incluye la noción de incremento. Finalmente, cabe comentar que se dará mayor prioridad a disponer de un producto funcional que a generar documentación.
- En relación con los eventos, se omitirá la *Daily Scrum*. Como este proyecto se basa en la investigación tampoco tiene sentido realizar un *Sprint Planning*, ya que el cambio es demasiado frecuente, por tanto, no se podrá obtener una estimación precisa (sino estaremos creando desperdicio). Se fusionará la *Sprint Review* con la *Sprint Retrospective*, en una sola reunión de revisión al final de cada iteración para reflexionar sobre el progreso y planificar cuales serán los próximos avances. Aunque en un comienzo, cuando el diseño no tiene una base, esto puede llegar a ocurrir diariamente por las mañanas a primera hora.

2.5. Framework Kanban

Kanban es una metodología ágil sencilla y con reglas mínimas. Fue creada por la compañía japonesa de fabricación de automóviles Toyota [7] en los años 50. La investigación centrada en la gestión de cadenas de suministro resultó en la creación de una metodología que

[6] Ordenamiento Topológico - Algoritmo para identificar una precedencia al ejecutar una serie de tareas
[7] Toyota - Compañía japonesa de fabricación de automóviles

dispone de un bucle de retroalimentación, mediante el cual, la demanda se puede utilizar para limitar la oferta en cada fase de producción [28]. Hoy en día, esta metodología consta de tres elementos clave: la definición y visualización del flujo de trabajo, la gestión de manera activa de las tareas y la continua mejora del flujo de trabajo [29]. Una de las ideas clave de Kanban es que se basa en la visualización fase/proceso.

- Definición y visualización del flujo de trabajo: las tareas se definen como unidades individuales de valor, se mantiene un *tracking* del estado de cada tarea, y se establece una limitación del trabajo máximo en progreso.

- Gestión de manera activa de las tareas: se aplica la limitación de la cantidad de tareas en progreso, se intenta evitar el apilamiento de las tareas, y el foco se pone en desbloquear las tareas bloqueadas.

- Continua mejora del flujo de trabajo: es responsabilidad de los miembros del equipo comentar su punto de vista, proponer e implementar mejoras que beneficien al equipo de manera continua.

Las diferentes elementos y principios de esta metodología aplicados al proyecto actual son:

- En cuanto a la definición y visualización del flujo de trabajo, se adoptará un tablero de Kanban, que irradiará información sobre el progreso. Dicho tablero se dividirá en tres columnas: tareas pendientes, tareas en progreso y tareas terminadas.

- Respecto a la gestión de manera efectiva de las tareas, se establecerá un límite máximo de tres tareas simultáneas en progreso, ya que múltiples cambios de contexto pueden llegar a perjudicar el avance.

- En relación con la continua mejora del flujo de trabajo, se irán proponiendo mejoras a lo largo del proceso, ya que no se quiere introducir mayor complejidad desde un principio.

2.6. Framework Cynefin

Cynefin es un marco de trabajo para ayudar en el proceso de toma de decisiones [30]. Fué creado por Dave Snowden [8] en 1999 cuando trabajaba para IBM Global Services [9]. Este modelo conceptual se basa en una matriz de dos dimensiones que ayuda a poner un problema en contexto y contemplar las diferentes decisiones que se pueden tomar en relación a su causa y efecto (ver Figura 1).

[8] Dave Snowden - Consultor de gestión e investigador
[9] IBM Global Services (IBM Consulting) - Reconocida empresa multinacional de consultoría y servicios

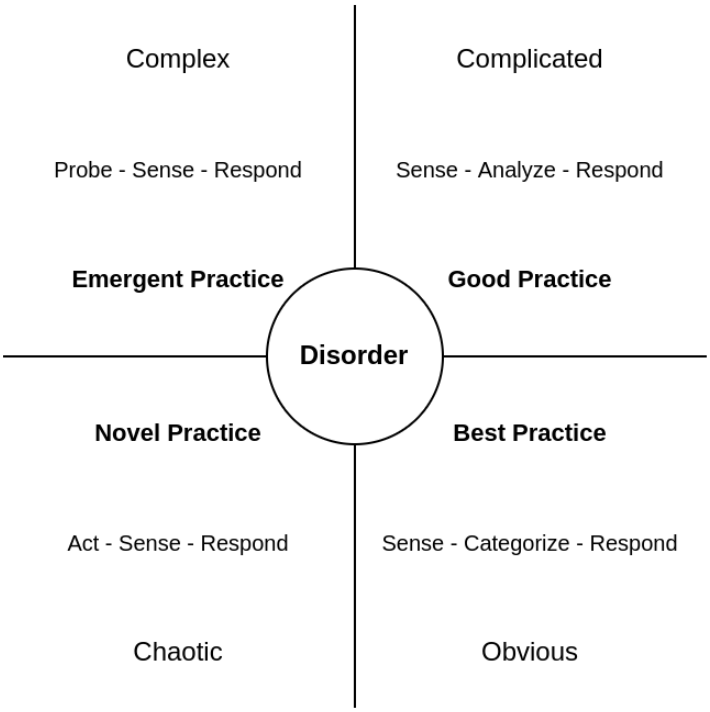


Figura 1: Framework de decisión de Cynefin

Hay que considerar que la información obtenida a lo largo del proceso precede al *framework*, por tanto, este gráfico solo sirve de apoyo para ponernos en contexto. Antes de tomar una decisión definitiva sobre cómo actuar, sería adecuado complementar la decisión de este modelo conceptual con un análisis heurístico.

Se parte desde el centro del modelo (en el cuadrante del desorden) y a medida que se progresa en la resolución del problema vamos moviendo el círculo por los diferentes cuadrantes.

El significado de cada uno de los cuadrantes del modelo es el siguiente [31]:

- Cuadrante obvio: se asemeja a la gestión tradicional, se establecen los diferentes conceptos (*sense*), se categorizan y finalmente se responde mediante la aplicación de la regla o buena práctica correspondiente.

- Cuadrante complicado: indica que no se dispone de buenas prácticas ya definidas que se ajusten al problema actual, solo de un conjunto de posibles buenas prácticas que no encajan del todo. Primero sentimos y analizamos, antes de responder. Aquí es donde suele encajar el trabajo de los expertos de dominio.

- Cuadrante complejo: se dispone un conjunto de problemas complejos y desordenados. Aquí ni siquiera se dispone de posibles buenas prácticas, solo se dispone de prácticas emergentes. No hay causalidad, por tanto, se procede a probar y sentir antes de responder.

- Cuadrante caótico: es visitado cuando el problema genera caos. Aquí la experimentación con nuevas prácticas es la principal herramienta. Primero se actúa (se experimenta), luego se siente (o percibe) y finalmente se responde. Raramente debemos entrar en este cuadrante, y si se hace, es normalmente para intentar innovar.

Este *framework* se utiliza a lo largo del desarrollo de este proyecto. El flujo habitual es movernos por los diferentes cuadrantes. A medida que nuestro conocimiento sobre el problema se incrementa, hay un paso del cuadrante del caos al complicado y complejo, y finalmente llegaremos al cuadrante claro (u obvio). La clave es, si pasamos por el cuadrante del caos, que sea el menor tiempo posible, y en caso de llegar a ocurrir (al ser este proyecto de investigación ocurre bastante), que se sepan las razones del paso por dicho cuadrante. Para así, generar conocimiento mediante la estandarización del proceso de resolución y de ese modo no repetir los mismos errores (es decir, definir la buena práctica y estandarizarla).

Capítulo 3

Diseño, desarrollo y despliegue

En este capítulo se definirá el diseño, desarrollo y despliegue del proyecto en detalle.

3.1. Tecnología usada

A la hora de modelar un problema, es necesario conocer el lenguaje de programación en el que se va a intentar resolver el problema propuesto, para así conocer las limitaciones que esto conlleva a nivel de código. En caso contrario, se puede llegar al *architectural mismatch* [32], es decir, lo diseñado no se puede representar (no encaja) a nivel de código.

Las tecnologías seleccionadas para modelar la solución al problema propuesto a nivel de código son:

- Un lenguaje de programación, es un lenguaje formal, es decir un lenguaje con unas reglas gramaticales bien definidas que usamos para especificar una serie de instrucciones que forman un programa. Para este proyecto, la elección fue usar Java, lenguaje típico de aplicaciones empresariales, concretamente la versión Amazon Corretto 8 (JDK) [1].
- Un gestor de dependencias, es una pieza de software que nos permite gestionar las dependencias de terceros (código externo del que depende nuestro proyecto), es decir, facilita la manera de incorporar y mantener *tracking* del versionado de las dependencias de un proyecto. Para este caso, por la madurez de la herramienta (han pasado 20 años desde su lanzamiento inicial), se eligió Apache Maven [2].
- Una prueba automatizada, se basa en redactar un código que prueba el comportamiento o funcionalidad de otro código. Para este proyecto se usarán diferentes utilidades (dependencias de terceros) para facilitar la gestión de las pruebas automatizadas como: JUnit [3] (para facilitar la creación de pruebas unitarias), Mockito [4] (para fabricar *mocks* [33], es decir, crear objetos falsos), JaCoCo [5] (para generar reportes de cobertura de código).

[1] Amazon Corretto - distribución del kit de desarrollo de Java lista para producción
[2] Apache Maven - herramienta de comprensión y gestión de proyectos de software
[3] JUnit - librería para facilitar la creación de pruebas unitarias en Java
[4] Mockito - framework permite la creación de objetos dobles de prueba (objetos simulados) en Java
[5] JaCoCo - librería de pruebas diseñada para la integración y cobertura de código en Java

3.2. Prácticas técnicas

Aparte de las metodologías a nivel de proceso, también es importante definir prácticas técnicas que complementen a dichos métodos organizativos. Por ello, se va a remarcar el uso de la disciplina de las pruebas automatizadas para verificar la funcionalidad del código. Tras la ejecución de cada prueba, se reporta el resultado de dicha prueba, que habitualmente se representa con un color verde, en caso de éxito, o rojo, en caso de fracaso.

A lo largo del desarrollo de este proyecto, hay que destacar el uso de Test Driven Development (TDD), el desarrollo guiado por pruebas automatizadas. En su mayoría centrado en el enfoque purista, es decir, se usa como herramienta de diseño, ayudándonos así a descubrir el diseño o la arquitectura a medida que se progresa. Se empleará TDD, pero no la definición establecida por las prácticas y convenciones modernas. Debido a que el significado de este término ha ido variando a lo largo del tiempo y su significado se ha ido distorsionando hacia Test-First Development (TFD), que indica que las pruebas se deben realizar antes que el código [34].

3.3. Diseño

La investigación comenzó con una búsqueda conceptual, ya que partimos del desconocimiento del dominio que implica a la arquitectura orientada a eventos. Las primeras cuestiones que emergieron fueron: ¿qué es un evento? y ¿qué es un escuchador?

Tras comprender que los eventos son plantillas de datos que transportan información de un suceso, y los escuchadores son quienes reciben dicha información y la procesan como proceda, la siguiente pregunta que surge es: ¿mediante qué mecanismo llega un evento a un escuchador?

