



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

ADAMS

Agile Development of the Aircraft Management System

Desarrollo Ágil del Sistema de Gestión de Aeronaves

Realizado por

**ANTONIO JESÚS CASTAÑO CAMINO
52228657-G**

Cotutor

**IRENE ALEJO TEISSIÈRE
28814859-E
Ingeniera De Área Senior de FADA-CATEC**

Dirigido por

PABLO TRINIDAD MARTÍN-ARROYO

Departamento

LENGUAJES Y SISTEMAS INFORMÁTICOS

Sevilla, Septiembre de 2013

Agradecimientos

Para empezar, agradecer su labor a mi tutor, quien ha mostrado en todo momento su confianza en el proyecto y se ha mostrado siempre disponible e interesado ante cualquier necesidad.

Por otro lado me gustaría, aunque es imposible, nombrar a todos y cada uno de los compañeros que me han apoyado durante la carrera y que me ayudaron de una forma u otra a sobrellevar las largas horas en las aulas y la biblioteca. A todos les doy las gracias por este tiempo compartido y por haber encontrado una gran amistad en más de uno de ellos.

Gracias también a mis compañeros de trabajo, con mención especial a Irene Alejo, que confiaron en mi para entrar a trabajar con ellos y que desde el primer momento me hicieron sentir uno más del equipo. Siempre han estado disponibles para mis dudas y han hecho del trabajo una experiencia estupenda día tras día. Además esta documentación no sería lo que es ni estaría revisada tan a fondo sin vuestra ayuda. Agradecerles también el haberme introducido al desarrollo ágil, que ha cambiado de golpe mi forma de trabajo y de entender el desarrollo de software.

Agradecer inmensamente todo el apoyo y fé constante de Alba, quien me ha acompañado en prácticamente todos mis años de estudio, para los momentos buenos y los malos, sabiendo sacar siempre lo mejor de mi. Por todos estos años y los que vendrán, gracias.

Por último y no por ello menos importante,a mis padres, que siempre lo dieron todo por sus hijos y que, sin su paciencia y buena fé, no podría haber llegado hasta aquí. Sin vuestra ayuda y cariño nada hubiera sido posible, gracias por estar siempre ahí.

Índice general

I Prólogo	1
1. Introducción	3
1.1. PLANET	3
1.2. Motivación: Desarrollo Ágil	5
1.3. Objetivos del proyecto	5
1.3.1. Objetivos orientados a la metodología	6
1.3.2. Objetivos orientados a la técnica	6
1.3.3. Objetivos personales	7
1.4. Estructura del documento	8
II Metodologías	9
2. Introducción a las Metodologías Ágiles	11
2.1. Programación Extrema	15
2.1.1. Valores	15
2.2. El Manifiesto Ágil	17
3. Herramientas Ágiles	21
3.1. Programación por parejas	21
3.2. T.D.D.	22
3.3. Scrum	24
3.3.1. Características Principales	24
3.3.2. Elementos de Scrum	24
3.3.3. Reuniones en Scrum	28
3.4. Kanban	29
3.5. Principios S.O.L.I.D.	31
3.5.1. Descripción de los Principios	32
3.6. Integración Continua	33

3.6.1. I.C. aplicado al proyecto	34
4. Desarrollo Ágil en el AMS	35
4.1. Sentando las bases	36
 III Estudio de las Tecnologías	 39
5. Herramientas para el desarrollo	41
5.1. Programación Orientada a Objetos	41
5.1.1. Características de la P.O.O.	42
5.2. POJO	44
5.3. Acoplamiento	45
5.4. Red de Petri	46
5.4.1. Propiedades y Validación	48
5.4.2. Ventajas frente a Grafos de estados	49
5.5. Patrones de Diseño	49
5.5.1. Patrón Singleton	52
5.5.2. Patrón Composite	53
5.5.3. Patrón Builder	54
5.5.4. Patrón Facade	56
5.5.5. Patrón Command	57
5.5.6. Patrón Observer	58
5.5.7. Patrón Interceptor	60
5.5.8. Patrón de Inversión de Control	60
5.5.9. Patrón MVC	61
5.6. Java	63
5.7. JUnit	65
5.8. DDS-RTI	65
5.8.1. Arquitectura DDS	66
5.9. Google Protocol Buffer	68
5.10. Boost	71
5.11. XML	71
5.12. XStream Parser	73
5.13. HTML	74
5.14. JSP	75
5.14.1. JSTL	76
5.15. Servlets	77
5.15.1. Contenedor de Servlets	78
5.16. Struts 2	81
5.16.1. Componentes	82

5.17. Spring	88
5.17.1. Inyección de Dependencias en Spring	89
5.17.2. Programación Orientada a Aspectos	90
5.17.3. Componentes de Spring	91
5.18. Maven	93
5.18.1. El POM	93
6. Control de Versiones	97
6.1. Arquitecturas de almacenamiento	98
6.1.1. Mercurial	100
IV Desarrollo del Proyecto	101
7. Planificación	103
7.1. Pasos a realizar	104
7.2. Historias de Usuario	104
7.3. Tareas	107
8. Diseño e Implementación	109
8.1. Montaje del entorno	111
8.1.1. Historia de usuario y Diagrama de BurnDown	112
8.2. Desarrollo de la Red de Petri	113
8.2.1. Objeto Transition	113
8.2.2. Objeto Place	114
8.2.3. Objeto PetriNet	114
8.2.4. Objeto PetriNetTransitionManager	116
8.2.5. Objeto PetriNetTransitionMatcher	117
8.2.6. Objeto PetriNetGenerator	118
8.2.7. Objeto PetriNetFramework	120
8.2.8. Historias de usuario y pruebas realizadas	120
8.3. Intérprete de expresiones	122
8.3.1. Expresiones regulares	122
8.3.2. Construcción de expresiones regulares en Java	124
8.3.3. Objeto RegularExpressionValidator	125
8.3.4. Objeto ExpressionParser	127
8.3.5. Historias de usuario y Pruebas realizadas	129
8.3.6. Diagrama de BurnDown	130
8.4. Parseador de XML	131
8.4.1. Historias de usuario y Pruebas realizadas	133
8.4.2. Diagrama de BurnDown	134

8.5. Señales y DDS	135
8.5.1. DDS Middleware	136
8.5.2. Capa de comunicación interna	136
8.5.3. Manejador de señales	137
8.5.4. Historias de usuario y Pruebas realizadas	138
8.5.5. Diagrama de BurnDown	139
8.6. Aplicación Web	140
8.6.1. Modelo	141
8.6.2. Vista	141
8.6.3. Controlador	144
V Conclusiones	147
9. Valoraciones y objetivos	149
9.1. Valoraciones	149
9.2. Objetivos Cumplidos	150
9.2.1. Objetivos orientados a la metodología	150
9.2.2. Objetivos orientados a la técnica	151
9.2.3. Objetivos personales	151
VI Apéndices	153
10. Ejemplos de Uso	155
10.1. Reserva de vuelo, despegue y aterrizaje para un UAV	157
10.2. Reserva de vuelo, despegue y aterrizaje para tres UAVs	165
10.3. Detección y retirada de un obstáculo	168
11. Hibernate	171
11.1. Introducción a Hibernate	171
11.2. ORM	172
11.3. Funcionamiento de Hibernate	172
Glosario	175

Indice de tablas

5.1. Tabla de los <i>Interceptors</i> más conocidos.	84
8.1. Símbolos comunes de concordancia.	123
8.2. Conjunto de metacaracteres disponibles.	123
8.3. Conjunto de cuantificadores disponibles.	124

Índice de figuras

1.1. Escenario de PLANET	4
1.2. Algunos de los participantes del Proyecto PLANET	4
2.1. Cómo fucionan realmente los proyectos.	12
2.2. Ciclo de vida del Desarrollo ágil de software.	14
2.3. Ciclo de vida de un proyecto usando eXtreme Programming.	17
2.4. El Manifiesto Ágil.	17
3.1. Sátira del Pair Programming.	22
3.2. Algoritmo de T.D.D.	23
3.3. Logo de Scrum.	24
3.4. Cómo entender un Diagrama de BurnDown	27
3.5. Diagrama de BurnDown.	27
3.6. Proceso de iteración en Scrum.	28
3.7. Ejemplo de pizarra en Kanban.	31
4.1. Pizarra de Scrum.	36
5.1. Ejemplo de funcionamiento de una red de Petri simple.	48
5.2. Diagrama de Clases del Singleton.	52
5.3. Diagrama de Clases del Composite.	54
5.4. Diagrama de Clases del <i>Builder</i>	54
5.5. Diagrama de Clases del <i>Facade</i>	56
5.6. Diagrama de Clases del Command.	58
5.7. Diagrama de Clases del Observer.	59
5.8. Arquitectura MVC.	63
5.9. Logos de OpenJDK y Sun Microsystems.	64
5.10. Logo de JUnit.	65
5.11. Logos de DDS y RTI.	66
5.12. Archiectura de DDS.	68
5.13. Logo de Google Protocol Buffer.	69
5.14. Logo de Boost.	71

5.15. Logo de XML.	72
5.16. Logo de XStream.	74
5.17. Pasos que siguen las páginas JSP.	75
5.18. Ejecución de Servlets.	77
5.19. Procesamiento de peticiones con un contenedor de Servlets.	78
5.20. Arquitectura de componentes de Tomcat.	79
5.21. Jerarquía estándar de Tomcat.	80
5.22. Logo de Tomcat.	81
5.23. Logo de Struts 2.	81
5.24. Secuencia de pasos de Struts2.	87
5.25. Módulos de Spring.	92
5.26. Logo de Maven.	95
6.1. Logos de <i>Mercurial</i> y <i>TortoiseHg</i>	100
8.1. Red de Petri a implementar.	110
8.2. Diagrama de BurnDown para Sprint 2.	112
8.3. Diagrama de BurnDown para Sprint 5.	131
8.4. Diagrama de BurnDown para Sprint 6.	135
8.5. Diagrama de BurnDown para Sprint 7.	140
8.6. Vista de la página principal del AMS.	142
8.7. Vista del Aeropuerto en el AMS.	143
8.8. Vista de la página principal del AMS.	144
10.1. Red de Petri a implementar.	156
10.2. Reservando un vuelo.	157
10.3. Vuelo reservado en la vista principal.	157
10.4. Vuelo reservado en la vista de control.	158
10.5. Vuelo reservado en la vista del aeropuerto.	158
10.6. Hora del despegue en la pantalla del aeropuerto.	159
10.7. Señales en la pantalla de control.	159
10.8. El UAV se dirige a la pista.	160
10.9. Estado de las señales en el control.	160
10.10El UAV está en la pista.	161
10.11El UAV puede despegar.	161
10.12Señales en el despegue.	161
10.13El UAV está volando.	162
10.14Señales tras el despegue.	162
10.15El UAV puede aterrizar.	163
10.16Señales al aterrizar.	163
10.17El UAV se dirige al hangar.	164

10.18Vista de las señales en la pantalla de controles avanzados.	164
10.19El UAV vuelve al hangar.	164
10.20Señales tras el aterrizaje.	165
10.21Horas de despegue en vista principal.	165
10.22UAVs disponibles en el aeropuerto.	166
10.23Nueva disponibilidad de reservas.	166
10.24Ya no hay hora reservada.	167
10.25Contador de UAVs actualizado.	167
10.26Contador actualizado tras despegue.	167
10.27Detección de un obstáculo.	168
10.28Señales en la pantalla de control.	168
10.29UGV enviado.	169
10.30Estado de las señales.	169
10.31Pista despejada de nuevo.	169
11.1. Logo de Hibernate.	171

Parte I

Prólogo

Capítulo 1. Introducción

CATEC[10], empresa de investigación en el mundo de la aeronáutica, desarrolla proyectos de diferente índole con prestigio internacional como, por ejemplo, PLANET.

1.1. PLANET

PLAtform for the deployment and operation of heterogeneous NETworked cooperating objects [9], es un proyecto europeo dentro del FP7[1], coordinado por la University of Duisburg-Essen (UDE)[2] y con más de diez partners europeos entre los que cabe destacar: DLR[3], Selex Galileo[4], Boeing Research and Technology Europe[5] y el CSIC[6]. El Proyecto PLANET tiene como objetivo la integración de distintos sistemas inteligentes como redes de sensores, UAVs y UGVs en dos entornos de aplicación: La vigilancia de la Reserva Biológica Doñana[7] con un muy alto valor ecológico y sensible a los efectos de la contaminación, y del aeródromo altamente automatizado en el que la seguridad juega un papel importante y donde la comunicación inalámbrica y de cooperación técnicas plantean grandes desafíos.