La primera idea que surge es disponer de un elemento intermedio, al que entregamos la información y éste realize un proceso de enrutamiento, entregando dichos datos al escuchador correspondiente (ver Figura 2).

Figura 2: Diagrama adhoc del flujo inicial

De inmediato, tras asimilar las ideas conceptuales, se intentó plasmar dicho conocimiento adquirido sobre el dominio a nivel de código, para ver de qué manera pueden encajan los conceptos con el lenguaje en sí.

Los conceptos de dominio encajaban con el código, lo que no encajaba era que la responsabilidad de contener todos los escuchadores fuese del transmisor, ya que el transmisor – como indica el término – debe transmitir o enrutar, pero no almacenar.

Por ello, emerge un nuevo concepto que nos permite abstraer dicha responsabilidad, un registro de escuchadores, cuya tarea es almacenar los escuchadores de cierta manera y devolverlos en caso de ser requeridos (ver Figura 3).

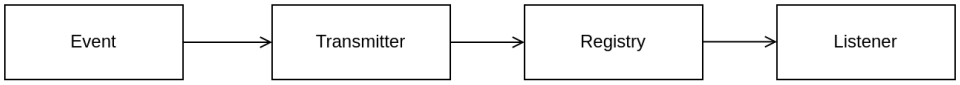


Figura 3: Diagrama adhoc del flujo con registro

Al intentar modelar este nuevo concepto, el registro de escuchadores a nivel de código, comienzan a surgir diversas dudas subyacentes sobre la manera en la que el transmisor va a proveer al escuchador de un evento.

Las dudas subyacentes fueron las siguientes: ¿cada escuchador tendrá un solo método?, ¿en caso de ser así, será mediante la implementación de una interfaz por parte del escuchador, y por tanto, la definición de un contrato?, ¿es realmente necesario restringir a un solo método por escuchador?

Nos podemos percatar de que las preguntas que estamos formulando implican un nuevo concepto, el concepto de método de una clase – en este caso de la clase escuchadora –, dicho método actuará de manejador, por tanto, en este contexto podemos denominarlo método manejador.

En caso de optar por esta primera aproximación, se definiría una interfaz funcional, es decir, una interfaz con un solo método que sería el método manejador. Dicho método, dispondría de un único parámetro que sería el evento a consumir. Cada clase que implementase dicha interfaz, se consideraría un escuchador. Pero ahora estamos limitados a un método manejador por cada clase escuchadora, aparte de tener que realizar un *casting* para convertir el evento genérico recibido como parámetro en el evento específico esperado.

Finalmente, se decidió no usar una interfaz para definir un método manejador por contrato, evitándo así la limitación de un solo método por escuchador que esto suponía. En vez de eso, cada método de la clase escuchadora que se quiera considerar método manejador, se identificará con una anotación.

Se definirá que un escuchador consta de uno o varios métodos manejadores, y que cada método manejador dispondrá de un único parámetro, que será la instancia del evento a consumir por éste. Ahora, en vez de almacenar escuchadores en el registro, en su lugar se deberá realizar con métodos manejadores, ya que estos son los que realmente van a consumir el evento (ver Figura 4).

Figura 4: Diagrama adhoc del flujo con métodos manejadores

Partimos de que el cliente (el usuario de esta librería) provee un escuchador al registro, pero para almacenar los métodos manejadores presentes en la clase de dicho escuchador, primero hay que identificar todos los métodos manejadores. Para ello, hay que filtrar y recolectar los métodos de la clase que estén anotados con una anotación de manejador personalizada, siéndo estos los métodos manejadores a registrar.

Todo método de una clase escuchadora que sea un método manejador, deberá anotarse con dicha anotación personalizada, esta anotación servirá para identificar que un método de un escuchador es un método manejador.

Cada método manejador también necesita disponer de metadatos asociados a este, por tanto, aplicando el principio de composición de programación orientada a objetos, decoraremos el método manejador añadiéndole un identificador, orden o prioridad y el tipo de evento que puede consumir.

Tras un análisis, se decidió que en vez de proveer al registro los métodos manejadores de forma directa, hacemos que el cliente provéa el escuchador en cuestión, evitando así *leaky abstractions* [17], ya que no es necesario que el cliente (el usuario de la libreria) conozca cómo se realizan la extración y almacenamiento internos.

El flujo sería que el cliente provee un escuchador, y a partir de éste, debemos extraer y almacenar todos los métodos manejadores, y también estar preparados para recuperar en cualquier momento los métodos manejadores que sean capaces de consumir un tipo de evento específico.

Por tanto, para poder realizar dicha recuperación, es necesario que el registro contenga la asociación de cada tipo de evento (la clase concreta) a una colección que contenga todas las instancias de métodos escuchadores que puedan consumir dicho tipo de evento.

La cuestión que surge ahora es: ¿qué tipo de colección debemos usar para almacenar los métodos? Para decidirlo debemos entender el flujo de ejecución diseñado hasta ahora (ver Figura 5). En primer lugar, se publica un evento, dicho evento se transmite por el transmisor, luego el registro recupera la colección de manejadores que puedan consumir dicho tipo de evento, y finalmente, los manejadores reciben el evento en sí. Pero, a la hora de proveer dicho evento a cada uno de los manejadores obtenidos, la pregunta que surge aquí es ¿en que orden?, esa es la clave para decidir el tipo de estructura de datos deseado para la colección.

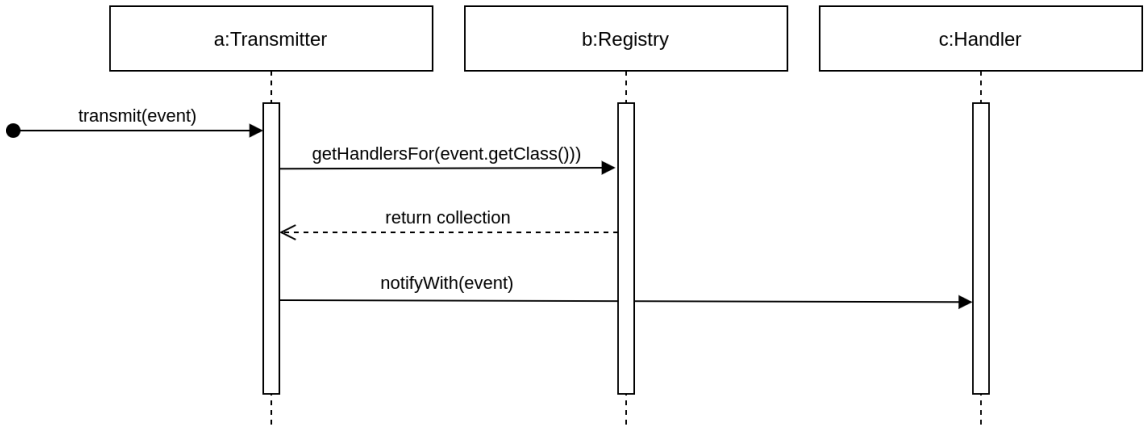


Figura 5: Diagrama de secuencia inicial de la transmisión de un evento

La colección será una cola de prioridad, debido a que cada manejador tiene un orden o prioridad de ejecución asociados en el que se debe consumir el evento transmitido.

En Java, la estructura nativa de la cola de prioridad (`Heap`) no mantiene el orden de iteración, por tanto, no se puede usar esta estructura ya existente. En caso de querer respetar el orden, se deberían ir sacando los elementos uno a uno, teniendo así que mutar la cola original del registro, pero no se quiere mutar, ya que se necesita disponer de los manejadores y su orden originales cada vez que se transmita un evento.

Un opción sería, que cada vez que se transmita un evento, se clone la cola de métodos manejadores asociada a dicho tipo de evento, e ir sacando los elementos de la cola copiada, uno a uno, de esa manera no se pierde la cola original ni su orden, es decir, no se pierde la información original. El problema surge cuando a medida que los métodos manejadores asociados a un tipo de evento específico aumentan en cantidad, también aumenta el *overhead* de copiar dicha cola e ir sacando los métodos manejadores de uno en uno.

Por ello, se propone la implementación de una cola de prioridad doblemente enlazada que permitirá mantener el orden de iteración intacto. No debe ser motivo de preocupación la complejidad de añadir un nuevo método manejador o eliminar uno existente en tiempo de ejecución, ya que dichas operaciones se realizarán en momentos puntuales, lo ideal es poblar el registro con métodos manejadores en tiempo de inicio.

Si se observa el esquema de flujo actual (ver Figura 6), se puede apreciar que el transmisor y el registro están demasiado acoplados, es decir, el alto y el bajo nivel están demasiado ligados, también a su vez surge la necesidad de proveer al cliente de una interfaz más sencilla – sin interacción con operaciones de bajo nivel –.

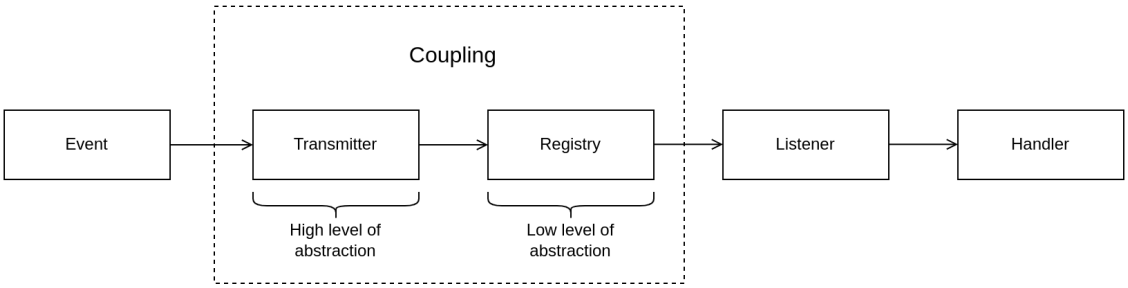


Figura 6: Diagrama adhoc del flujo enfatizado en el acoplamiento

Por tanto, se propone la creación de un gestor de eventos, un elemento intermedio que abstraiga las responsabilidades de extracción y persistencia de métodos manejadores del cliente (quien usa la librería), también a su vez se renombrará el transmisor, a bus de eventos, para enfatizar así un alto nivel de abstracción (ver Figura 7).

El término de registro, ahora a nivel de código se convierte en un conjunto de clases con sus respectivas estructuras de datos de bajo nivel, las cuales coordina el gestor, para así intentar reducir la complejidad de diferentes operaciones mediante la redundancia.

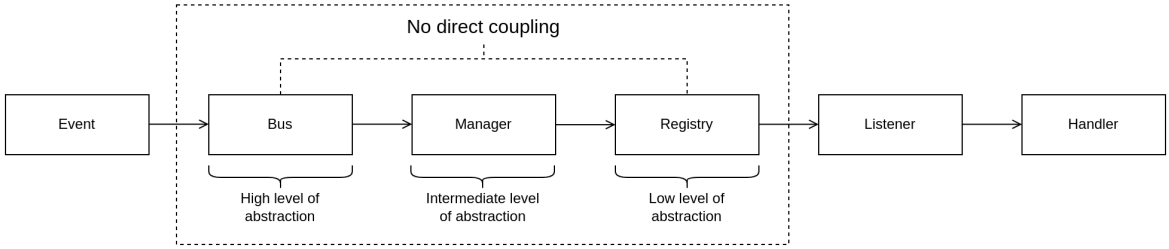


Figura 7: Diagrama adhoc del flujo final

La necesidad de la redundancia emergió debido a que nuestro objetivo era crear una interfaz flexible para el gestor, mediante la cual, el cliente (quien usa la librería) pudiese manipular el funcionamiento como desee, sin reducir el rendimiento original de manera notable.