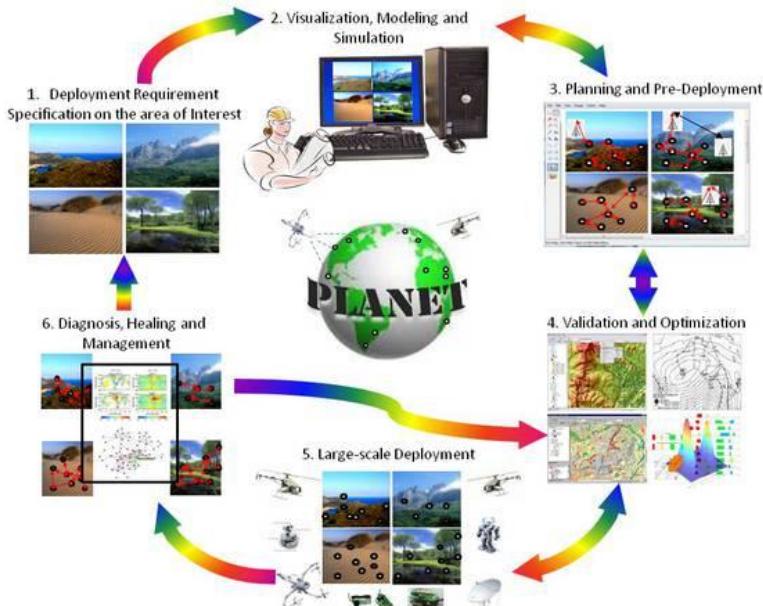


Figura 1.1: Escenario de PLANET.



Figura 1.2: Algunos de los participantes del Proyecto PLANET.

Es el segundo escenario en el que se sitúa este desarrollo. Se construirá una aplicación que gestione y visualice completamente el entorno de un aeródromo automatizado, el Aircraft Management System (AMS), comunicándose con protocolos DDS RTI^[8] entre los distintos elementos del entorno, como UAVs, UGVs, estaciones de meteorología, sensores de movimiento, etc.

1.2. Motivación: Desarrollo Ágil

Al trabajar en el departamento de Simulación y Software de CATEC, lo primero que se aprende es a ser ágil. Por lo general, en la carrera de Ingeniería Técnica Informática de Sistemas no se estudian las metodologías ágiles, el *código limpio*[11], los Principios S.O.L.I.D.[15] (véase Sección 3.5), ni T.D.D. (véase Sección 3.2) que, aunque no son muy conocidas actualmente en la industria, son intuitivas de utilizar, e incluso necesarias para sacar el máximo partido al código desarrollado.

Las metodologías ágiles nos enseñan a planificarnos las tareas de una forma distinta, haciéndolo todo precisamente más ágil y dinámico, marcando metas reales a cada paso que se da en el desarrollo y dejando una puerta abierta siempre a posibles cambios de los requisitos.

Los Principios S.O.L.I.D. y los conceptos de *código limpio* nos hacen sencillo extraer el código escrito en cualquier lenguaje de programación, como cualquier libro o texto que nos podemos encontrar en cualquier idioma. Es sencillo comprender que para leer un libro en cierto idioma las palabras deben ordenarse según cierta lógica, y que su significado será siempre el mismo, tanto para una persona que habla ese idioma nativo como el que lo aprendió hace un año en un curso. Al llevar esta idea al código que se trabaja cada día, debemos tener en cuenta que si en algún momento ese código lo tiene que tocar otra persona o, si por algún casual, hace años que no lo tocamos, debe poder entenderse perfectamente al leerse de nuevo y que no genere dolores de cabeza el sólo tener que echarle un vistazo. Un código limpio y bien estructurado es como un buen libro, dan ganas de sumergirse en él.

1.3. Objetivos del proyecto

El objetivo principal de este proyecto es desarrollar el AMS mediante la utilización de Metodologías Ágiles[16] (véase Capítulo 2), concretamente Scrum[17]

(véase Sección 3.3) y Kanban[18] (véase Sección 3.4). Se desarrollará una Red de Petri (véase Sección 5.4), que gestionará la recepción y envío de señales, un Parseador de XML[19] (véase Sección 5.11), y un validador de expresiones (véase Capítulo: 8.3.1). Tras desarrollar todo esto en Java[20] (véase Sección 5.6), será visualizado en una Aplicación Web, utilizando Struts2[21] (véase Sección 5.16), Spring[23] (véase Sección 5.17), y JSP[46] (véase Sección 5.14).

1.3.1. Objetivos orientados a la metodología

Estos son los objetivos con respecto a las metodologías y técnicas usadas:

- Aplicar el Marco de Trabajo Scrum usando para ello los conocimientos adquiridos de libros como *Agile Software Development with Scrum*[17][26].
- Aplicar metodologías de eXtreme Programming de libros como *Scrum*[17] and *X.P. from the Trenches*[25].
- Comprobar los beneficios colaterales a la realización de estas prácticas como, por ejemplo, la facilidad de añadir nuevas tareas durante el proceso de desarrollo.
- Aplicar Kanban al desarrollo del proyecto.

1.3.2. Objetivos orientados a la técnica

Estos son los objetivos con respecto a las metodologías y técnicas usadas:

- Adquirir conocimientos de distintas tecnologías, como Struts2, Spring y

DDS RTI[8], para seguidamente ponerlos en práctica.

- Aplicar correctamente cada una de las metodologías estudiadas y sacar conclusiones de su uso.

1.3.3. Objetivos personales

Se ha querido asegurar que se cumplen los siguientes objetivos a lo largo del desarrollo de la aplicación:

- **Adaptación:** Adecuarse a las exigencias de estas nuevas metodologías. Aprender nuevas técnicas que mejoren la calidad del software.
- **Soltura:** Al centrarse en desarrollar software, adquirir cierta fluidez a la hora de escribir código y enfrentarse a nuevos lenguajes.
- **Compleitud:** No dejar nada suelto, sin probar, o a medias. El producto final debe ser estable y de calidad.
- **Experiencia:** Trabajar junto a un equipo de investigación en un entorno de trabajo nuevo y desarrollar algo para otras personas conforme a los requisitos, suman una experiencia valorable para el mercado profesional.

Se tendrán en cuenta estos objetivos durante la realización del proyecto y se comprobará si han sido realizados. Esto se verá con detenimiento en los apartados de conclusiones, véase la parte V del presente documento.

1.4. Estructura del documento

Este documento se estructura en las siguientes partes:

En la Parte I se encuentra el capítulo actual, en el que se ha introducido el ámbito del proyecto realizado junto con los objetivos que se pretenden conseguir con su realización.

La Parte II se centra en introducir al lector en las materias relacionadas con el proyecto. Se hará una pequeña introducción de las metodologías utilizadas para facilitar la compresión de éstas.

Con la Parte III se explican brevemente las tecnologías usadas en el proyecto, así como la integración del mismo con la plataforma desarrollada por las otras partes involucradas en el proyecto.

Una vez entrados en materia, la Parte IV muestra los resultados de las distintas fases del desarrollo de nuestro sistema de software. En este capítulo se quieren exponer los costes y la duración del proyecto.

Finalmente, en Parte VI, se hace una retrospectiva de final de proyecto, donde se comentarán las impresiones generales que han marcado el desarrollo de este proyecto fin de carrera, y se sacarán algunas conclusiones.

Como apéndices se añaden un Capítulo con varios ejemplos de uso, en el cual se contempla la funcionalidad de la Aplicación Web, una pequeña introducción a Hibernate[22] y las referencias utilizadas.

Parte II

Metodologías



Capítulo 2. Introducción a las Metodologías Ágiles

Tradicionalmente se tiende hacia el rápido desarrollo de aplicaciones en un mundo en el que el cambio y la evolución rápida y continua están a la orden del día. El principal objetivo es aumentar la productividad y satisfacer las variantes necesidades del cliente en el menor tiempo posible para proporcionar un mayor valor al negocio. Sin embargo, las metodologías convencionales no funcionan tan bien como deberían, por eso se han estado buscando distintas alternativas.

Entre los problemas del desarrollo convencional se pueden encontrar los siguientes: la captura de requisitos del proyecto es una fase previa a su desarrollo que, una vez completada, debe proporcionar una fotografía exacta de qué desea el cliente. Se trata de evitar a toda costa que se produzcan cambios en el conjunto de requisitos iniciales, puesto que a medida que avanza el proyecto resulta más costoso solucionar los errores detectados o introducir modificaciones. Además, en caso de que estos cambios de requisitos se produzcan, pretenden delegar toda responsabilidad económica en el cliente, derrachando un tiempo valioso para los programadores y, casi siempre, un desembolso mayor por parte del mismo. Por este motivo, se les conoce también como "metodologías predictivas". Sin embargo, el esfuerzo, tanto en coste como en tiempo, que supone hacer una captura detallada de todos los requisitos de un proyecto al comienzo de este es enorme, y rara vez se ve justificado con el resultado obtenido. Además, en muchas ocasiones, el cliente no conoce sus propias necesidades con la profundidad suficiente como para definirlas de forma exacta a priori y, a menudo, estas necesidades y sus prioridades varían durante la vida del proyecto (ver figura: 2.1).

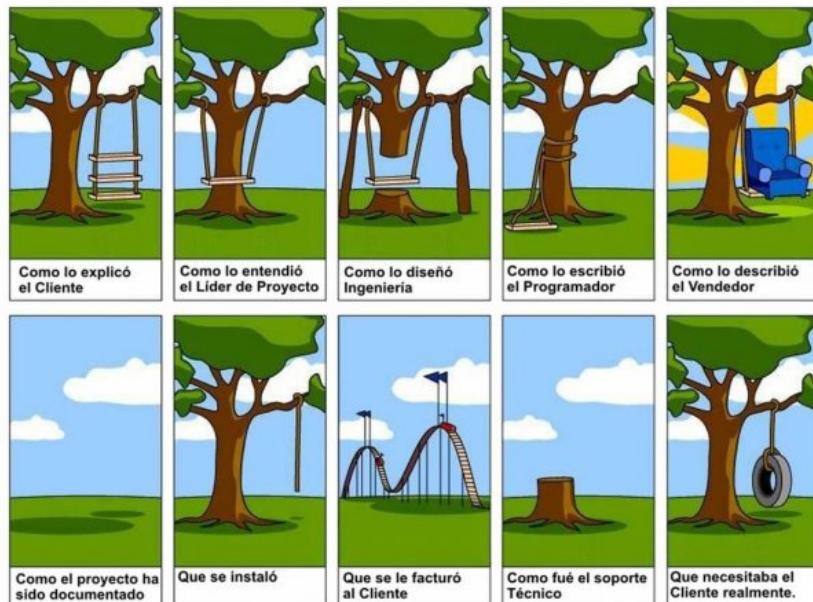


Figura 2.1: Cómo funcionan realmente los proyectos.

Establecer mecanismos de control es una de las opciones existentes para protegerse de estos cambios, aunque frecuentemente provocan la insatisfacción de los clientes, que perciben el desarrollo del proyecto como algo inflexible que no se adapta a sus necesidades y que, si lo hace, repercute negativamente en costes añadidos al presupuesto del proyecto. Por otro lado, en muchas ocasiones el proceso de desarrollo convencional está oprimido por excesiva documentación no siempre útil. Un porcentaje elevado del tiempo de desarrollo de un producto de software, desde el punto de vista de las Metodologías Ágiles^[16] (véase Capítulo 2), se malgasta en crear documentación que finalmente no se utiliza o queda obsoleta en un corto espacio de tiempo y que, por tanto, no aporta valor al negocio. Además, esta documentación innecesaria entorpece las labores de mantenimiento de la propia documentación útil, lo que provoca que en muchas ocasiones el mantenimiento de la documentación se vea agudizado.

Evidentemente, estas circunstancias no se adaptan a las restricciones de tiempo del mercado actual. Otra dificultad añadida al uso de metodologías convencionales es la lentitud del proceso de desarrollo. Es difícil para los desarrolladores entender un sistema complejo en su globalidad, lo que provoca que las diferentes etapas del ciclo de vida convencional transcurran lentamente. Dividir el

trabajo en módulos abordables ayuda a minimizar los fallos y, por tanto, el coste de desarrollo. Además, permite liberar funcionalidad progresivamente, según indiquen los estudios de las necesidades del mercado que aportan mayor beneficio a la organización. Podría decirse que es una aplicación práctica del algoritmo Divide y Vencerás.

Las Metodologías Ágiles^[16] luchan por hacer desaparecer todos estos problemas y hacer la vida del cliente y el programador mucho más sencilla, y con un canal de comunicación muy rico y poderoso. Estas son las principales ventajas de un Desarrollo ágil de software:

- Capacidad de respuesta a cambios a lo largo del desarrollo, ya que no se perciben como una piedra en el camino, sino como una oportunidad para mejorar el sistema e incrementar la satisfacción del cliente, considerando la gestión de cambios como un aspecto natural del propio proceso de desarrollo software.
- Entrega continua y en plazos breves de software funcional, lo que permite al cliente verificar *in situ* el desarrollo del proyecto, ir disfrutando de la funcionalidad del producto progresivamente y comprobando si satisface sus necesidades, lo cual repercute en una mayor satisfacción. Además, el desarrollo en ciclos de corta duración favorece que los riesgos y dificultades se repartan a lo largo del desarrollo del producto, principalmente al comienzo del mismo, y permite ir aprendiendo de estos riesgos y dificultades (ver figura 2.2).