Para poder realizar las acciones definidas en la interfaz sin reducir el rendimiento, surgió la necesidad de la redundancia para almacenar: un conjunto de escuchadores, la relación del tipo de escuchador a su instancia, la asociación del tipo de evento a su cola de métodos manejadores notificables, y finalmente la relación de un escuchador a sus métodos manejadores notificables (ver Figura 8).

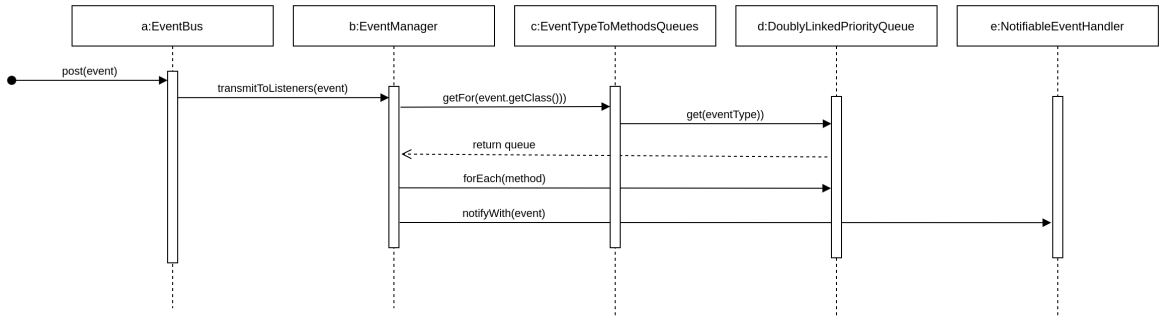


Figura 8: Diagrama de secuencia final de la transmisión de un evento

Se han añadido dos implementaciones diferentes del gestor de eventos, una implementación simple, que se considera que debe poblarse en tiempo de inicio, y otra concurrente, que permite cambios en el registro en tiempo de ejecución. Se permite al cliente instanciar de manera explícita el tipo de gestor de eventos que considere necesario e indicarlo al *builder* del bus.

Finalmente, se crea un elemento *builder* que proveerá al cliente (quien usa la librería) de una interfaz fluida [6], que permitirá evitar confusiones a la hora de construir un objeto complejo, en este caso del bus de eventos, facilitando establecer la configuración o preferencias del bus que se está construyendo. La arquitectura de la librería resultante, se puede apreciar en el diagrama de clases final (ver Figura 9).

[6] Interfaz Fluida - construcción del paradigma orientado a objetos que facilita transmitir contexto

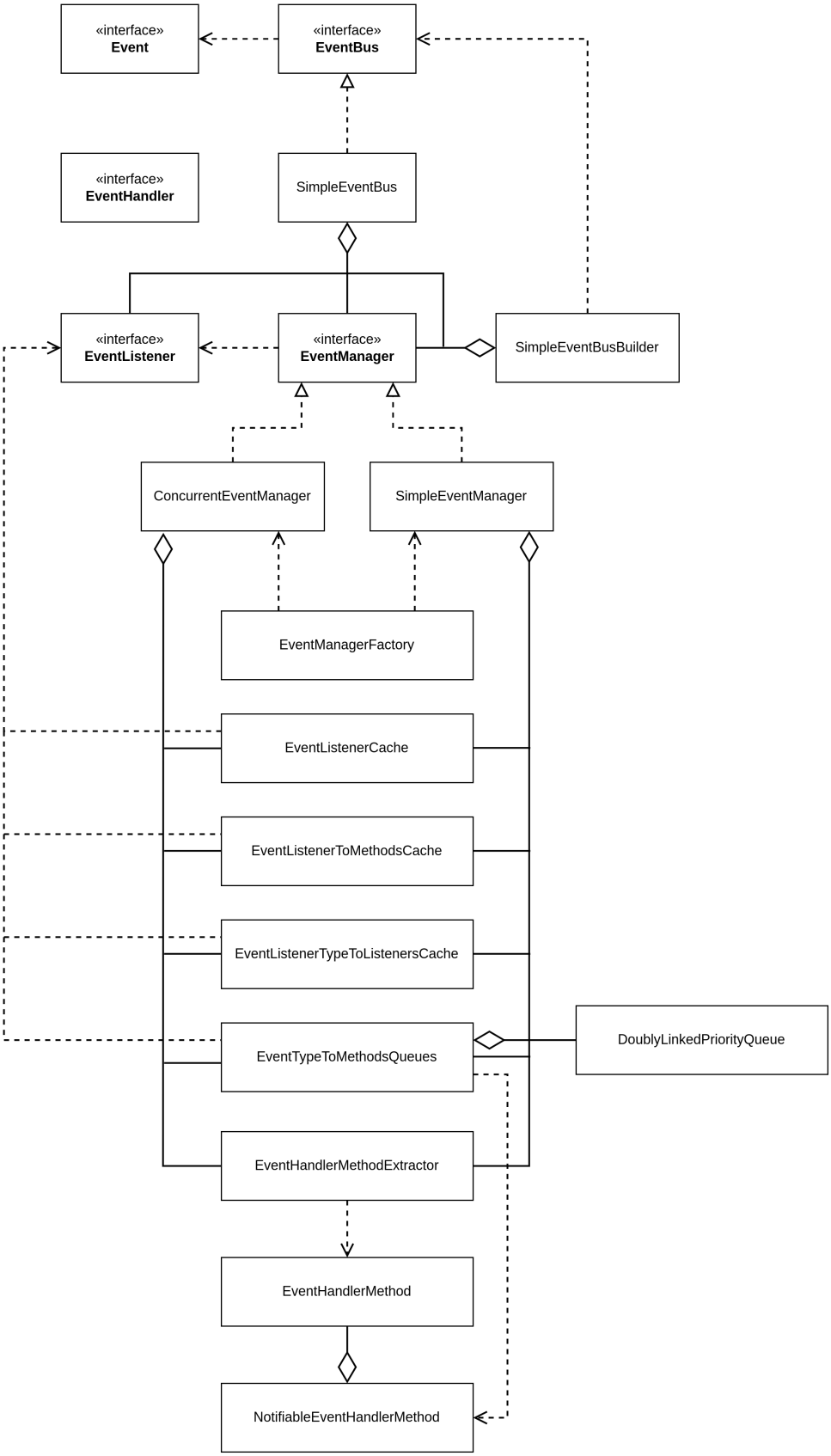


Figura 9: Diagrama de clases final

3.4. Desarrollo

Debido la naturaleza de este proyecto, basado en la investigación y refactorización constantes, tanto el modelo conceptual, cómo el código de la librería, han pasado por una serie de evoluciones a lo largo del tiempo.

El desarrollo en general ha tenido una duración aproximada de seis meses, en los que se han realizado 35 *commits* en total (ver Figura 10), con lo que se puede estimar que se han realizado más de 30 evoluciones durante el proceso. El resto del tiempo se ha dedicado a la documentación de las decisiones.

```
* 96d066a - (HEAD -> master, origin/master, origin/HEAD) Performed multiple refactorings (general javadocs) (4 months ago) <Serghei Sergheev>
* d805f82 - Modified «sonar.properties» to keep track of JaCoCo coverage (4 months ago) <Serghei Sergheev>
* 47246b3 - Added «sonar.properties» for static code analysis (4 months ago) <Serghei Sergheev>
* 0ce618b - Performed minor refactorings (names for syntactic sugar) (4 months ago) <Serghei Sergheev>
* 831a1e2 - Reformated parent and child «pom.xml» files (4 months ago) <Serghei Sergheev>
* 69fba67 - Performed minor refactorings (methods javadocs) (4 months ago) <Serghei Sergheev>
* 850f0c0 - Updated «README.md» to add reference to wiki explanations (4 months ago) <Serghei Sergheev>
* 3da58e0 - Updated «README.md» to improve explanation (4 months ago) <Serghei Sergheev>
* e83f674 - Performed minor refactorings (classes javadocs) (4 months ago) <Serghei Sergheev>
* 4d49de5 - Updated project modules names (4 months ago) <Serghei Sergheev>
* 0a34043 - Added «LICENSE» for copyright protection (5 months ago) <Serghei Sergheev>
* b728a5c - Added «README.md» to increase documentation (5 months ago) <Serghei Sergheev>
* 13f87a8 - Performed minor refactorings (names for syntactic sugar) (5 months ago) <Serghei Sergheev>
* 81a8dae - Added «.gitignore» to exclude unnecessary files (5 months ago) <Serghei Sergheev>
* e20683a - Fixed packaging to bundle spec with impl in a single artifact (5 months ago) <Serghei Sergheev>
* 08bdd3b - Adapted code and comments using new manual lint style (5 months ago) <Serghei Sergheev>
* 4ed375c - Performed multiple refactorings (focus on quality) (6 months ago) <Serghei Sergheev>
* 3613027 - Fixed «DoublyLinkedPriorityQueue» implementation bugs (6 months ago) <Serghei Sergheev>
* 2bc5dee - Performed multiple refactorings (auxiliary structures) (6 months ago) <Serghei Sergheev>
* bc37e72 - Focused on code coverage incrementing tests amount (6 months ago) <Serghei Sergheev>
* b20654a - Finished «DoublyLinkedPriorityQueue» implementation (7 months ago) <Serghei Sergheev>
* 6bcff8f - Performed multiple refactorings (auxiliary structures) (7 months ago) <Serghei Sergheev>
* 848f492 - Added draft auxiliary structures for handlers storage (7 months ago) <Serghei Sergheev>
* 55457ae - Added «DoublyLinkedPriorityQueue» implementation draft (7 months ago) <Serghei Sergheev>
* bb86e3a - Performed multiple refactorings (handlers storage) (8 months ago) <Serghei Sergheev>
* d91abfe - Added pending unit tests & Performed refactorings (8 months ago) <Serghei Sergheev>
* b760d11 - Performed multiple refactorings (handlers storage) (9 months ago) <Serghei Sergheev>
* 6ee56e5 - Split project into 2 modules (spec and impl) (9 months ago) <Serghei Sergheev>
* 43f8dec - Performed multiple refactorings (handlers storage) (9 months ago) <Serghei Sergheev>
* b68c483 - Added and improved javadocs for several classes (10 months ago) <Serghei Sergheev>
* b714378 - Excluded compiled class files from source (10 months ago) <Serghei Sergheev>
* 0bd5276 - Documented «EventHandlerMethodExtractor» with javadocs (10 months ago) <Serghei Sergheev>
* 9fed31a - Documented «EventHandlerMethod» with javadocs (10 months ago) <Serghei Sergheev>
* b860a17 - Designed specification and advanced in implementation (10 months ago) <Serghei Sergheev>
* a7c7cb4 - Initial commit (10 months ago) <Serghei Sergheev>
```

Figura 10: Histórico de commits

A continuación, se detallarán las decisiones tomadas y los cambios más relevantes que se han efectuado durante las iteraciones realizadas a lo largo de cada mes:

- En el primer mes, se estableció que la reunión de revisión (mezcla de *Sprint Review* y *Sprint Retrospective*) se realizaría todos los días por la mañana. Durante este mes se diseñaron las interfaces que definen la especificación de la librería y se terminó parte de la implementación. Finalmente se procedió a analizar las posibles maneras de almacenar los métodos manejadores (ver Figura 11).
    - En la primera iteración, se creó el tablero de Kanban. Se fué construyendo el *Backlog* de tareas a la par que se investigaba sobre la arquitectura orientada a eventos para ganar conocimiento sobre el dominio.
    - En la segunda iteración, ya se tiene conocimiento básico de los conceptos que implican a la arquitectura orientada a eventos. Por tanto, se comenzó a modelar dichos conceptos (evento y manejador) a nivel de código. Al mismo tiempo, se investigaba más a fondo sobre este tipo de arquitectura y se analizaban proyectos similares ya existentes (Spring Events y Google Guava).