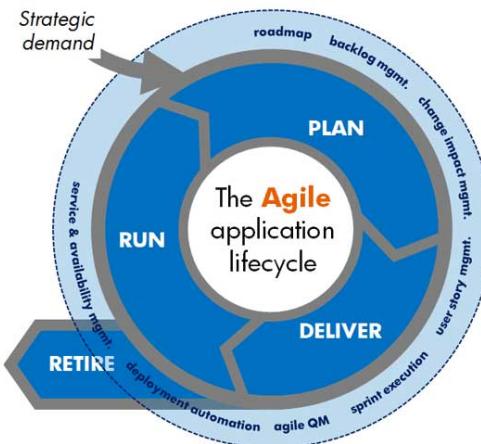


Figura 2.2: Ciclo de vida del Desarrollo ágil de software.

- Trabajo conjunto entre el cliente y el equipo de desarrollo con una comunicación directa para minimizar en gran medida los malentendidos, que constituyen una de las principales fuentes de errores en productos de software y de exceso de documentación improductiva.
- Importancia de la simplicidad, por lo que se prescinde del trabajo innecesario que no aporta valor al negocio.
- Atención continua a la excelencia técnica y al buen diseño para mantener una alta calidad de los productos.
- Mejora continua de los procesos y del equipo de desarrollo, entendiendo que el éxito depende de tres factores: éxito técnico, éxito personal y éxito de la organización.

2.1. Programación Extrema

El eXtreme Programming (X.P.) es una metodología de desarrollo de la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, *Extreme Programming Explained: Embrace Change*[27]. Es la más destacada de las metodologías del Desarrollo ágil de software.

X.P. se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad. Los defensores de la X.P. consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

2.1.1. Valores

Se puede considerar el eXtreme Programming como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto, y aplicarlo de manera dinámica durante el ciclo de vida del software. A continuación se expondrán los cinco valores en los que se funda X.P.:

- **Simplicidad:** es la base del X.P.. Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento. Un diseño complejo del código junto a sucesivas modificaciones por parte de diferentes desarrolladores hacen que la complejidad aumente exponencialmente. Para mantener la simplicidad es necesaria la Refactorización del código a medida que va creciendo. Ésta idea también debe ser aplicada en la documentación, de esta manera el código debe comentarse en su justa medida, intentando, eso sí, que el código esté autodocumentado.

- **Comunicación:** se realiza de diferentes formas. Para los programadores el código comunica mejor cuanto más simple sea. Si el código es complejo hay que esforzarse para hacerlo inteligible. El código autodocumentado es más fiable que los comentarios ya que éstos últimos pronto quedan desfasados a medida que el código es modificado. Debe comentarse sólo aquello que no va a variar, por ejemplo el objetivo de una clase o la funcionalidad de un método.

Las Pruebas Unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de como utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la Programación por parejas (véase Sección 3.1). La comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo. El cliente decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.

- **Retroalimentación:** al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real. Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en lo que es más importante.
- **Valentía:** programar para hoy y no para mañana. Esto es un esfuerzo para evitar empantanarse en el diseño y requerir demasiado tiempo y trabajo para implementar el resto del proyecto. La valentía le permite a los desarrolladores que se sientan cómodos para reconstruir su código cuando sea necesario. Esto significa revisar el sistema existente y modificarlo si con ello los cambios futuros se implementaran mas fácilmente. Otro ejemplo de valentía es saber cuando desechar un código: valentía para quitar código fuente obsoleto, sin importar cuanto esfuerzo y tiempo se invirtió en crear ese código. Además, valentía significa persistencia: un programador puede permanecer sin avanzar en un problema complejo por un día entero, y luego lo resolverá rápidamente al día siguiente, sólo si es persistente.
- **Respeto:** se manifiesta de varias formas. Los miembros del equipo se respetan los unos a otros, porque los programadores no pueden realizar cambios que hacen que las pruebas existentes fallen o que demore el trabajo de

sus compañeros. Los miembros respetan su trabajo porque siempre están luchando por la alta calidad en el producto y buscando el diseño óptimo o más eficiente para la solución a través de la Refactorización del código. Los miembros del equipo respetan el trabajo del resto, no menospreciándolo nunca, procurando mejorar autoestima en el equipo y elevando el ritmo de producción.

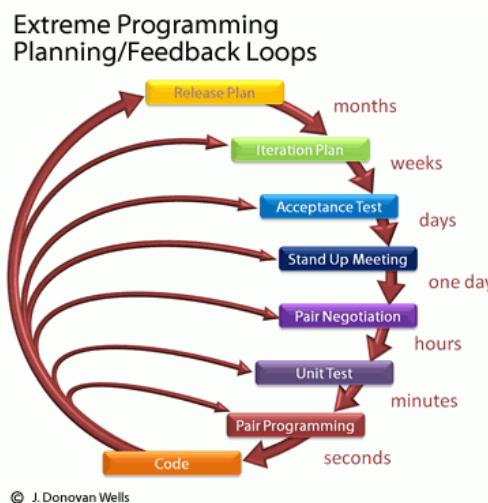


Figura 2.3: Ciclo de vida de un proyecto usando eXtreme Programming.

2.2. El Manifiesto Ágil



Figura 2.4: El Manifiesto Ágil.

El 17 de febrero de 2001, diecisiete críticos de los modelos de mejora del desarrollo de software basados en procesos, convocados por Kent Beck, se reunieron en Snowbird, Utah, para tratar sobre técnicas y procesos de desarrollo software. En la reunión se acuñó el término Metodologías Ágiles para definir a los métodos que estaban surgiendo como alternativa a las metodologías formales (C.M.M.I.^[28], SPICE^[29]) que consideraban excesivamente “pesadas” y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo. Los integrantes de la reunión resumieron los principios sobre los que se basan los métodos alternativos en cuatro postulados, lo que ha quedado denominado como *Manifiesto Ágil*^[16].

El Desarrollo ágil de software no especifica unos procesos o métodos que seguir, aunque bien es cierto que han aparecido algunas prácticas asociadas a este movimiento, el Desarrollo ágil de software es más bien una filosofía. En este documento se resume la filosofía Ágil y establece cuatro valores y doce principios.

Los 4 Valores del Manifiesto Ágil:

- 1.- Valorar más a los individuos y su interacción que a los procesos y las herramientas.** La gente es el principal factor de éxito de un proceso de software. Este primer valor expresa que es preferible utilizar un proceso indocumentado con buenas interacciones personales, que un proceso documentado con interacciones hostiles. Se considera que no se debe pretender la construcción del entorno en primer lugar y esperar que el equipo se adapte automáticamente, sino lo contrario: construir primero el equipo y que este configure su propio entorno. El talento, la habilidad, la capacidad de comunicación y de tratar con personas son características fundamentales para los miembros de un equipo ágil.
- 2.- Valorar más el software que funciona que la documentación exhaustiva.** Este valor es utilizado por muchos detractores de las Metodologías Ágiles, que argumentan que esta es la excusa perfecta para aquellos que pretenden evitar las tareas menos gratificantes del desarrollo de software, como las tareas de documentación. Sin embargo, el propósito de este valor es acentuar la supremacía del producto por encima de la documentación. El objetivo de todo desarrollador es obtener un producto que funcione y cumpla las necesidades del cliente y la documentación es un artefacto más que

utiliza para cumplir su objetivo. Por tanto, no se trata de no documentar, sino de documentar aquello que sea necesario para tomar de forma inmediata una decisión importante. Los documentos deben ser cortos y centrarse en lo fundamental. Dado que el código es el valor principal que se obtiene del desarrollo, se enfatiza el hecho de seguir ciertos estándares de programación para mantener el código legible y documentado.

- 3.- **Valorar más la colaboración con el cliente que la negociación contractual.** Se propone una interacción continua entre el cliente y el equipo de desarrollo, de tal forma que el cliente forme un tandem con el equipo. Se pretende no diferenciar entre las figuras cliente y equipo de desarrollo, sino que se apuesta por un solo equipo persiguiendo un objetivo común.

- 4.- **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.

Los 12 Principios del Manifiesto Ágil:

- 1.- La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporten valor.

- 2.- Dar la bienvenida a los cambios de requisitos. Se capturan los cambios para que el cliente tenga una ventaja competitiva.

- 3.- Liberar software que funcione frecuentemente, desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.

- 4.- Los miembros del negocio y los desarrolladores deben trabajar juntos dia-

riamente a lo largo del proyecto.

- 5.-** Construir el proyecto en torno a individuos motivados. Darles el entorno y apoyo que necesiten y confiar en ellos para conseguir finalizar el trabajo.
- 6.-** El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.
- 7.-** El software que funciona es la principal medida de progreso.
- 8.-** Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.
- 9.-** La atención continua a la calidad técnica y al buen diseño mejora la agilidad.
- 10.-** La simplicidad es esencial.
- 11.-** Las mejores arquitecturas, requisitos y diseños surgen de los equipos que se organizan ellos mismos.
- 12.-** En intervalos regulares, el equipo debe reflexionar sobre cómo ser más efectivo y, según estas reflexiones, ajustar su comportamiento.

Estos principios marcan el ciclo de vida de un Desarrollo ágil de software, así como las prácticas y procesos a utilizar.

Capítulo 3. Herramientas Ágiles

Aunque el movimiento ágil está sustentado por valores y principios para el desarrollo de productos de software, la mayoría de estas metodologías tienen asociadas un conjunto de prácticas, en muchos casos comunes, que buscan la agilidad en el desarrollo. En este capítulo se analizarán algunas de las más relevantes, derivadas directamente de X.P..

3.1. Programación por parejas

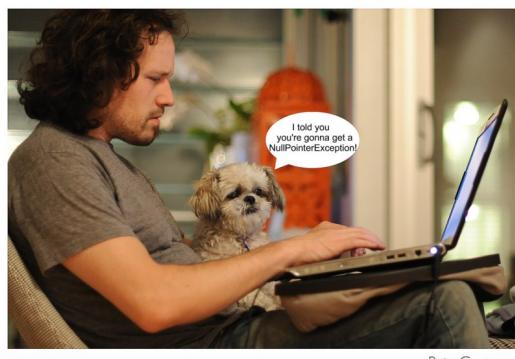
La Programación por parejas (Pair Programming) es una técnica de X.P. que consiste en lo siguiente: dos desarrolladores trabajan juntos en el mismo equipo físico, de tal manera que, mientras uno escribe código, el otro supervisa, y así sucesivamente, cambiando los roles cada cierto tiempo. Esto permite ir desarrollando en conjunto. Ambos aportan ideas de diseño, ambos deciden la solución óptima y el camino a seguir. Beneficia tener cuatro ojos sobre el código, puesto que el desarrollador que no escribe está pendiente en todo momento del cómo se está haciendo y le es más fácil mantener una perspectiva global de qué se quiere hacer.

Es especialmente útil que en las parejas haya miembros con perfiles distintos para que aprendan uno del otro. Por ejemplo, parejas expertas en distintos campos o parejas compuestas por un desarrollador senior y otro junior, para que este último obtenga el conocimiento del senior y aprenda de él sobre la marcha.

También, cuando llega alguien nuevo al equipo, es mucho más sencillo el traspaso de conocimientos de esta manera. Así, el nuevo miembro empieza a producir cuanto antes.

Con esta técnica, las personas tienden a estar mucho más atentas a lo que están haciendo y se reduce el tiempo improductivo y los despistes, ya que adquieren un compromiso el uno con el otro. Aparte, mejora el trabajo en grupo y une al equipo, aunque siempre pueden surgir pequeños roces por desacuerdos durante el desarrollo.

Este método es especialmente útil cuando hay que enfrentarse a un problema complejo o cuando hay que hacer transferencia de conocimientos que, de este modo, se hace de una manera muy natural, rápida y efectiva. Para codificaciones más sencillas, puede no ser el método más acertado si se trata con desarrolladores que están dentro del mismo nivel, por lo que sería preferible en este caso el desarrollo individual.



PetaCross.com

Figura 3.1: Sátira del Pair Programming.

3.2. T.D.D.

Test-Driven Development (T.D.D.) es una práctica de programación que involucra otras dos prácticas basadas en eXtreme Programming: Escribir las pruebas primero y el proceso de Refactorización. Para escribir las pruebas generalmen-

te se utilizan las Unit Tests (Pruebas Unitarias). En primer lugar, se escribe una prueba y se verifica que falla. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido (ver figura 3.2).

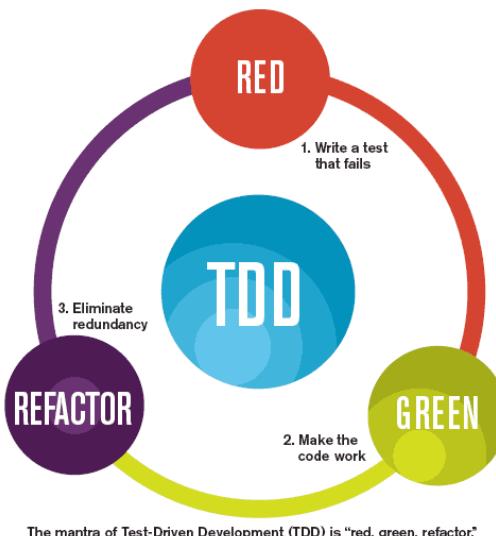


Figura 3.2: Algoritmo de T.D.D.

Para que funcione el desarrollo guiado por pruebas, el sistema que se programa tiene que ser lo suficientemente flexible como para permitir que sea probado automáticamente. Cada prueba será suficientemente pequeña como para que permita determinar únicamente si el código probado pasa o no la verificación que ésta le impone. El diseño se ve favorecido ya que se evita el indeseado "sobre diseño" de las aplicaciones y se logran interfaces más claras y un código más cohesionado. *Frameworks* como JUnit^[43] (véase Sección 5.7), proveen de un mecanismo para manejar y ejecutar conjuntos de pruebas automatizadas.

3.3. Scrum

Se define como un Marco de Trabajo para la gestión y desarrollo de software. Está basado en un proceso iterativo e incremental utilizado comúnmente en entornos con Desarrollo ágil de software. Está indicado para pequeños grupos de trabajo de hasta ocho programadores.



Figura 3.3: Logo de Scrum.

3.3.1. Características Principales

Scrum[17] basa sus principios en no imponer una forma fija de hacer las cosas. Es un Marco de Trabajo y propone herramientas que se pueden o no aplicar, en función del sentido que tengan en un determinado grupo de trabajo. Lo más importante para Scrum es el grupo de trabajo, ya que unos desarrolladores contentos e implicados contagian al resto de las partes, incluido el cliente.

A pesar de su flexibilidad, tiene como característica principal que toda actividad de Scrum está acotada en el tiempo. Durante cada *Sprint*, un periodo entre una y cuatro semanas (la magnitud es definida por el equipo), el equipo crea un incremento de software potencialmente entregable (utilizable).

3.3.2. Elementos de Scrum

Scrum usa un conjunto de elementos que sirven para dar forma al Marco de Trabajo. En esta sección se pasa a describir cada uno de ellos.

Desarrolladores: son los que realizan el trabajo duro, los que hacen viable el nuevo sistema.

Scrum Master: que mantiene los procesos y se postula como líder servil o como un guía para los demás. No se puede llamar jefe o director de proyecto, ya que el equipo en sí es su propio jefe.

Dueño del producto: representa, principalmente, a los clientes y usuarios; es el único punto de acceso entre éstos y el equipo de desarrollo, principalmente, a través del Scrum Master.

Stakeholder: son los clientes, usuarios e inversores involucrados en el proyecto. Solo participan en las revisiones de sprint (las demos).

Pila de productos o *Backlog*

Es una lista de requisitos, historias de cliente, funcionalidades, etc. Suelen estar definidos en algún documento o programa de gestión en donde se llevará el estado de cada uno de los componentes que forman la Pila de productos.

Historias de usuario

Es un resumen de acciones que debe realizar el programa. Generalmente una historia tiene que tener los siguientes datos:

- **ID:** número identificador de la historia. Sirve también para priorizar entre historias.
- **Nombre:** debe ser lo mas descriptivo posible.
- **Importancia:** un número para indicar la prioridad de la historia.
- **Estimación inicial:** estimación de la dificultad en la realización de una historia.
- **Test de aceptación:** resuelve y prueba si la historia está correctamente implementada.

Una vez que se tengan las Pilas de productos con las Historias de Usuario iniciales, se planifica el primer *Sprint*, que será siempre el primer paso dentro de cada iteración.

Diagrama de BurnDown

Es una gráfica de progreso, cada día se actualizan los puntos restantes por implementar. Esta gráfica muestra el estado actual del *Sprint*, en caso de ser muy horizontal la línea de progreso, se debería plantear para el siguiente *Sprint* reducir las historias que se quieren abarcar.

Se puede apreciar en la imagen 3.4) cuándo es necesario añadir más Historias de Usuario al *Sprint* y cuándo es preciso quitar alguna al no poderse realizar todas en el tiempo propuesto.

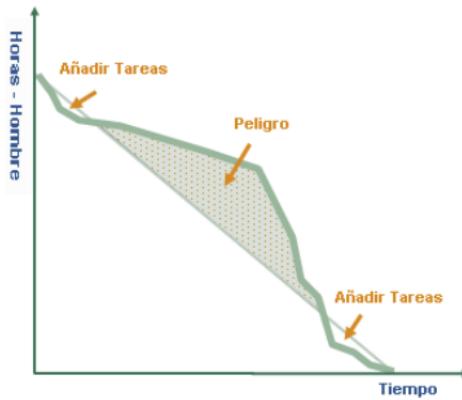


Figura 3.4: Cómo entender un Diagrama de BurnDown

La siguiente imagen muestra el diagrama de *BurnDown* del proyecto desarrollado. Se incluye una gráfica de velocidad, en la cual se indican los puntos de historia implementados por *Sprint*. En la siguiente gráfica, se indican los puntos que faltan por implementarse, los puntos añadidos en los *Sprint* correspondientes. Como se puede observar, la linea se asemeja a la ideal y por lo tanto se puede indicar que la planificación ha sido exitosa.

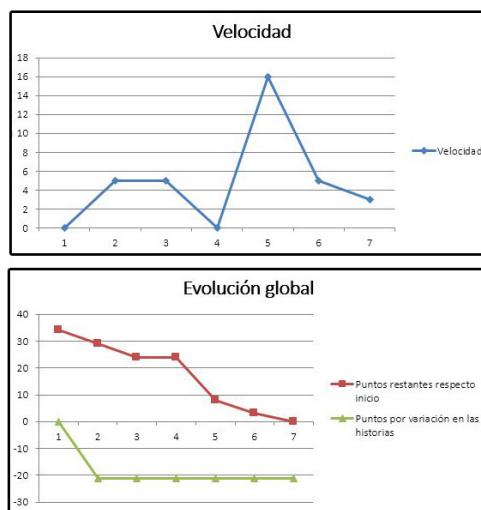


Figura 3.5: Diagrama de BurnDown.

3.3.3. Reuniones en Scrum

En cada *Sprint*, se determina un conjunto de reuniones, siempre acotadas en el tiempo, que cumplen con el objetivo de una mejora continua, tanto del producto como de las relaciones interpersonales de los participantes. Son importantes para tener el control de qué y cómo se están haciendo las cosas. Además, previenen desviaciones, ya que su propósito es detectar los problemas a tiempo.

Planificación del *Sprint*

Se realiza una reunión, que es crítica: la planificación del *Sprint*. Su función es dar trabajo sin interrupciones durante unas semanas (entre dos y cuatro normalmente).

Aquí se determina una meta de *Sprint*, una lista de miembros, una pila de *Sprint*, una fecha para la demo, y un lugar y una hora para el Scrum diario. Es muy importante que asista el Dueño del producto puesto que es él quien determina el alcance e importancia de la tarea. En caso de que no quiera o pueda asistir, el dueño asignará a un componente del grupo la función de mediador y responsable de tomar las decisiones.

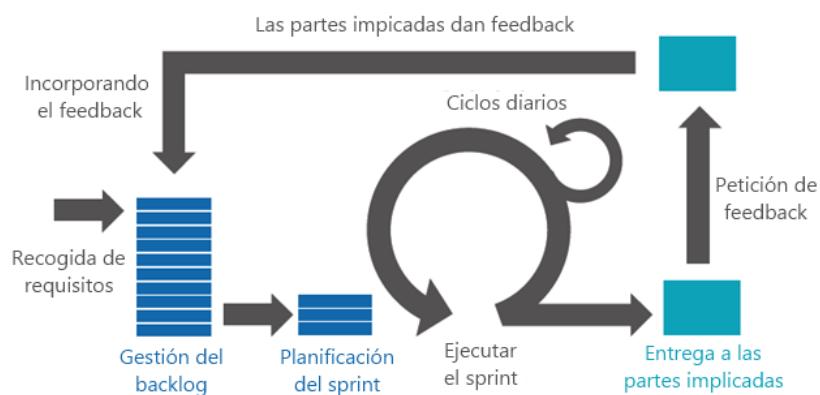


Figura 3.6: Proceso de iteración en Scrum.

El conjunto de características que forma parte de cada *Sprint* viene del *Product Backlog*, que es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar. Los elementos del *Product Backlog* que forman parte del *Sprint* se determinan durante la reunión de *Sprint Planning*. Durante esta reunión, el *Product Owner* identifica los elementos del *Product Backlog* que quiere ver completados y los hace del conocimiento del equipo. Entonces, el equipo determina la cantidad de ese trabajo que puede comprometerse a completar durante el siguiente *Sprint*. Durante el *Sprint*, nadie puede cambiar el *Sprint Backlog*, lo que significa que los requisitos están congelados durante el *Sprint*, aunque sí que se pueden cambiar el *Product Backlog*, por si es necesario sortear posibles obstáculos surgidos en el desarrollo (ver figura 3.6).

En cada *Sprint*, se determina un conjunto de reuniones, siempre acotadas en el tiempo, que cumplen con el objetivo de una mejora continua, tanto del producto como de las relaciones interpersonales de los participantes. Son importantes para tener el control de qué y cómo se están haciendo las cosas. Además, previenen desviaciones, ya que su propósito es detectar los problemas a tiempo.

Este Marco de Trabajo permite la creación de equipos autoorganizados impulsando la co-localización de todos los miembros del equipo, la comunicación verbal entre ellos y las disciplinas involucradas en el proyecto. Un principio clave de *Scrum* es hacer ver a los clientes que pueden cambiar de idea sobre lo que quieren y necesitan, y que los desafíos impredecibles no pueden ser fácilmente enfrentados de una forma predictiva y planificada. Por lo tanto, *Scrum* adopta una aproximación pragmática, aceptando que el problema no puede ser completamente entendido o definido, y centrándose en maximizar la capacidad del equipo de entregar rápidamente y responder a requisitos emergentes. Existen varias implementaciones de sistemas para gestionar el proceso de *Scrum* que van desde notas *Post-it* y pizarras hasta paquetes de software. Una de las mayores ventajas de *Scrum* es que es muy fácil de aprender, y requiere muy poco esfuerzo de adaptación una vez que se empieza a utilizar.

3.4. Kanban

Kanban[18] es un método para gestionar la carga de trabajo. Hace especial hincapié en la entrega justo a tiempo sin sobrecargar de trabajo a los miembros del equipo.

Las principales reglas de Kanban son las tres siguientes:

- 1.- **Visualizar el trabajo en Kanban y las fases del ciclo de producción, o flujo de trabajo.** Al igual que Scrum, Kanban se basan en el desarrollo incremental, dividiendo el trabajo en partes. Una de las principales aportaciones es que utiliza técnicas visuales para ver la evolución de cada tarea. La práctica más común es utilizar post-it en una pizarra.

Para empezar, el trabajo se divide en partes, normalmente cada una de esas partes se escribe en un post-it y se pega en una pizarra. Los Post-it suelen tener información variada. A parte de la descripción, deben tener la estimación de la duración de la tarea. La pizarra tiene tantas columnas como estados por los que puede pasar la tarea (en espera de ser desarrollada, en análisis, en diseño, etc). El objetivo de esta visualización es que quede claro el trabajo a realizar, con qué está cada persona, que todo el mundo tenga algo que hacer y el tener clara la prioridad de las tareas. Las fases del ciclo de producción o flujo de trabajo se deben decidir según el caso, no hay nada acotado (ver 3.7).

- 2.- **Determinar el límite de Trabajo en curso.** Quizás una de las principales ideas del Kanban es que el *Trabajo en curso* debería estar limitado, es decir, que el número máximo de tareas que se pueden realizar en cada fase debe ser algo conocido. En Kanban se debe definir cuantas tareas como máximo pueden realizarse en cada fase del ciclo de trabajo (como máximo 4 tareas en desarrollo, como máximo 1 en pruebas, etc). A este número de tareas se le llama límite del *Trabajo en curso*. De esta idea, surje otra igualmente razonable, que impide empezar una tarea nueva antes de haber finalizado alguna de las que actualmente están en curso. En la figura de ejemplo 3.7, el número límite del *Trabajo en curso* se ha colocado entre paréntesis debajo del nombre de cada tarea.
- 3.- **Medir el tiempo en completar una tarea.** El tiempo que se tarda en

terminar cada tarea se debe medir, a ese tiempo se le llama *Lead Time*. Este tiempo cuenta desde que se hace una petición hasta que se hace la entrega. Aunque la métrica más conocida del Kanban es el *Lead Time*, normalmente se suele utilizar también otra métrica importante: el *Cycle Time*. Éste mide desde que el trabajo sobre una tarea comienza hasta que termina. Puede haber más métricas, pero las anteriores son las realmente importantes, y necesarias para el control y mejora continua.