- En la tercera iteración, se estableció el *Product Goal* (la meta global del proyecto para todas las iteraciones) que se basó en desarrollar un prototipo funcional que persiga la calidad de software. Se siguió con el continuo proceso de refactorización del código, a la vez que se iban descubriendo más conceptos de dominio (transmisor y registro).
- En la cuarta iteración, tras varias refactorizaciones, se terminó la especificación (las definición de las diferentes interfaces) y se comenzó con su correspondiente implementación. Debido a que se fue siguiendo Test Driven Development (TDD) a lo largo del proceso, se descubrió que los eventos no se estaban almacenando de manera correcta, ni respetaban su prioridad de ejecución.

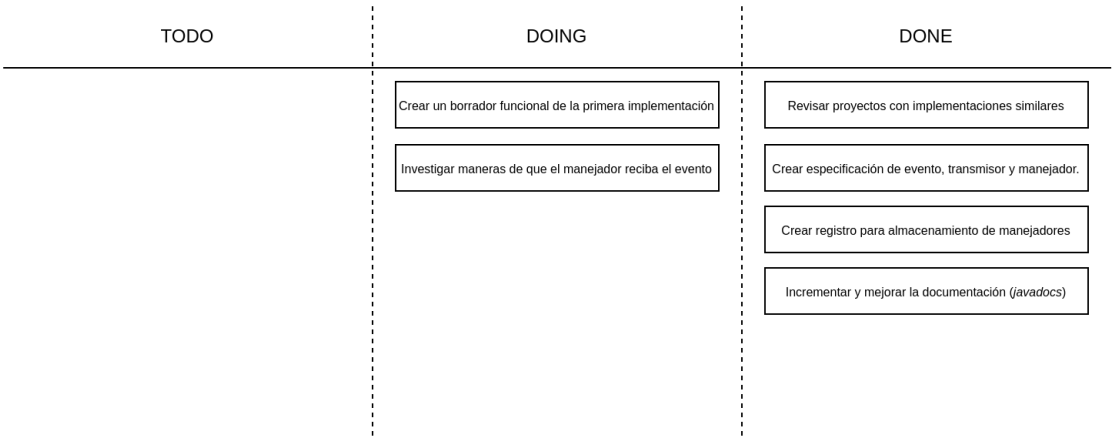


Figura 11: Tablero al final del primer mes

- En el segundo mes, se realizó la división del proyecto en dos módulos: especificación e implementación. Se avanzó en el modelado de los conceptos restantes a nivel de código y sus respectivas pruebas automatizadas. Finalmente, se aplicó el tercer principio de Kanban, que se basa en la mejora continua del flujo de trabajo, ya que se estableció un nuevo algoritmo de prioridad al *Backlog* (ver Figura 12).
  - En la primera iteración, se comenzó a realizar un cambio en la estructura original de paquetes (para agrupar por lo que cambia al mismo ritmo y tiempo) y se siguió con la continua investigación de las diferentes maneras de representar los conceptos a nivel de código. Finalmente, comentar que en este *Sprint* surgió el concepto de gestor de eventos.
  - En la segunda iteración, se consiguió terminar la restructuración de la paquetería. También se fue finalizando la implementación a nivel de código de algunos conceptos pendientes (ej. bus de eventos) y se avanzó en sus correspondientes pruebas unitarias.
  - En la tercera iteración, se detectó de que el bajo nivel de abstracción puede influir al alto (definición de interfaces empleada por el usuario o cliente de la librería). Por ello, surgió la necesidad de priorizar por ordenación topológica el *Backlog* (implementar el bajo nivel primero, es decir, usar el enfoque *bottom-up*).

- En la cuarta iteración, se intentó desbloquear la tarea asociada al almacenamiento de los métodos manejadores. Por ello, se aplicó el *framework* de Cynefin, esta vez no a nivel de proyecto, sino al problema específico (para dar contexto al problema y ayudar en el proceso de toma de decisiones).

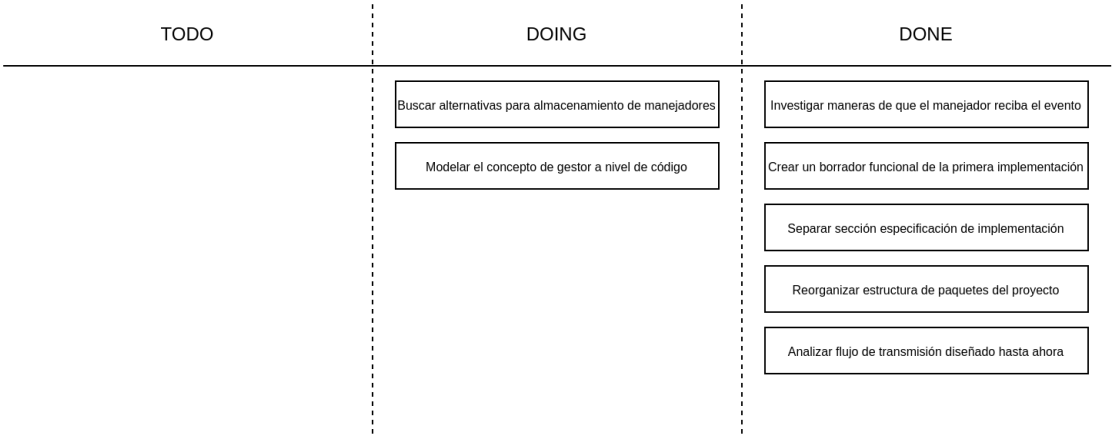


Figura 12: Tablero al final del segundo mes

- En el tercer mes, se siguió con la refactorización del código en general, se consiguió desbloquear la tarea de almacenamiento (aunque ahora se está entre los cuadrantes complejo y complicado del modelo de Cynefin) (ver Figura 13).

  - En la primera iteración, se descubrió durante la realización de las pruebas unitarias que no había manera de que cada manejador conociera el tipo que evento que puede consumir (ya que cada manejador está limitado a consumir un tipo de evento específico).

  - En la segunda iteración, se bloqueó de nuevo la tarea de almacenamiento de manejadores ya que no había manera de informar que un manejador concreto solo puede consumir un tipo de evento concreto. Por ello, se propuso el uso de métodos manejadores decorados, es decir, se aplicó el patrón de diseño decorador para añadir nuevas propiedades a un método de una clase (esta solución se obtuvo por prueba y error).

  - En la tercera iteración, se avanzó en el análisis e implementación de todas las posibles maneras de almacenar los métodos manejadores (mediante el uso de experimentación).

  - En la cuarta iteración, se alteró el *WIP* (*Work In Progress*) limitando el número máximo de tareas a una (característica típica de Kanban). Finalmente, se puso todo el enfoque en la tarea de investigación de las posibles maneras de almacenamiento de los métodos manejadores.

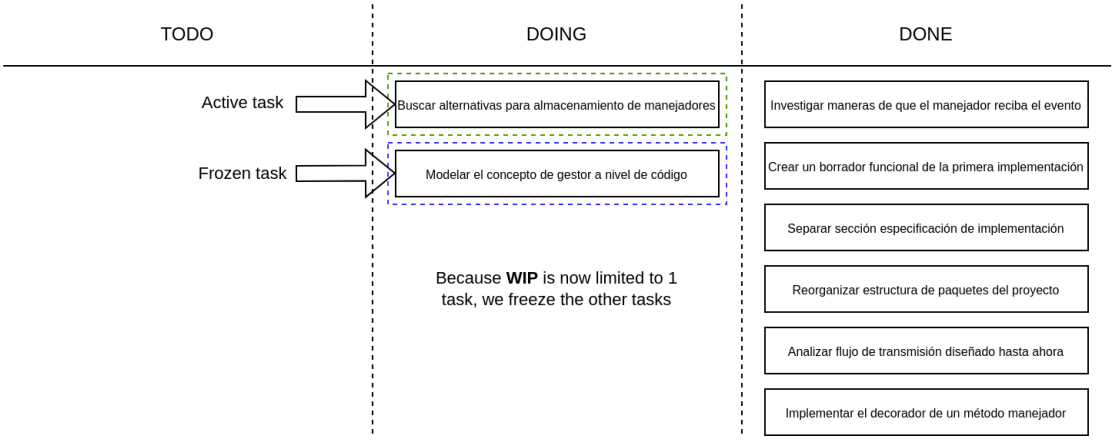


Figura 13: Tablero al final del tercer mes

- En el cuarto mes, se comenzó con una refactorización y reorganización de las estructuras de almacenamiento. También surgió la necesidad de implementar una cola de prioridad doblemente enlazada. Finalmente, se empezó a crear pruebas automatizadas para las clases restantes sin cobertura hasta ahora (ver Figura 14).

  - En la primera iteración, se terminó la implementación del almacenamiento de métodos manejadores y sus respectivas pruebas unitarias. Finalmente se devolvió el *WIP* (*Work In Progress*) a su tamaño original para poder trabajar en tres tareas (como máximo) de manera simultánea.

  - En la segunda iteración, apareció la necesidad de implementar una cola de prioridad para asociar cada tipo de evento a una colección. Para que así, pueda ser respetado el orden o prioridad de recepción del evento transmitido por cada uno de los manejadores.

  - En la tercera iteración, surgió la necesidad de implementar nuevos métodos para flexibilizar la interfaz del cliente (estructuras auxiliares), tarea que se añadió y se categorizó con máxima prioridad en el *Backlog*. También se avanzó en la implementación de la cola de prioridad doblemente enlazada.

  - En la cuarta iteración, se enfocó en terminar de probar las posibles implementaciones (experimentación con TDD purista) y en realizar pruebas automatizadas para el almacenamiento de los manejadores (estructuras auxiliares y cola de prioridad).