Figura 3.7: Ejemplo de pizarra en Kanban.

3.5. Principios S.O.L.I.D.

Estas 5 letras representan cinco principios básicos de la P.O.O.. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo.

3.5.1. Descripción de los Principios

Los Principios S.O.L.I.D.[\[15\]](#) son guías que pueden ser aplicadas en el Desarrollo ágil de software para eliminar código sucio provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible.

- **Principio de una sola responsabilidad:** Cada clase debería tener un único motivo para ser modificada. Por ejemplo, una clase que se puede ver obligada a cambiar ante una modificación en la B.D. y que también puede variar ante un cambio en el proceso de negocio, se puede afirmar que tiene más de una responsabilidad o más de un motivo para cambiar. Este principio se aplica tanto a la clase como a cada uno de sus métodos, con lo que cada método también debería tener una única responsabilidad.
- **Principio de abierto/cerrado:** Una entidad de software (una clase, módulo o función) debe estar abierta a extensiones pero cerrada a modificaciones. Puesto que el software requiere cambios y que unas entidades dependan de otras, las modificaciones en el código de una de ellas puede generar indeseables efectos colaterales en cascada. Para evitarlo, el principio dice que el comportamiento de una entidad debe poder alterarse sin tener que modificar su propio código fuente.
- **Principio de sustitución de Liskov:** Introducido por Bárbara Liskov en 1987. Se basa en que los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Es decir, si un método recibe un objeto como parámetro de tipo X y, en su lugar, se le pasa otro de tipo Y, que hereda de X, dicho método debería funcionar correctamente.

Este principio está estrechamente relacionado con el anterior en cuanto a la extensibilidad de las clases cuando ésta se realiza mediante herencia o subtipos.

- **Principio de la segregación de la interfaz:** Defiende que no se obligue

a los clientes a depender de clases o interfaces que no necesitan usar. Tal imposición ocurre cuando una clase o interfaz tiene más métodos de los que un cliente (otra clase o entidad) necesita para sí mismo. Seguramente sirve a varios objetos cliente con responsabilidades diferentes, con lo que debería estar dividida en varias entidades. Por lo tanto, siempre se implementan todos los métodos de la interfaz y de las clases de las que se hereda.

- **Principio de Inversión de Dependencias:** Este principio dice que un módulo concreto A no debe depender directamente de otro módulo concreto B, sino de una abstracción de B. Tal abstracción es una interfaz o una **Entidad Abstracta**, que sirve de base para un conjunto de clases hijas. Este principio da lugar a un **Patrón de Diseño Inyección de Dependencias** (véase: 5.5.8) que se estudiará más adelante. Este patrón es una de las mejores técnicas para lidiar con las colaboraciones entre clases y que produce un código reutilizable, sobrio y preparado para cambiar de manera limpia.

3.6. Integración Continua

Prácticas como T.D.D. son ideales para usar junto con la I.C., dado que se centran en disponer de una buena batería de pruebas y en realizar pequeños cambios que se suben al control de versiones. Aunque en realidad, la metodología de desarrollo no es determinante, siempre y cuando se cumplan una serie de buenas prácticas. Los beneficios que aporta proporcionan gran valor a las **Metodologías Ágiles**. En algunos proyectos, la integración se lleva a cabo como un evento, por ejemplo, un día a la semana, se integra todo. La I.C. elimina esta forma de ver la integración, ya que forma parte de nuestro trabajo diario. El resultado es que siempre se tendrá un entregable preparado para entregar al cliente.

3.6.1.I.C. aplicado al proyecto

Al ser este un proyecto de investigación, sin la figura del cliente ni entorno de producción, en el que no es necesario disponer de un entregable disponible en cada momento, y teniendo en cuenta que el proyecto ha sido desarrollado por una sola persona, con ayuda en algunos casos de una segunda persona, pero siempre en el mismo PC y con los dos desarrolladores presentes, se ha determinado que no era necesario aplicar esta práctica.

Capítulo 4. Desarrollo Ágil en el AMS

Con Scrum los *Sprints* se planifican conforme a las funcionalidades necesarias a medida que avanza el proyecto, haciendo que el desarrollo sea mucho más flexible y cómodo para el desarrollador, que se planifica sus tareas conforme necesita, sin tener demasiadas restricciones temporales.

Una de las características del Marco de Trabajo Scrum es la existencia de determinados roles; el proyecto cuenta con todos ellos. Irene Alejo, ,al tener más experiencia como desarrolladora, adopta el rol de Scrum Master. El papel de Dueño del producto es para Paco Alarcón, supervisor del proyecto PLANET en CATEC.

Es necesario recalcar que su labor ha sido muy realista, puesto que, a medida que se iban realizando las tareas, cambiaba el alcance y la prioridad de estas e, incluso, se añadió más funcionalidad de la que se iba a realizar en un principio, lo cual da más valor al trabajo aplicando Scrum[17] .

Es la Scrum Master la que se encarga de la reunión de pila de productos, en la que se recogen los requisitos iniciales, así como de organizar la duración de los *Sprints* y las Historias de Usuario iniciales.

Las Historias de Usuario quedan divididas a su vez en pequeñas tareas, para su mejor desarrollo. Para la pila de productos de Scrum y Kanban se utiliza una pizarra física donde poder administrar y visualizar de forma sencilla las Historias de Usuario y el avance día a día del desarrollo.

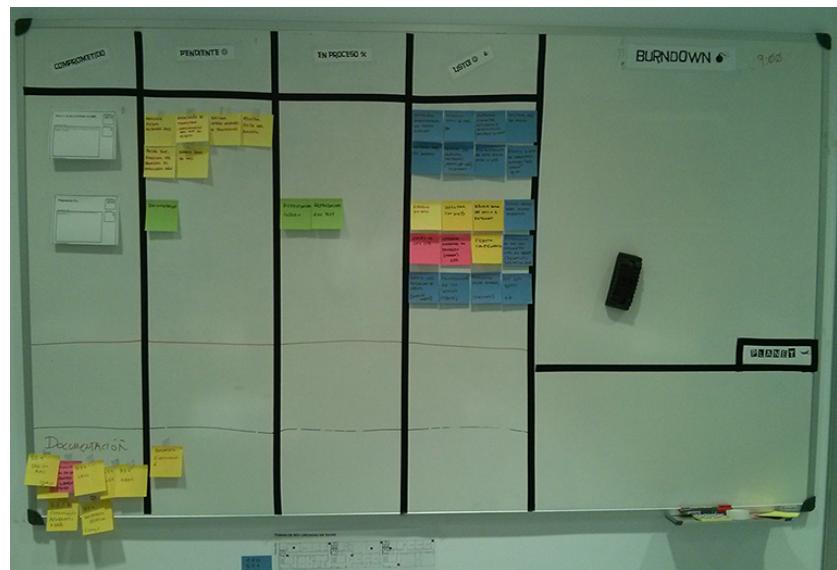


Figura 4.1: Pizarra de Scrum.

Por otro lado, las reuniones de Scrum diarios formaron parte del día a día habitualmente siempre a la misma hora, con una duración de unos 5 minutos donde se explicaba qué se había avanzado y cual era el siguiente paso a tomar en ese momento. Los DeadLines se han cumplido en su mayoría y algunas de las tareas que iban surgiendo sobre la marcha, se acometieron perfectamente sin demasiado impacto en el *Sprint* final.

4.1. Sentando las bases

Como se ha ido explicando en capítulos anteriores, con el uso de las Metodologías Ágiles viene implícito el uso de ciertas metodologías en el ámbito de la programación.

Para empezar con buen pie, se realizó un repaso a los Principios S.O.L.I.D., T.D.D. y Scrum[17].

Durante el desarrollo del proyecto, se ha utilizado T.D.D. en cada paso. Para ellos, primero se realiza el test donde se especifica el comportamiento y después se implementa el código. Finalmente, al terminar de realizar el código, este se refactoriza para que el programa resultante sea completamente robusto.

En un principio, esta parte de la programación y el repaso del código se hacía preferentemente entre dos, haciendo *Pair Programming*, ya que el conocimiento del desarrollador con más experiencia en esta materia hacía más sencilla la curva de aprendizaje de estas metodologías. Pasado un tiempo, adaptado ya a T.D.D., y Scrum el siguiente paso fue desarrollar la aplicación sin más ayuda, aplicando los conocimientos aprendidos. Sin embargo el *Pair Programming* siguió estando presente en algunos momentos en los el programador *Junior* ha necesitado ayuda de los compañeros de más experiencia para solucionar fallos en el código y aconsejando a la hora de tomar ciertas decisiones de optimización en el mismo y en el proyecto en general.

Parte III

Estudio de las Tecnologías

Capítulo 5. Herramientas para el desarrollo

En este capítulo se estudiará cada uno de los lenguajes y librerías usadas para el desarrollo del proyecto, introduciendo los conocimientos necesarios para su entendimiento. Además se detallaran aquellas definiciones importantes y necesarias para entender los contenidos de este proyecto.

5.1. Programación Orientada a Objetos

La P.O.O. es un paradigma de programación que usa los objetos en sus interacciones para diseñar aplicaciones y programas informáticos.

Un *Objeto* es una entidad que tiene un determinado estado, comportamiento e identidad:

- **Estado:** está compuesto de datos o informaciones; serán uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
- **Comportamiento:** está definido por los métodos o mensajes a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.

- **Identidad:** propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un *Objeto* contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece de la misma manera el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

Los métodos y atributos están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro.

5.1.1. Características de la P.O.O.

Existe un acuerdo acerca de qué características contempla la P.O.O., estas son las más importantes:

- **Abstracción:** denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un *Agente* abstracto que puede realizar trabajo, informar y cambiar su estado, así como comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos

de entidades en el mundo real.

- **Encapsulamiento:** significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema.
- **Modularidad:** es la propiedad que permite subdividir una aplicación en partes más pequeñas (también llamados módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- **Principio de ocultación:** se basa en aislar cada objeto del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas.
- **Polimorfismo:** comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. Dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en tiempo de ejecución, esta última característica se llama asignación tardía o asignación dinámica.
- **Herencia:** las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el *encapsulamiento* y el *encapsulamiento*, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Éstos pueden compartir y extender su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos

en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

- **Recolección de basura:** es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando.

5.2. POJO

La convención general define a un Plain Old Java Object (P.O.J.O.) como un objeto de una clase que no responde a ningún *Framework* determinado, es decir, un P.O.J.O. es un objeto de una clase que tiene atributos privados, métodos getters y setters públicos, y no necesita tener nada más.

Un P.O.J.O. es una estructura de datos, no tiene funcionalidad ni almacena datos. Por ejemplo, un *Servlet*, que es un objeto de una clase que extiende a *HttpServlet* lo que conlleva implementar los métodos de ese tipo), no sería un P.O.J.O., sin embargo, un objeto de una clase *Persona*, cuyos atributos fuesen privados, y sus métodos get y set fueran de acceso público, sí cumpliría las condiciones para ser un simple P.O.J.O.. Para terminar esta Sección, se verá un ejemplo sencillo de implementación:

```
public class Dog {  
  
    private String name;  
    private char sex;  
    private float weight;  
  
    public Dog() {  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
public void setName(String name) {
    this.name = name;
}

public char getSex() {
    return sex;
}

public void setSex(char sex) {
    this.sex = sex;
}

public float getWeight() {
    return weight;
}

public void setWeight(float weight) {
    this.weight = weight;
}
```

5.3. Acoplamiento

El acoplamiento se refiere al grado de conocimiento directo que tiene una clase de otra, hay de dos tipos, Fuerte y Débil:

- **Acoplamiento Fuerte:** significa que las clases relacionadas necesitan saber detalles internos unas de otras, los cambios se propagan por el sistema y el sistema es posiblemente más difícil de entender.
- **Acoplamiento Débil:** es aquel en el que cada uno de sus componentes tiene o necesita poco o ningún conocimiento de las definiciones de otros componentes por separado.

5.4. Red de Petri

La primera Red de Petri fue definida en la década de los años 1960 por matemático Carl Adam Petri[49], de ahí su nombre. Una Red de Petri, es una representación matemática o gráfica de un sistema a eventos discretos en el cual se puede describir la topología de un sistema distribuido, paralelo o concurrente. Son una generalización de la teoría de autómatas que permite expresar un sistema a eventos concurrentes.

En esencia, una Red de Petri, es un grafo orientado con dos tipos de nodos: lugares (representados mediante circunferencias), transiciones (representadas por segmentos rectos verticales), arcos dirigidos o flechas y marcas (representadas mediante un punto en el interior del círculo).