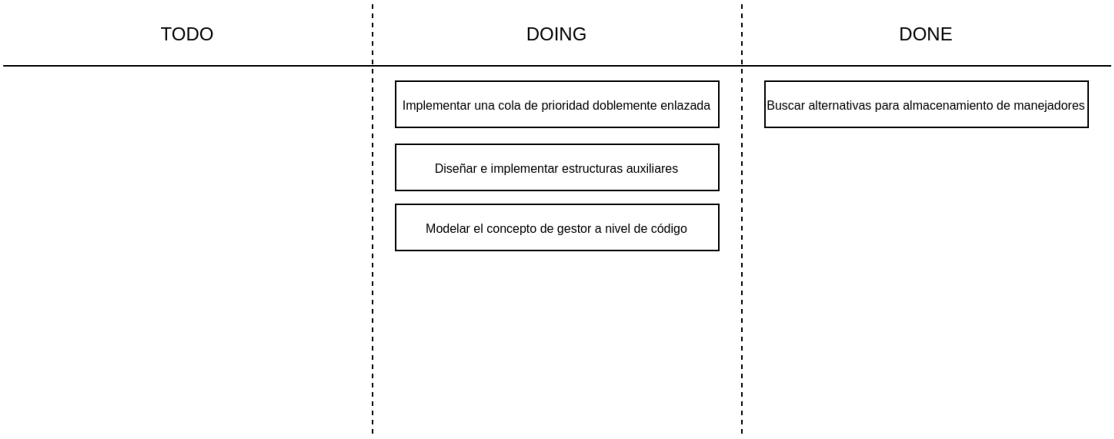
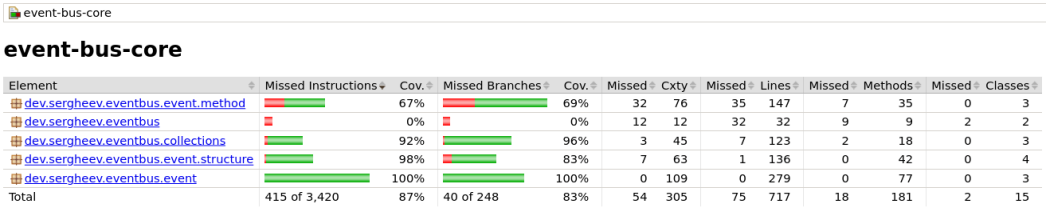


Figura 14: Tablero al final del cuarto mes

- En el quinto mes, el enfoque estuvo en aumentar considerablemente la cantidad de pruebas automatizadas (llegando a disponer de más de 100 pruebas) e ir revisando los informes de la cobertura de código que ayudaron a arreglar problemas de implementación de la cola de prioridad doblemente enlazada. A partir de este mes, la reunión de revisión pasó de ser diaria a semanal, es decir, ahora se realizará una vez a la semana el último día del *Sprint* (ver Figura 16).
    - En la primera iteración, se centró en la revisión del comportamiento incorrecto de la cola de prioridad que hacia fallar las pruebas unitarias. Finalmente, la tecnología de JaCoCo (ver Figura 15) nos ayudó a descubrir las ramas por las que el código se desviaba .


event-bus-core

event-bus-core

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
dev.sergheev.eventbus.event.method		67%		69%	32	76	35	147	7	35	0	3
dev.sergheev.eventbus		0%		0%	12	12	32	32	9	9	2	2
dev.sergheev.eventbus.collections		92%		96%	3	45	7	123	2	18	0	3
dev.sergheev.eventbus.event.structure		98%		83%	7	63	1	136	0	42	0	4
dev.sergheev.eventbus.event		100%		100%	0	109	0	279	0	77	0	3
Total	415 of 3,420	87%	40 of 248	83%	54	305	75	717	18	181	2	15

Figura 15: Informe interactivo de JaCoCo para visualizar ramas de código no alcanzadas

    - En la segunda y tercera iteración, el foco estuvo en aumentar la cobertura de código, cubriendo todos los posibles casos de prueba. Se añadieron pruebas automatizadas a objetos restantes (sin prueba alguna que verificase su comportamiento hasta ahora) y se aumentó la cantidad de pruebas de los objetos que ya disponían de pruebas.
    - En la cuarta iteración, la *Sprint Goal* (meta de esta iteración) fue arreglar lo antes posible varios errores encontrados con las pruebas unitarias fallidas. Estos errores en la implementación de las estructuras de almacenamiento de manejadores fueron encontrados aplicando la técnica *red-green-refactor* (TDD). También se dedicó tiempo a mejorar la calidad de la documentación a nivel de código (con comentarios que indiquen la intención).

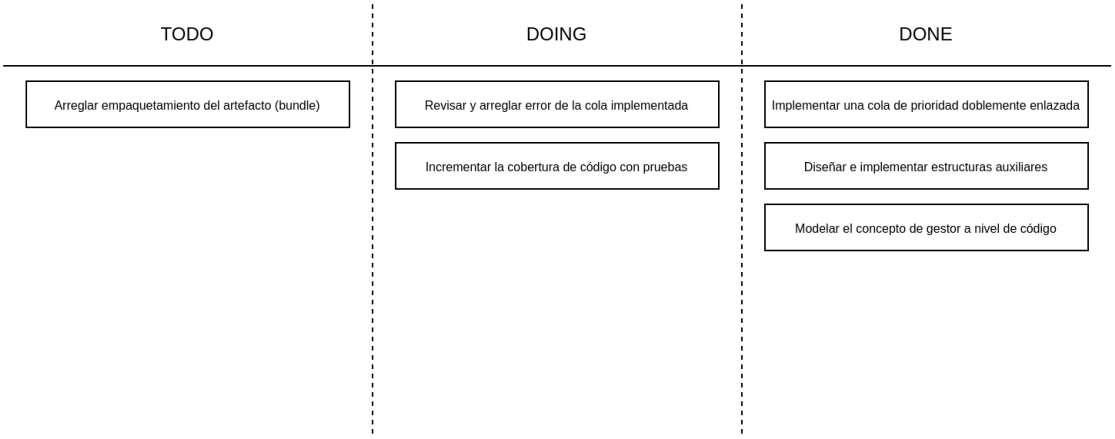


Figura 16: Tablero al final del quinto mes

- En el último mes, se arregló el empaquetamiento (proceso de construcción) de la librería, para unir la parte de especificación e implementación en un único artefacto (también se suele llamar *bundle*). Finalmente, se centró en la refactorización de la nomenclatura y en la mejora de la documentación del código (ver Figura 17).
    - En la primera iteración, se centró en arreglar el empaquetamiento de la especificación e implementación en un solo artefacto (o *bundle*).
    - En la segunda iteración, se enfocó en mejorar la calidad de los comentarios y la documentación, tanto a nivel de código, como a nivel de *wiki* (página especial para la documentación de la librería presente en el repositorio de código).
    - En la cuarta iteración, se dedicó el tiempo a la preparación de la infraestructura de despliegue, es decir, a delegar a una máquina aparte, la automatización de la compilación, la ejecución de pruebas y el análisis estático del código.

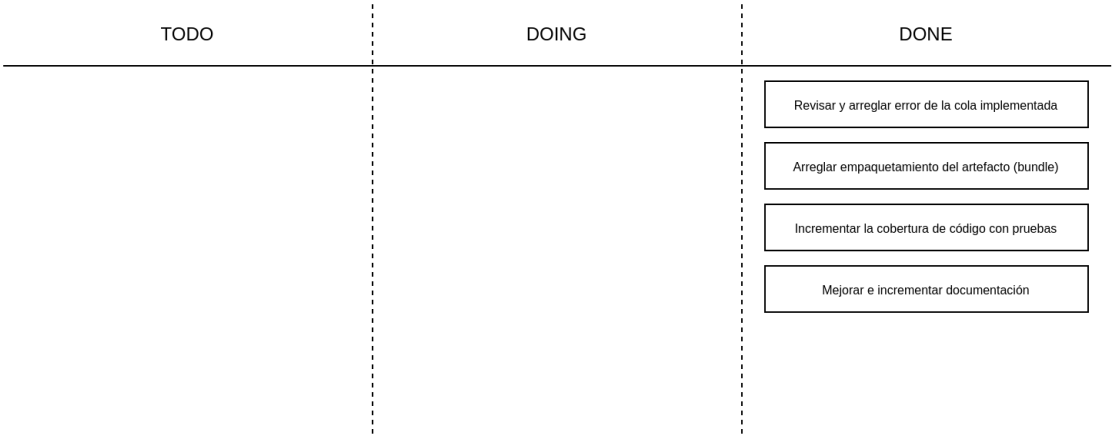


Figura 17: Tablero al final del último mes

Hay que comentar que durante una iteración (o *Sprint*) ocurren varias microevoluciones (tanto cambios conceptuales, como a nivel de código) semanales, o incluso diarios.

Al aplicar el *framework* Cynefin al desarrollo general del proyecto, partimos del cuadrante desorden, y a medida que progresamos, surgen problemas bloqueantes durante el proyecto. Por tanto, nos encontramos los cuadrantes complejo y complicado la mayor parte del tiempo.

- En el cuadrante complejo, se indica qué se desconocen o están desordenados diversos conceptos sobre el dominio de eventos, aunque sí que se dispone de un cercano equilibrio entre buenas prácticas emergentes y conocidas, a pesar de que haya mayor presencia de prácticas que son emergentes.

- En el cuadrante complicado, a parte del equlibrio de prácticas ya mencionado, nos encontramos en un dominio simple, en el sentido que se dispone de pocos conceptos que modelar, aunque haya que investigar entre un rango de posibles modelados mediante experimentación.

3.5. Despliegue

Para este proyecto, al tratarse del desarrollo de una librería, carece de sentido disponer de una infraestructura como código, es decir, tener utilidades para automatizar el despliegue (instalación y configuración) en una máquina. Esto si tendría sentido en caso de tratarse de una aplicación que fuese a usarse por un usuario de manera interactiva (ej. una aplicación web). Por tanto, a la hora de hablar del despliegue de la librería nos referiremos a delegar el proceso de compilación, ejecución de pruebas y publicación del artefacto correspondiente.

Al disponer de pruebas automatizadas para nuestra librería, surge la siguiente cuestión respecto al despliegue: ¿cada cuanto se ejecutarán las pruebas? La idea sería relanzar las pruebas cada vez que se realicen pequeños cambios.

En todo desarrollo, hay un tiempo en el que el código no se está ejecutando, a dicho tiempo se puede denominar periodo de latencia. El código es texto, y podemos realizar todo tipo de cambios que aumentan en cantidad a lo largo del tiempo, dejar que este periodo se alarge es añadir más riesgo (porque podemos equivocarnos al implementar algún aspecto).

Por tanto, se podría extender el entorno de desarrollo, mediante un *script* que ejecute las pruebas cada vez que se realice un guardado, o que cada cierto periodo de tiempo lance las pruebas de manera automática (ej. cada 30 segundos se ejecuten las pruebas).

Otra cuestión que surge es ¿quién debe ejecutar las pruebas? Se pueden lanzar en la máquina del desarrollador, pero como las pruebas son una unidad aislada de nuestro código, podemos trasladar su ejecución a un entorno aparte (delegar esa responsabilidad).

Para este proyecto, por simplicidad, en primer lugar el usuario subirá los ficheros fuente a su repositorio de código, y finalmente accionará de manera manual el despliegue, que automaticamente lanzará: la ejecución de pruebas, la generación de reportes, y finalmente la publicación del artefacto (ver Figura 18).

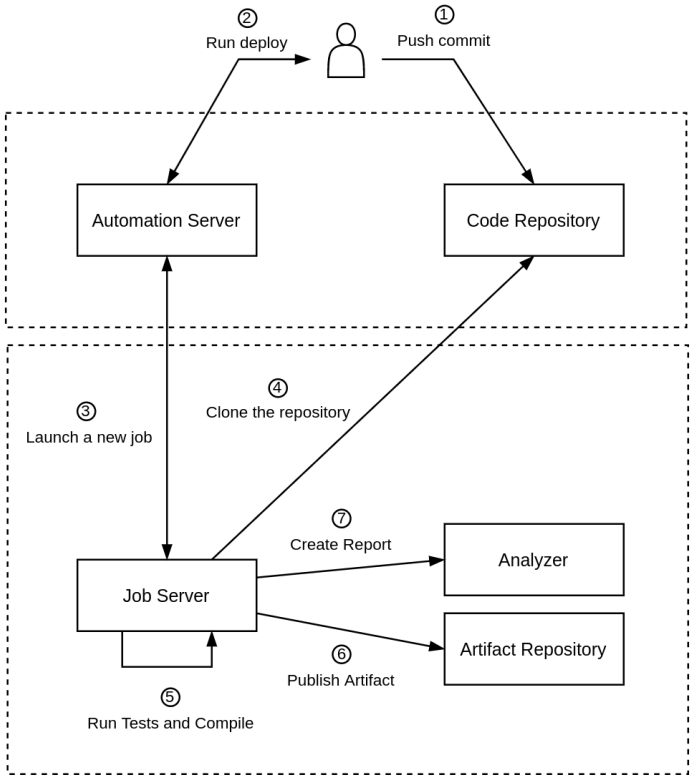


Figura 18: Flujo de la integración continua

Respecto a la parte técnica del depliegue, se ha elegido utilizar Docker [7] para facilitar la portabilidad del entorno de despliegue (contenerización). Las tecnologías individuales que componen la definición del flujo de integración continua son:

- Un repositorio de código, sistema distribuido de versionado que permite a múltiples desarrolladores colaborar simultáneamente en un proyecto sobre una misma base de código fuente. La tecnología elegida por su simplicidad es Gitea [8].
- Un servidor de automatización, es un servidor que permite automatizar trabajos como la compilación del código fuente, la ejecución de pruebas y el despliegue de software en general. Por su madurez (han pasado más de 10 años desde su lanzamiento inicial) se ha elegido Jenkins [9].
- Un analizador de código, es una herramienta para evaluar el código fuente de un proyecto y proveer un análisis estático generando métricas que pueden ayudar a mejorar la calidad de dicho código. Por su popularidad, se usará SonarQube [10].
- Un repositorio de artefactos, es una plataforma que permite publicar artefactos (ej. binarios, imágenes de contenedores, etc) y facilitar su distribución. La tecnología elegida por su simplicidad es Reposilite [11].

[7] Docker - tecnología estandar que permite crear contenedores para aislar procesos
[8] Gitea - repositorio de código para control de versiones de desarrollo de software
[9] Jenkins - servidor de automatización de trabajos de código abierto
[10] SonarQube - plataforma para evaluar código fuente y obtener métricas para mejorar su calidad
[11] Reposilite - repositorio de artefactos ligero especializado para artefactos de Maven

El resumen del flujo de la *pipeline* (secuencia de pasos automatizados) representado en el diagrama (ver Figura 18) es el siguiente:

1. El usuario hace push del *commit* al repositorio de código remoto.
2. El usuario lanza el despliegue desde la interfaz del servidor de automatización.
3. El servidor de automatización lanza un nuevo trabajo al servidor de trabajos.
4. El servidor de trabajos lanza las pruebas automatizadas y compila el proyecto.
5. El servidor de trabajos publica el resultado en el repositorio de artefactos.
6. El servidor de trabajos informa al analizador para crear un reporte sobre el proyecto.

En tecnologías más avanzadas (ej. GitHub [12] o GitLab [13]), los repositorios de código incluyen de forma nativa una sección de integración continua que muestra el progreso del trabajo lanzado, abstrayendo así al desarrollador el conocimiento específico sobre servidores de automatización. Una de las razones por las que no se ha usado GitHub ni GitLab, es porque estas requieren de la adquisición de un plan especial para poder utilizar características avanzadas (como la integración continua de forma nativa), lo cual, no se ajusta al alcance (o *scope*) de este proyecto. En cuanto a GitHub, se trata de una plataforma privativa (o de código cerrado). Por tanto, se puede dar el caso de que el *compliance* de la empresa (que esté desarrollando una dependencia de primeros) imponga la restricción del acceso mediante el uso de la red privada. Por ello, se necesita una solución que sea de código abierto y *self-hosted* (para instalarla en un servidor interno). Finalmente, se podría considerar la versión de código abierto de GitLab, aunque ahora se podría cumplir el *compliance*, ésta igualmente carece de ciertas características avanzadas en su plan *community*.

[12] GitHub - repositorio de código para control de versiones privativo de Microsoft
[13] GitLab - repositorio de código para control de versiones de código abierto de GitLab Inc

Capítulo 4

Experimentos, pruebas y métricas

En este capítulo se comentarán en detalle los diferentes experimentos realizados y sus resultados, las pruebas automatizadas y las métricas obtenidas de los analizadores.

4.1. Experimentos y pruebas

Para visualizar qué implicaciones tiene el adoptar una arquitectura orientada a eventos, se propone la creación de una aplicación que implica al patrón productor/consumidor, en la que se irán produciendo (generando aleatoriamente), encolando y consumiendo (metiendo a un cubo) números.

4.1.1. Antes de usar eventos

En primer lugar, disponemos de un productor y un consumidor, ambos conocen y dependen de manera directa de una cola de productos, meter un producto al cubo representará en nuestro caso la acción consumir un producto (ver Figura 19).

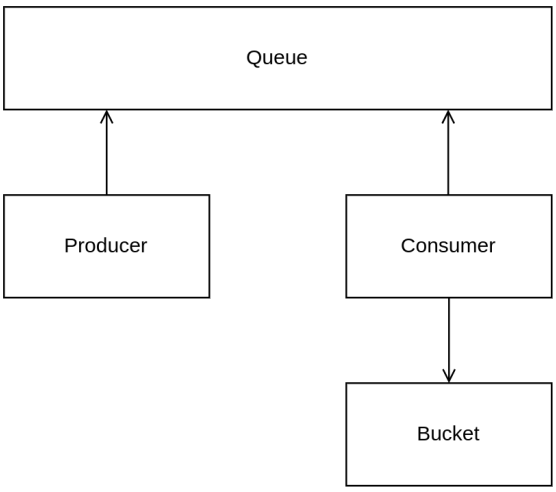


Figura 19: Diagrama previo al uso de eventos

Supongamos el caso en el que ahora se dispone de un bus de eventos, la cuestión que surge es la siguiente: ¿cómo afectan las modificaciones que permiten adoptar la arquitectura orientada a eventos a este ejemplo?

4.1.2. Después de usar eventos

Tras un exhaustivo análisis y la realización de los cambios pertinentes para adoptar una arquitectura orientada a eventos (ver Figura 20), nos podemos percatar de algunas de las mejoras que genera esta adopción a nivel de diseño:

- Antes, el productor necesitaba conocer directamente la estructura de datos en la que se van a almacenar los diferentes productos, ahora se delega dicha responsabilidad al escuchador, ya que cada vez que el productor genera un producto se emite un evento. Por ello, ahora también se podría crear lógica que permita guardar cada producto en una estructura diferente según su tipo (el tipo del producto generado). Esto se conseguiría haciéndo que cada tipo de producto sea consumido por su respectivo manejador.

- En el diseño anterior (ver Figura 19), la lógica de qué realizar con el producto al consumirlo estaba altamente acoplada al consumidor, ahora se delega al manejador (contenido en el escuchador) del evento que se emite.

- Si se quiere agregar, modificar o eliminar lógica sobre el comportamiento del productor y consumidor, se pueden cancelar los eventos de manera dinámica – ya que se deja a los manejadores decidir qué hacer – y ahora también existe la posibilidad de añadir prioridades de ejecución.

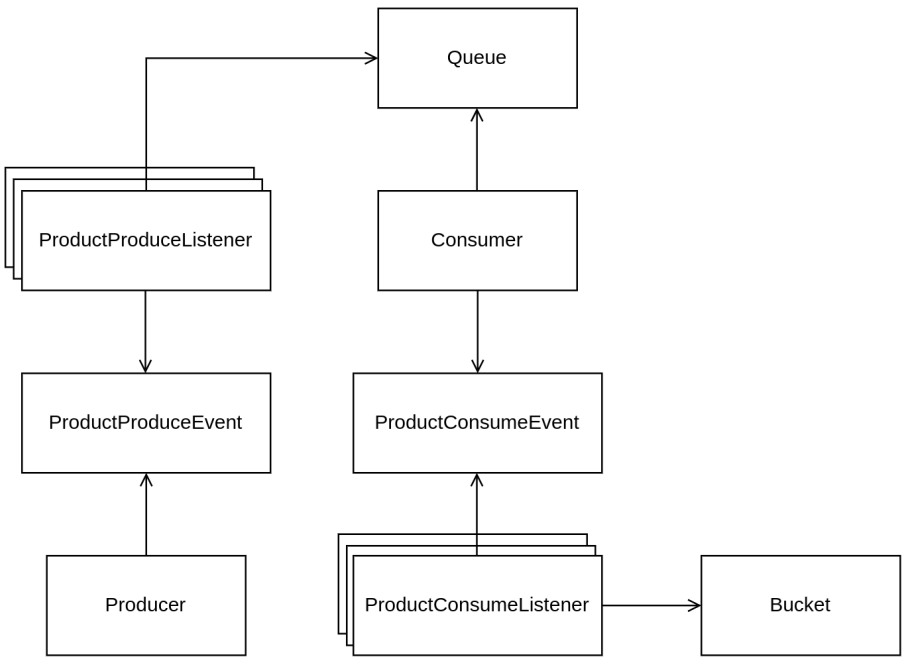


Figura 20: Diagrama posterior al uso de eventos

Aunque el diseño de la aplicación se ha complicado en cierta manera, hay un gran beneficio en cuanto a mantenimiento a largo plazo, ya que se puede modificar, extender o incluso cancelar la ejecución de cierta lógica de manera dinámica, aparte de reducir en gran medida el acoplamiento.

En caso de querer sustituir la cola que almacena productos, por otra estructura de datos más compleja, se reduce la cantidad de cambios a realizar en código y el núcleo se mantiene intacto, es decir, diferimos las decisiones de implementación al contexto de negocio [35].