- **Lugar o Place:** Son los estados en los que se encuentra la Red de Petri, puede haber uno o varios activados.
- **Marcas:** Los lugares contienen un número finito o infinito contable de marcas. Las marcas se *consumen* en un lugar y se *producen* en otro cuando se dispara una transición entre estos os lugares.
- **Transición o Transition:** Se ubica entre un lugar y otro, se puede decir que es la puerta de entrada y salida de un lugar. Una marca no puede *pasar* de un lugar a otro hasta que la transición entre ambos no esté disparada, o activa.
- **Arcos:** Conectan un lugar a una transición, así como una transición a un lugar. No puede haber arcos entre lugares ni entre transiciones. Los arcos son unidireccionales, la marca no puede volver atrás.

Características:

- Una transición puede ser destino de varios lugares, y un lugar puede ser destino de varias transiciones.
- Una transición puede ser origen de varios lugares y un lugar puede ser origen de varias transiciones.
- Cada lugar tiene asociado una acción o salida.
- Los lugares que contienen marcas se consideran lugares *activos*.
- A las transiciones se les asocia eventos o funciones lógicas de las variables de entrada. en nuestro caso, se les asigna el resultado de la evaluación de una expresión regular.
- Una transición está sensibilizada cuando todos sus lugares origen están marcados. No confundir con disparada, ya que puede estar disparada o validada pero no sensibilizada, y viceversa. Cuando una transición está disparada y sensibilizada, se llama franqueo.
- Cuando una transición está disparada y sus lugares de origen marcados, desaparece una marca del lugar o lugares de origen y se crea una en el o los lugares destino.
- En su forma más básica, las marcas que circulan en una Red de Petri son todas idénticas. Se puede definir una variante de las redes de Petri en las cuales las marcas pueden tener un color (una información que las distingue), un tiempo de activación y una jerarquía en la red. Para nuestro caso, se usará la básica.

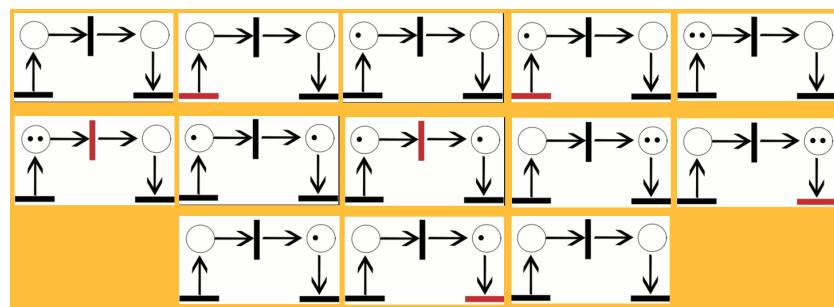


Figura 5.1: Ejemplo de funcionamiento de una red de Petri simple.

5.4.1. Propiedades y Validación

La validación consiste en comprobar que se cumplen las propiedades de Vivacidad, Limitación y Reversibilidad.

Vivacidad: Se trata de un concepto relacionado con la idea de *no bloqueo*. Es decir, una transición se dice viva si para un marcado inicial existe una secuencia para la cual se puede franquear. Si todas las transiciones son vivas, entonces la Red de Petri se llama viva y nunca se bloqueará. Si hay algunas vivas y otras no y la Red de Petri no se bloquea totalmente, se llamará pseudo-viva, en otro caso, la red queda bloqueada.

Limitación: La red estará k-limitada si para todo marcado alcanzable, se tiene que ningún lugar tiene un número de marcas mayor que k. Las redes con k=1 son conocidas como binarias. Una Red de Petri no podrá generar más marcas que las que su limitación le permite.

Reversibilidad: Una Red de Petri es reversible si para cualquier marcado alcanzable, es posible volver al marcado inicial.

5.4.2. Ventajas frente a Grafos de estados

A la hora de representar casos complejos, la Red de Petri se aproxima más al problema y trata mejor la ejecución de las tareas. Es decir, se hace un tratamiento individual de procesos independientes más óptimo, así como de procesos paralelos o concurrentes.

La Red de Petri permite modelar sistemas donde un recurso es compartido por dos procesos, de forma que el uso del recurso durante la ejecución de un proceso impide que dicho recurso sea utilizado por el otro proceso, de esta forma un recurso compartido se modelaría mediante un lugar con una marca inicial y transiciones en conflicto.

5.5. Patrones de Diseño

En los inicios de la informática, la programación se consideraba un arte y se desarrollaba como tal, debido a la dificultad que entrañaba para la mayoría de las personas, pero con el tiempo se han ido descubriendo y desarrollando formas y guías generales, con base a las cuales se puedan resolver los problemas. A estas guías se les ha denominado *Arquitectura de Software*, porque, a semejanza de los planos de un edificio, estas indican la estructura, funcionamiento e interacción entre las partes del software.

Los Patrones de Diseño son la base para la búsqueda de Soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Para que una Solución sea considerada un Patrón debe poseer ciertas características:

- Se debe de haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.
- Debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente, no debe imponer ciertas alternativas de diseño frente a otras.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.
- No debe eliminar la creatividad inherente al proceso de diseño.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el Patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable, abusar o forzar el uso de los patrones puede ser un error.

Para describir un Patrón se usan plantillas más o menos estandarizadas, de forma que se expresen de forma semejante y puedan constituir efectivamente un medio de comunicación uniforme entre diseñadores. Varios autores eminentes en esta área han propuesto plantillas ligeramente distintas, si bien la mayoría definen los mismos conceptos básicos. La plantilla más común es la utilizada precisamente por el GoF y consta de los siguientes apartados:

- Nombre del Patrón: nombre estándar del Patrón por el cual será reconocido en la comunidad (normalmente se expresan en inglés).
- Clasificación del Patrón: creacional, estructural o de comportamiento.

- Intención: ¿Qué problema pretende resolver el Patrón?
- También conocido como: Otros nombres de uso común para el Patrón.
- Motivación: Escenario de ejemplo para la aplicación del Patrón.
- Aplicabilidad: Usos comunes y criterios de aplicabilidad del Patrón.
- Estructura: Diagramas de clases oportunos para describir las clases que intervienen en el Patrón.
- Participantes: Enumeración y descripción de cada Entidad Abstracta que participan en el Patrón.
- Colaboraciones: Explicación de las interrelaciones que se dan entre los participantes.
- Consecuencias: Consecuencias positivas y negativas en el diseño derivadas de la aplicación del Patrón.
- Implementación: Técnicas o comentarios oportunos de cara a la implementación del Patrón.
- Código de ejemplo: Código fuente ejemplo de implementación del Patrón.
- Usos conocidos: Ejemplos de sistemas reales que usan el Patrón.
- Patrones relacionados: Referencias cruzadas con otros patrones.

A continuación se describen algunos de los patrones GoF usados en el proyecto.

5.5.1. Patrón Singleton

Singleton o *Instancia Única*, garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia, de esta forma, se restringe la creación de objetos de esta clase.

Se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

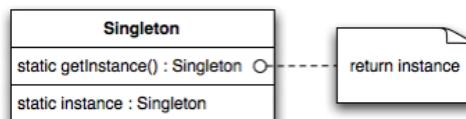


Figura 5.2: Diagrama de Clases del Singleton.

Las situaciones más habituales de aplicación de este Patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

Consecuencias de su uso:

- Acceso controlado a una única instancia.
- Reduce el espacio de nombres.

- Permite el refinamiento de operaciones y representación heredando de ella.
- Permite un número variable de instancias.

Por último, se muestra un ejemplo de implementación en Java:

```
public class Singleton {  
    private static Singleton INSTANCE = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

5.5.2. Patrón Composite

Composite u *Objeto compuesto*, permite tratar objetos compuestos como si de uno simple se tratase, gracias a la composición recursiva y a una estructura en forma de árbol, construye objetos complejos a partir de otros más simples y similares entre sí.

Por ejemplo, un objeto cuya clase es *GrupoDelmágenes* podría contener un *Cuadrado*, un *Triángulo* y otro *GrupoDelmágenes*, este grupo de imágenes podría contener un *Círculo* y un *Cuadrado*. Posteriormente, a este último grupo se le podría añadir otro *GrupoDelmágenes*, generando una estructura de composición recursiva en árbol, por medio de muy poca codificación y un diagrama sencillo y claro.

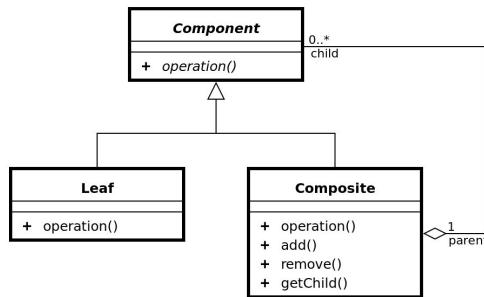


Figura 5.3: Diagrama de Clases del Composite.

5.5.3.Patrón Builder

Builder o *Constructor Virtual*, por definición, construye también el Patrón *Composite*. Su labor consiste en permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase *AbstractBuilder*.

La intención principal de este Patrón es, en resumidas cuentas, abstraer el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

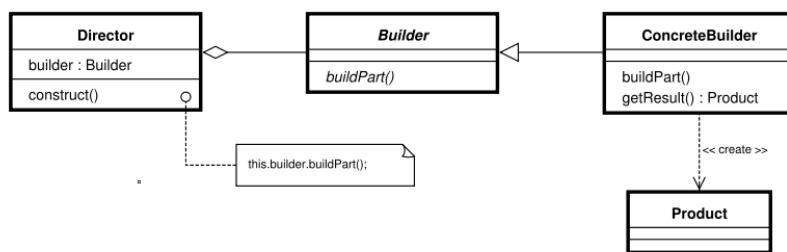


Figura 5.4: Diagrama de Clases del *Builder*.

Se muestra un ejemplo sencillo:

```
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() { return pizza; }
    public void crearNuevaPizza() {
        pizza = new Pizza();
        buildMasa();
        buildSalsa();
        buildRelleno();
    }

    public abstract void buildMasa();
    public abstract void buildSalsa();
    public abstract void buildRelleno();
}

/** "Director" */
class OtraCocina {
    private OtroPizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(OtroPizzaBuilder pb) { pizzaBuilder = pb; }
    public Pizza getPizza() { return pizzaBuilder.getPizza(); }