4.2. Métricas

En esta sección vamos a presentar el cuadrante de deuda técnica, el diagrama de flujo cumulativo, y finalmente el análisis de pruebas y cobertura de código.

4.2.1. Cuadrante de deuda técnica

En primer lugar, hay que comentar que el concepto de deuda técnica [36], se refiere a los compromisos que se tendrán que tomar entre el código ideal y el código necesario para cumplir la fecha límite. Pero no con la visión de que hay que tolerar las malas prácticas de código, sino con el enfoque de que hay mayor valor en hacer algo sencillo y funcional en este momento, pero con la visión de mejorarlo en el futuro.

Si se quisiera categorizar la deuda técnica de este proyecto usando el cudrante de deuda técnica de Martin Fowler [1] [37], estaríamos entre el cuadrante prudente/inadvertido (ver Figura 21) debido a que seguimos las mejores prácticas pero nos percatamos de que emergen mejores maneras de haber hecho las cosas.

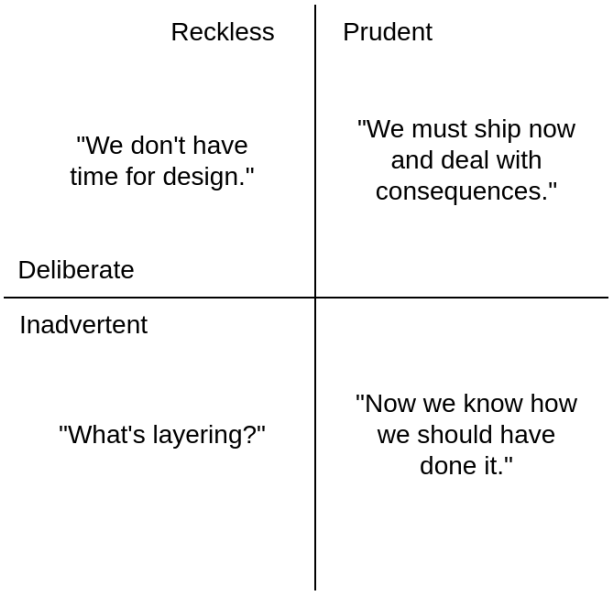


Figura 21: Cuadrante de deuda técnica

[1] Martin Fowler - ingeniero de software, autor y orador internacional sobre desarrollo de software

Esto ocurrió con uno de los aspectos de la librería, concretamente con el gestor de eventos, ya que se intentó diseñar una interfaz más flexible que cubriera la mayor parte de todas las operaciones posibles. Se consiguió el objetivo, pero se creó cierto nivel de complejidad a nivel de implementación, debido a que el gestor actúa como una *façade* [38], uniendo diferentes estructuras de almacenamiento que no encajan de otra manera y proveyendo así una interfaz unificada. La razón de existencia de dichas estructuras de almacenamiento es reducir la complejidad en tiempo de ejecución de las operaciones que flexibilizan la interfaz del gestor.

La solución a ello sería disponer de dos tipos de gestor de eventos, es decir, promover el uso de la composición de objetos, proveyendo una factoría simple que permitiese elegir entre un gestor de eventos sencillo y otro complejo, dejando al cliente decidir según sus necesidades y así evitando la explosión de conocimiento (forzando el uso de una gran interfaz).

4.2.2. Diagrama de flujo cumulativo

Los diagramas de flujo cumulativo [39] tabulan la cantidad de elementos completados de cada categoría a lo largo del tiempo, este tipo de gráfico permite identificar patrones que ayudan a asesorar la salud del proceso de *delivery* y diagnosticar problemas que necesitan nuestra atención.

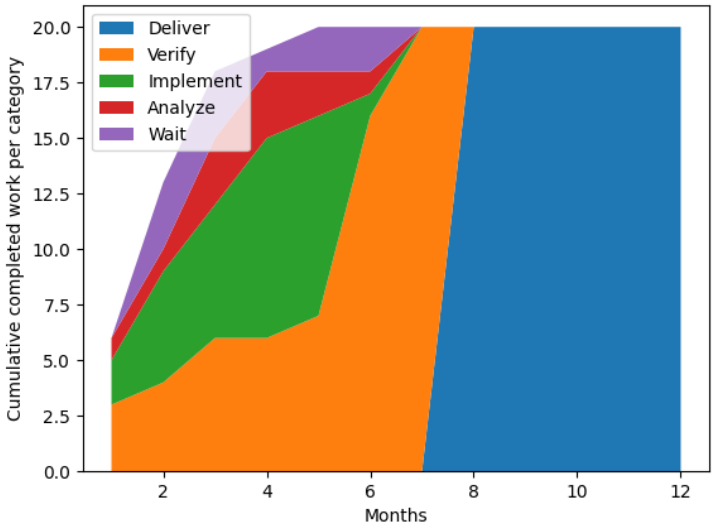


Figura 22: Diagrama de flujo cumulativo del proyecto

Nos encontramos con un diagrama de flujo en estado saludable (ver figura 22), debido a que todos los elementos propuestos se han completado, a pesar de que el proceso de entrega solo se realiza una vez el software esté completo (en el sentido de que funciona, pero no hay tiempo para más mejoras).

El motivo de tardar tanto en llegar a la fase de *delivery* – que ocurre en nuestro caso – es debido a que el proyecto se basa en la investigación continua, es decir, el uso de prácticas

emergentes y la refactorización constante, por tanto, la entrega no se realiza hasta que se finalize el proyecto.

Se puede apreciar la aplicación de la estrategia iterativa, porque poco a poco va emergiendo el todo, e incremental, debido a que cada fase (análisis, implementación y verificación) se realiza varias veces en cada iteración. Finalmente comentar que el motivo del *spike* (pico) en la categoría de verificación, es debido a que ese periodo de tiempo se dedicó especialmente a la realización de pruebas automatizadas y la cobertura de código.

4.2.3. Análisis de pruebas y cobertura de código

Tras el análisis con la tecnología SonarQube, se nos reportan las siguientes estadísticas:

- Niguna vulnerabilidad, *bug* o *new security hotspot*.
- Un 88 % de cobertura de código (con un total de 103 pruebas automatizadas).
- Una deuda técnica de 6 horas y 51 minutos.
- Un 8,2 % de código duplicado.
- Una cantidad de 28 *code smells*.

El resultado del informe generado tras el análisis de código es muy positivo, ya que el proyecto no dispone de ninguna vulnerabilidad o agujero de seguridad, se dispone de casi un 90 % de cobertura de código, y de un tiempo de deuda técnica bastante reducido (ver Figura 23).

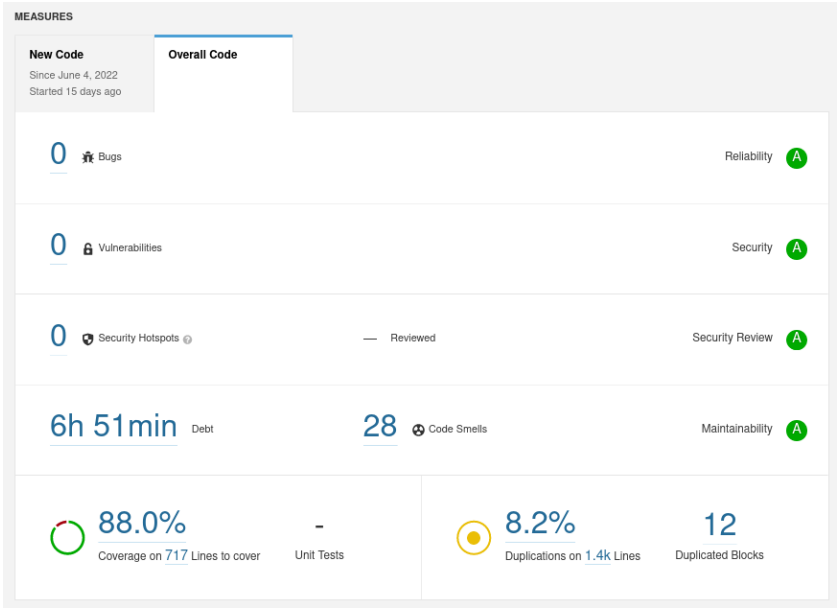


Figura 23: Estadístiscas resultantes del análisis de Sonar

Uno de los motivos de las casi siete horas de deuda técnica, es el supuesto duplicado de código, pero en caso de haber dos casos de uso independientes – aunque dispongan de una porción similar de código – en realidad dicho código no es código duplicado.

Si se hace caso a dicho reporte y no se llegan a cuestionar motivos los resultados, se puede llegar a refactorizar a peor, llegando a generar un alto acoplamiento de los diferentes casos de uso que existen, por el mero hecho de disponer de una porción de código similar.

En nuestro caso, las repeticiones son de pequeños bloques de código, y carece de sentido abstraer eso en una función más generica, en caso de hacerlo, se saltaría de función pequeña a función pequeña, dificultando así la comprensión y la depuración, creando por tanto complejidad.

En cuanto a los *code smells*, la mayoría son debido a que se nos sugiere extraer el mensaje de una validación en una constante, y la minoría sobre una clase abstracta, la cual no dispone de ningún *test* (prueba automatizada), pero en realidad dicha clase es un patrón de código limpio [40] aunque no se detecte como tal (ver Figura 24).

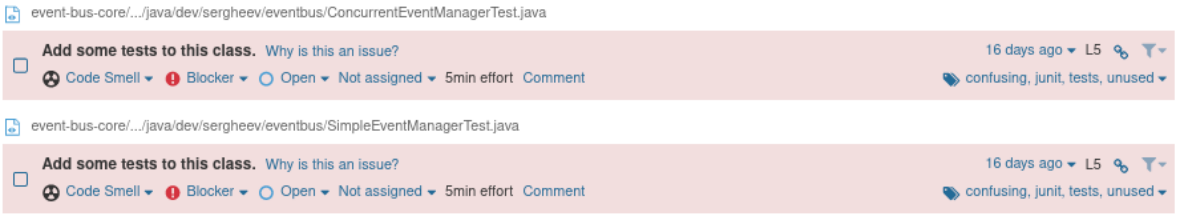


Figura 24: Code smells erróneos reportados por Sonar

Capítulo 5

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones y las posibles continuaciones de este trabajo.

5.1. Conclusiones

El producto resultante es un bus de eventos completamente descoplado, que sigue las mejores prácticas de calidad de software. Por tanto, podemos afirmar que se han cumplido todos los objetivos propuestos en el capítulo inicial:

- La librería resultante ha sido codificada en lenguaje *vanilla* (lenguaje puro) y se comporta como una dependencia mínima, es decir, no incluye ningún comportamiento extra no relacionado con eventos. Finalmente, hay que mencionar que esta librería no depende de terceros (código externo) y volver a remarcar que el desarrollo se centra totalmente en la funcionalidad de eventos.
- Se dispone de una documentación completa de las decisiones de desarrollo y diseño que se han ido tomando presentes en este documento. Aparte, también se dispone de una explicación en la *wiki* del repositorio de código y así como de comentarios explicativos de alta calidad en toda la base de código.
- Al tener métricas que demuestren casi un 90 % de cobertura de código (con 103 pruebas automatizadas), podemos afirmar con gran seguridad que el comportamiento de la librería es el esperado.

5.2. Trabajos futuros

Un posible trabajo futuro, sería disponer de una arquitectura de computación algorítmica distribuida por cluster con *plugins*, que permitiéra añadir o modificar de manera dinámica los diferentes algoritmos de computación.

Cada uno de los nodos de un cluster constaría de diversos elementos como un *core* (con las reglas de negocio), una infraestructura (con el *delivery system*, especialmente para *streaming*) y luego una parte de eventos, comandos y *plugins* (ver figura 25).

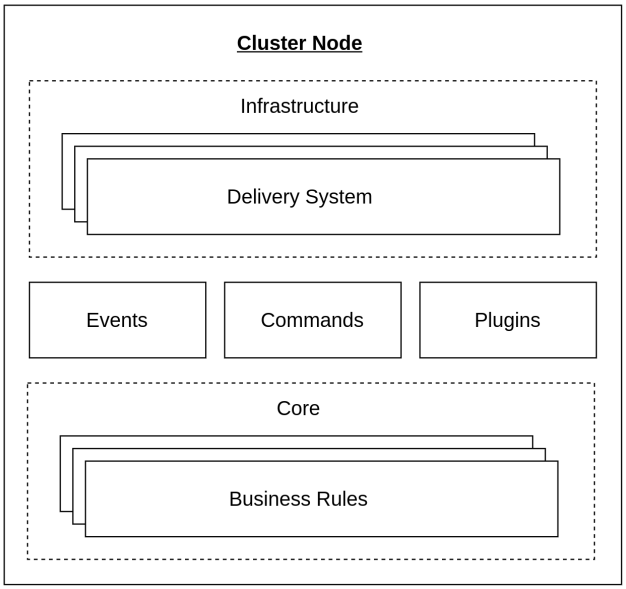


Figura 25: Diagrama que representa de que constaría cada cluster

La parte ya existente, sería la de eventos. Lo siguiente, sería diseñar el resto de partes. Una de ellas siendo una estructura similar al patrón productor/consumidor, que permita la transmisión de eventos al resto de nodos que formen parte de un *cluster* (infraestructura para *streaming* de eventos).

Para ello se necesitaría usar serialización o diseñar un elemento específico que permitiese realizar el mapeo de cualquier objeto que sea un evento a un formato concreto que fuera texto o secuencia de *bytes* antes de transmitirlo – se necesitaría un formato estandár de transmisión de información –.

Si se comienza a pensar a alto nivel de diseño de sistemas, también en cierto punto habría que centrarse en cómo diseñar un sistema de *leader election*. En caso de que el nodo líder se cayese, ¿cuál sería el mecanismo para elegir otro nodo líder?, ¿habría un sistema de *scoring* o se produciría esto de manera aleatoria?

Luego finalmente, habría que considerar la manera en la que los *clusters* se deberían comunicar entre ellos, cómo se balancearía la carga, y también cómo se podría realizar el escalado horizontal de manera dinámica, mediante una interfaz web, esto siendo uno de los objetivos finales.

Bibliografía

[1] B. Jack Copeland. "The Modern History of Computing". En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University, 2020 (vid. pág. 1).

[2] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Preface. Addison-Wesley, sep. de 2017, pág. XX. ISBN: 978-0134494166 (vid. pág. 1).

[3] Jim Keller. "Moore's Law is Not Dead". En: *EECS Colloquium*. Fall 2019. Berkeley EECS, sep. de 2019 (vid. pág. 1).

[4] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Preface. Addison-Wesley, sep. de 2017, pág. XXII. ISBN: 978-0134494166 (vid. pág. 1).

[5] José F. Vélez Serrano. *Técnicas Avanzadas de Diseño de Software: Una introducción a la Programación Orientada a Objetos usando UML y Java*. Capítulo 1. Introducción a la Programación Orientada a Objetos: Complejidad del Software, Origen y Tratamiento. Universidad Rey Juan Carlos, 2009, págs. 11-12 (vid. pág. 1).

[6] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Chapter 17. Smells and Heuristics: Avoid Transitive Navigation. Pearson, ago. de 2008, pág. 307. ISBN: 978-0132350884 (vid. pág. 2).

[7] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Chapter 16. Independence: Leaving Options Open. Addison-Wesley, sep. de 2017, pág. 150. ISBN: 978-0134494166 (vid. pág. 2).

[8] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Chapter 14. Component Coupling: The Stable Dependencies Principle, Not All Components Should Be Stable. Addison-Wesley, sep. de 2017, pág. 123. ISBN: 978-0134494166 (vid. pág. 2).

[9] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 3. Dependencies and Layering. Microsoft Press, abr. de 2017, págs. 69-70. ISBN: 978-1509302581 (vid. pág. 2).

[10] José F. Vélez Serrano. *Técnicas Avanzadas de Diseño de Software: Una introducción a la Programación Orientada a Objetos usando UML y Java*. Capítulo 1. Introducción a la Programación Orientada a Objetos: Complejidad del Software, Paradigmas de Programación. Universidad Rey Juan Carlos, 2009, pág. 14 (vid. pág. 2).

[11] Eric Freeman. *Head First Design Patterns (A Brain Friendly Guide)*. Chapter 13. Patterns In The Real World: Design Pattern Defined. O'Reilly Media, nov. de 2004, pág. 585. ISBN: 978-0596007126 (vid. pág. 2).

[12] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley professional computing series)*. Chapter 5. Behavioral Patterns: Chain Of Responsibility. O'Reilly Media, oct. de 1994, pág. 223. ISBN: 978-0201633610 (vid. pág. 2).

[13] Hugh Taylor. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise (English Edition)*. Introduction. Addison-Wesley, feb. de 2009, pág. 1. ISBN: 978-0321322118 (vid. pág. 2).

[14] *Characteristics of Event-Driven Architecture*. The Ideal EDA. 2009. URL: https://www.informit.com/articles/article.aspx?p=1327184&seqNum=3 (vid. pág. 3).

[15] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 3. Dependencies and Layering: Third-Party Dependencies. Microsoft Press, abr. de 2017, pág. 76. ISBN: 978-1509302581 (vid. pág. 3).

[16] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 3. Dependencies and Layering: First-Party Dependencies. Microsoft Press, abr. de 2017, pág. 75. ISBN: 978-1509302581 (vid. pág. 3).

[17] *Leaky Abstraction*. The Law of Leaky Abstractions. 2021. URL: https://en.wikipedia.org/wiki/Leaky_abstraction (vid. págs. 4, 16).

[18] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Chapter 21. Screming Architecture: But What About The Web, Purpose Of An Architecture. Addison-Wesley, sep. de 2017, pág. 197. ISBN: 978-0134494166 (vid. pág. 4).

[19] Kent Beck. *Implementation Patterns*. Chapter 10. Evolving Frameworks. Addison Wesley, oct. de 2007, pág. 117. ISBN: 978-0321413093 (vid. pág. 5).

[20] Winston Royce. "Managing the Development of Large Software Systems". En: *Proceedings of IEEE WESCON, 26*. IEEE, ago. de 1970 (vid. pág. 7).

[21] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 1. Introduction to Scrum: Scrum vs Waterfall. Microsoft Press, abr. de 2017, pág. 6. ISBN: 978-1509302581 (vid. pág. 7).

[22] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 1. Introduction to Scrum:Scrum is Agile. Microsoft Press, abr. de 2017, pág. 4. ISBN: 978-1509302581 (vid. pág. 7).

[23] Kent Beck. "Manifiesto por el Desarrollo Ágil de Software". En: 2001 (vid. pág. 8).

[24] Javier Garzás Parra. "Veterano Ciclo de Vida Iterativo e Incremental". En: 2010. URL: https://www.javiergarzas.com/2010/01/veterano-ciclo-de-vida-iterativo-incremental.html (vid. pág. 8).

[25] Hirotaka Takeuchi y Ikujiro Nonaka. "The New New Product Development Game". En: ene. de 1986. URL: https://hbr.org/1986/01/the-new-new-product-development-game (vid. pág. 8).

[26] Schwaber y Jeff Sutherland. "La Guía Definitiva de Scrum". En: nov. de 2020. URL: https://scrumguides.org/scrum-guide.html (vid. pág. 8).

[27] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 1. Introduction to Scrum: Sprint Planning. Microsoft Press, abr. de 2017, pág. 31. ISBN: 978-1509302581 (vid. pág. 9).

[28] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 2. Introduction to Kanban: Kanban Quickstart. Microsoft Press, abr. de 2017, págs. 45-46. ISBN: 978-1509302581 (vid. pág. 10).

[29] Orderly Disruption Limited y Daniel S. Vacanti. "La Guía de Kanban". En: dic. de 2020. URL: https://kanbanguides.org/espanol (vid. pág. 10).

[30] Dave Snowden. "Cynefin Framework". En: abr. de 2022. URL: https://en.wikipedia.org/wiki/Cynefin_framework (vid. pág. 10).

[31] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 1. Introduction to Scrum: Cynefin. Microsoft Press, abr. de 2017, pág. 5. ISBN: 978-1509302581 (vid. pág. 11).

[32] Len Bass. *Software Architecture in Practice, Second Edition*. Chapter 18. Building Systems from Off-the-Shelf Components: Architectural Mismatch. Addison Wesley, abr. de 2003. ISBN: 978-0321154958. URL: https://people.ece.ubc.ca/matei/EECE417/BASS/ch18lev1sec2.html (vid. pág. 13).

[33] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 5. Testing: Test Doubles. Microsoft Press, abr. de 2017, pág. 160. ISBN: 978-1509302581 (vid. pág. 13).

[34] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 5. Testing: What is TDD? Microsoft Press, abr. de 2017, págs. 179-181. ISBN: 978-1509302581 (vid. pág. 14).

[35] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Chapter 17. Boundaries: Drawing lines. Addison-Wesley, sep. de 2017, pág. 160. ISBN: 978-0134494166 (vid. pág. 31).

[36] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 1. Introduction to Scrum: Technical Debt. Microsoft Press, abr. de 2017, págs. 20-21. ISBN: 978-1509302581 (vid. pág. 31).

[37] Martin Fowler. "Technial Debt Quadrant". En: oct. de 2009. URL: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html (vid. pág. 31).

[38] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley professional computing series)*. Chapter 4. Structural Patterns: Facade. O'Reilly Media, oct. de 1994, pág. 185. ISBN: 978-0201633610 (vid. pág. 32).

[39] Gary McLean Hall. *Adaptive Code: Agile coding with design patterns and SOLID principles (Developer Best Practices)*. Chapter 2. Intro to Kanban: Cumulative Flow Diagrams. Microsoft Press, abr. de 2017, págs. 59-64. ISBN: 978-1509302581 (vid. pág. 32).

[40] Joshua Kerievsky. *Refactoring to Patterns*. Chapter 6. Creation: Introduce Polymorphic Creation with Factory Method. Addison Wesley, ago. de 2004, págs. 88-95. ISBN: 978-0321213358 (vid. pág. 34).