    public void construirPizza() {
        pizzaBuilder.crearNuevaPizza();
        //notar que no se necesita llamar a cada build.
    }
}
```

Ventajas de su uso:

- Reduce el acoplamiento.
- Permite variar la representación interna de estructuras compleja, respetando la interfaz común de la clase *Builder*.
- Se independiza el código de construcción de la representación. Las clases concretas que tratan las representaciones internas no forman parte de la interfaz del *Builder*.
- Cada *ConcreteBuilder* tiene el código específico para crear y modificar una estructura interna concreta.

- Distintos Director con distintas utilidades (visores, parsers, etc), pueden utilizar el mismo *ConcreteBuilder*.
- Permite un mayor control en el proceso de creación del objeto. El *Director* controla la creación paso a paso, solo cuando el *Builder* ha terminado de construir el objeto lo recupera el *Director*.

5.5.4. Patrón Facade

El Patrón *Facade* o *Fachada* se aplicará cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel.

Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable (esto es, reduciendo dependencias entre los subsistemas y los clientes).

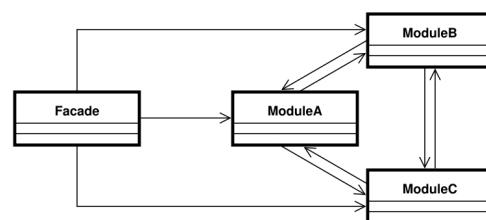


Figura 5.5: Diagrama de Clases del *Facade*.

Participantes:

Facade: conoce qué clases del subsistema son responsables de una determinada petición, y delega esas peticiones de los clientes a los objetos apropiados

del subsistema.

Subclases(ModuleA, ModuleB, ModuleC...): implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la *Fachada*. No conocen la existencia de la *Fachada*.

Ventajas e inconvenientes:

La principal ventaja del Patrón *Facade* consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.

Como inconveniente, si se considera el caso de que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global.

5.5.5. Patrón Command

Este Patrón permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además se facilita la parametrización de los métodos.

Ventajas de su uso:

- Se independiza la parte de la aplicación que invoca las órdenes de la implementación de los mismos.

- Al tratarse las órdenes como objetos, se puede realizar herencia de las mismas, composiciones de órdenes (mediante el Patrón *Composite*).
- Se facilita la ampliación del conjunto de órdenes.

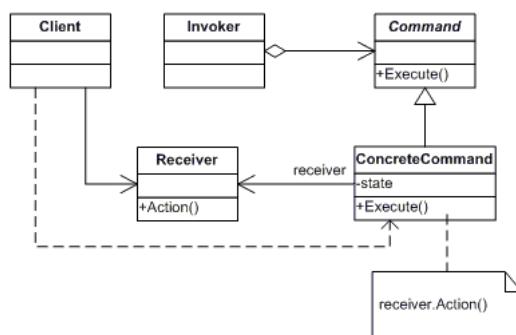


Figura 5.6: Diagrama de Clases del Command.

5.5.6. Patrón Observer

El Patrón *Observer* u *Observador*, define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. Este Patrón también se conoce como de publicación-inscripción.

Explicado más detalladamente, se dispondrá de el objeto de datos o *Sujeto*, el cual contiene atributos mediante los cuales cualquier objeto Observador se puede suscribir a él pasándole una referencia a sí mismo. El *Sujeto* mantiene así una lista de las referencias a sus observadores. Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el *Sujeto* es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el *Sujeto*, ejecutado, por ejemplo, por alguno de los observadores, el objeto de datos puede recorrer la lista de *observadores* avisando a cada uno.

En resumidas cuentas, se usará el Patrón *Observador* cuando un elemento necesita saber del estado actual de otro, sin tener que estar preguntando de forma permanente si éste ha cambiado o no.

Participantes:

- **Sujeto (Subject):** El sujeto concreto proporciona una interfaz para agregar (*attach*) y eliminar (*detach*) observadores. El Sujeto conoce a todos sus observadores.
- **Observador (Observer):** Define el método que usa el sujeto para notificar cambios en su estado (*update/notify*).
- **Sujeto Concreto (ConcreteSubject):** Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.
- **Observador Concreto (ConcreteObserver):** Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

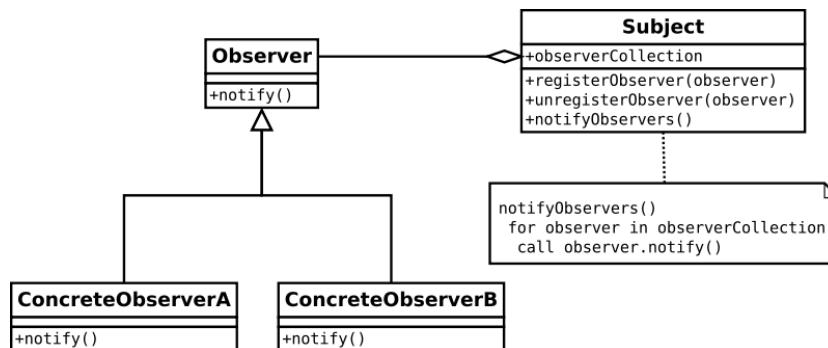


Figura 5.7: Diagrama de Clases del Observer.

Consecuencias de su uso

Por una parte abstrae el acoplamiento entre el sujeto y el observador, lo cual es beneficioso ya que se consigue una mayor independencia y además el sujeto no necesita especificar los observadores afectados por un cambio.

Por otro lado, con el uso de este Patrón se desconocerán las consecuencias de una actualización, lo cual, dependiendo del problema, puede afectarnos en mayor o menor medida.

5.5.7. Patrón Interceptor

El Patrón de Diseño *Interceptor* se utiliza cuando los sistemas de software o *Frameworks* necesitan ofrecer una manera de cambiar o aumentar su ciclo normal de procesamiento. Por ejemplo, una secuencia de tratamiento típica para un Servidor Web es recibir una *URI* del navegador, asignarla a un archivo en el disco, abrir el archivo y enviar su contenido al navegador. Cualquiera de estos pasos podría ser sustituido o cambiado, es decir, mediante la sustitución de las *URLs* asignadas a los nombres de archivos o insertando un nuevo paso que procese el contenido de los mismos.

Una de las claves de este Patrón de Diseño es que el cambio es transparente, y se utiliza de forma automática. Simplemente, el resto de los sistemas no tiene que saber si algo se ha añadido o ha cambiado, y puede seguir trabajando como antes. Para conseguir esto, es necesario implementar una interfaz predefinida para la extensión, requiriéndose algún tipo de mecanismo de envío en el que los *Interceptors* están registrados. Esto puede ser dinámico, en tiempo de ejecución, o estático.

5.5.8. Patrón de Inversión de Control

Inversión de Control (IoC)^[30] es un método de programación en el que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales.

Normalmente, la interacción se expresa de forma imperativa haciendo llamadas a procedimientos o funciones, donde el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.

En su lugar, en IoC se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir. En cierto modo es una implementación de la conocida frase: *No nos llames; nosotros te llamaremos*.

Patrón de Inyección de Dependencias

Este Patrón deriva directamente de IoC. Basa su funcionamiento en suministrar objetos a una clase en lugar de ser la propia clase quien lo cree.

La forma habitual de implementación es mediante un *Contenedor D.I.* y P.O.J.O.s. El contenedor inyecta a cada objeto los objetos necesarios según las relaciones plasmadas en un fichero de configuración. Éste es implementado por un *Framework* externo a la aplicación como Spring^[23], el cual se procede a estudiar en la Sección 5.17.

5.5.9. Patrón MVC

El Modelo Vista Controlador (M.V.C.) es un Patrón de Diseño que separa los datos y la Lógica de Negocio de una aplicación de la interfaz de usuario y

el módulo encargado de gestionar los eventos y las comunicaciones. Para ello propone la construcción de tres componentes distintos que son el **Modelo**, la **Vista** y el **Controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este Patrón aplica las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

Descripción del Patrón

El Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación de la (Lógica de Negocio). Envía a la **Vista** aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al **Modelo** a través del **Controlador**.

El Controlador: Responde a eventos (usualmente acciones del usuario), e invoca peticiones al **Modelo** cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una B.D.). También puede enviar comandos a su **Vista** asociada si se solicita un cambio en la forma en que se presenta de **Modelo** (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una B.D.), por tanto se podría decir que el **Controlador** hace de intermediario entre la **Vista** y el **Modelo**.

La Vista: Presenta el **Modelo** (información y Lógica de Negocio), en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto requiere de dicho **Modelo** la información que debe representar como salida.

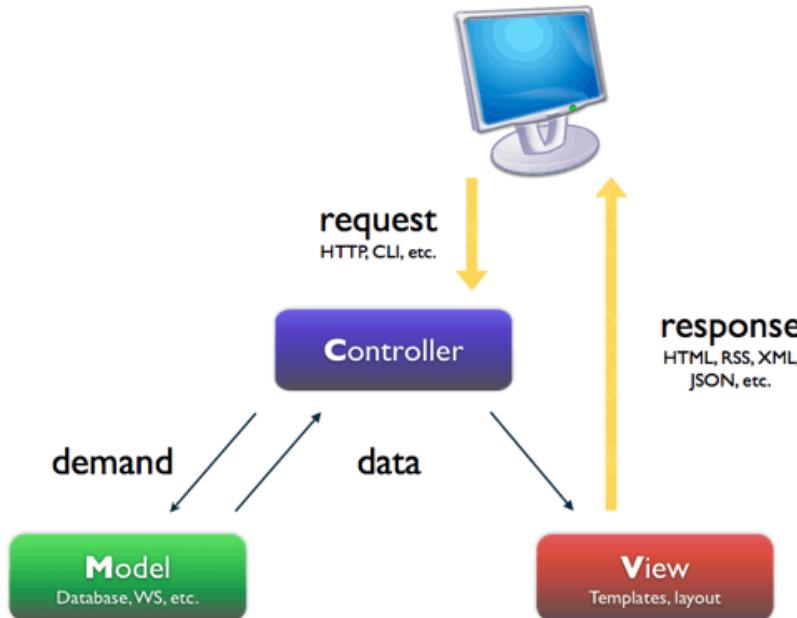


Figura 5.8: Arquitectura MVC.

5.6. Java

Java^[20] es una plataforma desarrollada al comienzo de los años 1990 con el objetivo concreto de permitir ejecutar programas sin tener relativamente en cuenta el hardware final, sin volver a reescribir todo el código del programa, ni tener que recomilar un programa para un cierto procesador. Consiste en tres grandes bloques, el lenguaje Java, una máquina virtual y una A.P.I..

Las aplicaciones de Java son generalmente compiladas a bytecode (clase Java) que puede ejecutarse en cualquier máquina virtual Java^[20], sin importar la arquitectura de la computadora subyacente. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra. Java es, a partir del 2012, uno de los lenguajes de programación más populares en uso, particu-

larmente para aplicaciones de cliente-servidor de Web.

Este lenguaje es de propósito general, como su propio nombre indica, puede ser usado para varios propósitos: acceso a bases de datos, comunicación entre computadoras, comunicación entre dispositivos, captura de datos, cálculos matemáticos, diseño de imágenes o páginas, crear sistemas operativos, manejadores de bases de datos, compiladores, etc.

Otra de sus características es ser concurrente, que quiere decir simultaneidad en la ejecución de múltiples tareas interactivas. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa.

Es por esto la mejor candidata para desarrollar un proyecto como el AMS. Además la curva de aprendizaje es mínima, ya que como estudiante de informática ya tenía conocimientos previos de este lenguaje y experiencia en el uso del Entorno de Desarrollo Integrado eclipse[51]. A todo se le suma un curso de programación online Java impartido por **MASTER-D**[32].

Para el desarrollo en Java del proyecto en Ubuntu[52] como sistema operativo principal, se ha usado OpenJDK[53], versión libre de la plataforma Java e implementación por defecto instalado en el S.O. citado.

JDK significa Java Development Kit, es un software que provee herramientas de desarrollo para la creación de programas en Java. OpenJDK[53] es la versión libre, la cual es mantenida por la comunidad y está basada en las especificaciones para cada una de las versiones de la JRE descritas por Sun Microsystems[13], empresa creadora de Java.



Figura 5.9: Logos de OpenJDK y Sun Microsystems.

5.7. JUnit

JUnit[43] es un conjunto de clases (*Framework*), que permite realizar la ejecución de clases Java de manera controlada (Pruebas Unitarias). De esta forma se puede evaluar el correcto funcionamiento de la Lógica de Negocio, aplicando de forma habitual un *Test Case* a cada clase. Los casos de prueba o *Test Case* son un conjunto de condiciones o variables bajo las cuáles se determina si el requisito de una aplicación es parcial o completamente satisfactorio.

En otras palabras, con JUnit se podrá evaluar el valor de retorno esperado en función de algún valor de entrada; si la clase cumple con la especificación, entonces se devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, se devolverá un fallo en el método correspondiente.



Figura 5.10: Logo de JUnit.

5.8. DDS-RTI

Data Distribution Service (DDS RTI)[8] está diseñado para atender las necesidades de las misiones y aplicaciones críticas de negocio como la negociación financiera, control del tráfico aéreo, la gestión de redes inteligentes, y otras aplicaciones con una gran cantidad de datos. El estándar está siendo utilizado cada vez más en una amplia gama de industrias, incluyendo sistemas inteligentes. Entre las diversas aplicaciones, DDS RTI se está utilizando actualmente en los sistemas operativos de smartphones, los sistemas y vehículos de transporte, radio definido por software, y proveedores de salud.

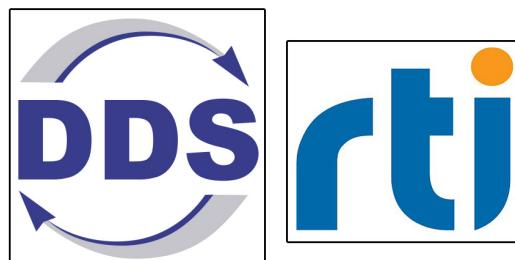


Figura 5.11: Logos de DDS y RTI.

5.8.1. Arquitectura DDS

Como se ha explicado justo antes, DDS RTI proporciona los medios y protocolos necesarios para que dos o más dispositivos se intercambien mensajes entre sí. De esta forma se explicará su funcionamiento con un ejemplo sencillo:

Un elemento de la red puede enviar un mensaje a otro elemento de la red. Para ello, el primer elemento, ubicado en un dominio o canal, publica el mensaje en el que está suscrito a su vez el elemento que debe recibir el mensaje, éste lo recibe y realiza el tratamiento que deba. El mensaje enviado tendrá definido su *QoS*, donde se especifica la calidad del servicio que debe llevar, ya que no es lo mismo enviar un mensaje de texto que se puede perder sin suponer preocupación ninguna, que enviar una orden a un avión que necesita saber si puede aterrizar en un momento crítico.

Componentes:

- **DomainParticipantFactory:** Factoría Singleton, principal punto de entrada a DDS RTI.
- **DomainParticipant:** Punto de entrada para la comunicación en un ámbito específico, representa la participación de una aplicación en un dominio DDS RTI. Además, actúa como una factoría para la creación de *Editores*

DDS RTI, *Suscriptores*, *Temas*, *MultiTopics* y *ContentFilteredTopics*.

- **TopicDescription:** Clase base abstracta para el *Topic*, *ContentFilteredTopic* y *MultiTopic*.
- **Topic:** Una especialización de *TopicDescription* que es la descripción más básica de los datos a ser suscritas y publicadas.
- **ContentFilteredTopic:** Un *TopicDescription* especializado como el *Topic* que permite, además, las suscripciones basadas en el contenido.
- **MultiTopic:** Una especialización de *TopicDescription* como el *Topic* que, además, permite a las suscripciones de combinar, filtrar y reorganizar los datos procedentes de varios temas.
- **Publisher:** Objeto responsable de la difusión actual de publicaciones.
- **DataWriter:** Permite que la aplicación establezca el valor de los datos que se publicará bajo un *Topic* determinado.
- **Subscriber:** Objeto responsable de la recepción real de los datos resultantes de sus suscripciones.
- **DataReader:** Permite la aplicación de declarar los datos que desea recibir (haciendo una suscripción con un tema, *ContentFilteredTopic* o *MultiTopic*) y para acceder a los datos recibidos por el Subscriber adjunta.
- **QoS:** Conjunto integral de políticas de *Calidad de Servicio*. Estos proporcionan control sobre el descubrimiento dinámico, enrutamiento y filtración basado en el contenido, la tolerancia a fallos y el comportamiento en tiempo real determinístico.

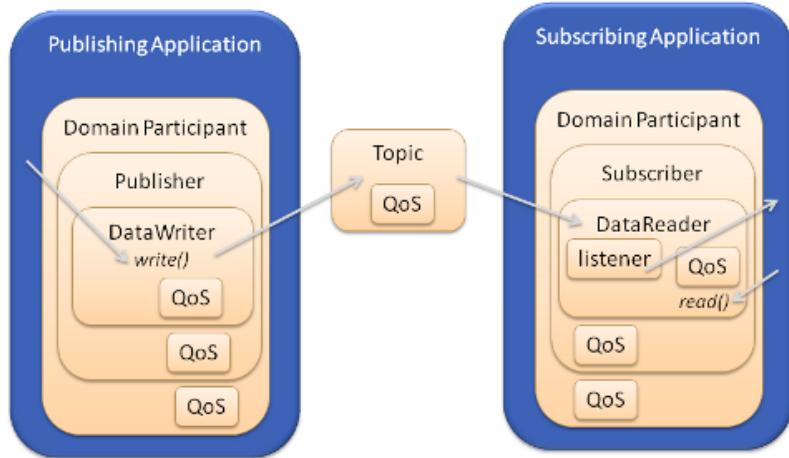


Figura 5.12: Architettura de DDS.

5.9. Google Protocol Buffer

Google nos proporciona unas librerías y utilidades para serializar estructuras de datos y usarlos en comunicaciones o almacenamiento, similar a las opciones existentes en torno a XML[19] (véase Sección 5.11). El punto fuerte de este componente es que la representación es más eficiente que las actuales basadas en XML. Esto se traduce en menos uso de memoria, mayor velocidad y mucho más simple de usar. Este lenguaje permite especificar mensajes y servicios. El compilador de *IDL* es capaz de generar código C++, Java o Python. Esto nos permite solucionar la comunicación de estructuras complejas de manera independiente del lenguaje y del sistema operativo.

Google Protocol Buffer[44] lee archivos de extensión .proto que después traduce al lenguaje apropiado. Por ejemplo, en Java transforma en una clase cada archivo de extensión .proto, a la que le añade los métodos y referencias automáticamente, simplificando muchísimo el trabajo del programador.



Figura 5.13: Logo de Google Protocol Buffer.

Se muestra un ejemplo sencillo de conversión de un archivo de extensión `.proto` a clase Java:

```
package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    repeated PhoneNumber phone = 4;
}

message AddressBook {
    repeated Person person = 1;
}
```

A continuación se ejecuta la librería de *Google Protocol Buffer*:

```
protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/addressbook.proto
```

Esto generará una clase Java `AddressBookProtos.java` en el directorio destino `com/example/tutorial/`. A continuación se muestra parte del código generado a partir del archivo de extensión `.proto`:

```
// required string name = 1;
public boolean hasName();
public String getName();

// required int32 id = 2;
public boolean hasId();
public int getId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;
public List<PhoneNumber> getPhoneList();
public int getPhoneCount();
public PhoneNumber getPhone(int index);
```

En el código generado se podrá encontrar un *Constructor* o *Builder*, Patrón de Diseño estudiado anteriormente en la Sección: 5.5.3. Este objeto será el encargado de generar las distintas instancias de los mensajes.

Las clases generadas por el compilador de *Google Protocol Buffer* son inmutables. Una vez que un objeto de mensaje se construye, no se puede modificar. Para construir un mensaje, primero debe construir un constructor, configurar los campos que desee con sus valores elegidos, y a continuación, llamar al método *build()* de la clase. El objeto devuelto es en realidad el mismo constructor en el que se llama al método, de forma que pueda encadenar varios emisores juntos en una sola línea de código. Se muestra un ejemplo de creación de un mensaje mediante el *Builder*:

```
Person john =
Person.newBuilder()
.setId(1234)
.setName("John Doe")
.setEmail("jdoe@example.com")
.addPhone(
    Person.PhoneNumber.newBuilder()
        .setNumber("555-4321")
        .setType(Person.PhoneType.HOME))
.build();
```

De esta forma se definirán los distintos tipos de mensajes en un archivo de extensión *.proto*, luego se compilarán con *Google Protocol Buffer* y generarán las clases que se usarán para la comunicación.

5.10. Boost

Boost[45] es un conjunto de librerías de software libre y revisión por pares preparadas para extender las capacidades del lenguaje de programación C++. Su diseño e implementación permiten que sea utilizada en un amplio espectro de aplicaciones y plataformas. Abarca desde librerías de propósito general hasta abstracciones del sistema operativo.

Con el objetivo de alcanzar el mayor rendimiento y flexibilidad se hace un uso intensivo de plantillas. Actualmente *Boost* está formada por más de 80 librerías individuales, incluidas las librerías de álgebra lineal, la generación de números pseudoaleatorios, multihilos, procesamiento de imágenes, expresiones regulares y Pruebas Unitarias entre otras muchas. La mayoría de las librerías *Boost* están basadas en cabeceras, funciones en línea y plantillas, por lo que no tienen que ser construidas antes de su uso.

Ésta librería es necesaria para poder usar el canal Middleware de comunicación creado por otras partes del proyecto PLANET.



Figura 5.14: Logo de Boost.

5.11. XML

eXtensible Markup Language (XML) es un lenguaje de marcas desarrollado por el W3C[12] utilizado para almacenar datos en forma legible, permitiendo definir la gramática de lenguajes específicos para estructurar documentos grandes.

En este proyecto, se utilizará este lenguaje para definir de forma sencilla una Red de Petri de la manera más abstracta y reutilizable posible. De esta forma, se podrá tener la información perfectamente estructurada, quedando los elementos bien definidos, que a su vez podrán contener otros elementos.

Ventajas:

- Es extensible: Después de diseñado y puesto en producción, es posible extender XML con la adición de nuevas etiquetas, de modo que se pueda continuar utilizando sin complicación alguna.
- El analizador es un componente estándar, no es necesario crear un analizador específico para cada versión de lenguaje XML. Esto posibilita el empleo de cualquiera de los analizadores disponibles. De esta manera se evitan bugs y se acelera el desarrollo de aplicaciones.
- Si un tercero decide usar un documento creado en XML, es sencillo entender su estructura y procesarla. Mejora la compatibilidad entre aplicaciones.
- Transformar datos en información, pues se le añade un significado concreto y los asocia a un contexto, con lo cual se consigue flexibilidad para estructurar documentos.



Figura 5.15: Logo de XML.

5.12. XStream Parser

XStream[54] es una librería para serializar objetos a XML y viceversa. Esto permitirá pasar el archivo XML formado de la Red de Petri directamente a clases contenedoras de los objetos necesarios para formar la misma, de una forma sencilla y rápida.

Características:

- **Facilidad de uso.** Suminista una capa de alto nivel, que simplifica los casos de uso comunes.
- **No hay asignaciones requeridas.** La mayoría de los objetos se pueden serializar sin necesidad de especificar asignaciones.
- **Rendimiento.** La velocidad y el bajo consumo de memoria son una parte fundamental del diseño, por lo que es adecuado para grandes gráficos de objetos o sistemas con un alto procesamiento de mensajes.
- **XML Limpio.** No se duplica la información obtenida a través de reflexión. Esto se traduce en un XML más fácil de leer para el usuario y más compacto que la serialización nativa de Java.
- **No requiere modificaciones a objetos.** Serializa campos internos, incluyendo privados y finales. Soporta clases no públicas e internas. Las clases no están obligados a tener constructor predeterminado.
- **Completo soporte gráfico de objetos.** Se mantendrán las referencias duplicadas encontradas en el objeto-modelo. Compatible con las referencias circulares.

- **Se integra con otras A.P.I. XML.** Mediante la implementación de una interfaz, *XStream* puede serializar directamente a o desde cualquier estructura de árbol (no sólo XML).
- **Estrategias de conversión personalizadas.** Las estrategias pueden ser registradas, permitiendo personalizar cómo se representan los tipos particulares como XML.
- **Mensajes de error.** Cuando se produce una excepción debido a un XML malformado, se ofrecen diagnósticos detallados para ayudar a aislar y solucionar el problema.
- **Formato de salida alternativo.** El diseño modular permite otros formatos de salida.



Figura 5.16: Logo de XStream.

5.13. HTML

HyperText Markup Language o Lenguaje de Marcado Hipertextual, hace referencia al lenguaje de marcado predominante para la elaboración de Página Web Dinámica que se utiliza para describir y traducir la estructura y la información en forma de texto, así como para complementar el texto con objetos tales como imágenes.

El *HTML* se escribe usando de *etiquetas*, rodeadas por corchetes angulares. Estas *etiquetas* describen distintos tipos de elementos que conforman el documento, estas se abren y cierran para definir en su interior nombres y distintos comportamientos, según el tipo de cada una de ellas.

En el siguiente ejemplo se pueden ver las *etiquetas* básicas o mínimas necesarias para crear un documento *HTML*:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Ejemplo1</title>
  </head>
  <body>
    <p>ejemplo1</p>
  </body>
</html>
```

HTML describe, hasta un cierto punto, la apariencia de un documento, y puede incluir o hacer referencia a un tipo de programa llamado script, el cual puede afectar el comportamiento de navegadores Web y otros procesadores de *HTML*.

5.14. JSP

JavaServer Pages (JSP)[46], es una tecnología que ayuda a los desarrolladores de software a crear Dinamic Web Apps basadas en *HTML*, entre otros tipos de documentos. JSP usa el lenguaje de programación Java con lo que encaja a la perfección en el desarrollo de este proyecto.

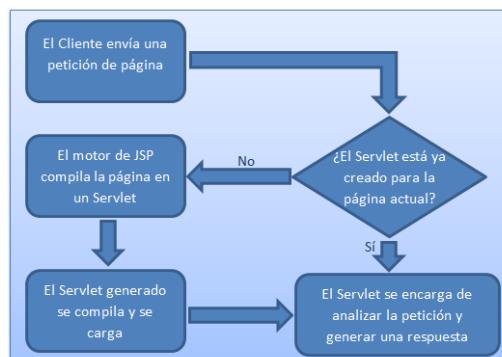


Figura 5.17: Pasos que siguen las páginas JSP.

5.14.1.JSTL

La *Java Server Pages Standard Tag Library* o *JSTL*[[47](#)], es un conjunto de 5 librerías de custom tags estandarizadas:

- Core (prefijo c): gestión de variables, control de flujo, gestión de URLs.
- XML[[19](#)] (prefijo x): similar a core, pero orientado a XML[[19](#)] incluyendo transformación con XSL.
- i18n (prefijo fmt): internacionalización, incluyendo formateo de fechas y números.
- SQL (prefijo sql): acceso a bases de datos con SQL, sólo para prototipos (la capa de presentación no debe acceder directamente a los datos).
- Funciones (prefijo fn): funciones de longitud de colecciones y manipulación de cadenas.

La versión actual cubre gran parte de las necesidades en la programación de páginas JSP, evitando el uso de scriptlets de código Java.

Para desplegar y ejecutar JSPs, es requerido un Servidor Web compatible con Contenedor de Servlets como Apache Tomcat[[48](#)], el cual se verá en detalle en la Sección 5.15.1.

5.15. Servlets

Los Servlets son clases de Java que dan una respuesta alternativa a la programación Web con *CGI*, ampliando su funcionalidad. Actúan de capa intermedia entre las peticiones de los navegadores Web u otros protocolos HTTP y la B.D. o aplicaciones en el Servidor Web HTTP. El protocolo HTTP sobre el que se construye la Web, funciona por medio de un mecanismo de solicitudes y respuestas en el que un Servidor Web recibe una solicitud, la procesa y devuelve la respuesta correspondiente. El A.P.I. Java Servlet modela este proceso y lo orienta a objetos para que el código que se programa pueda procesar solicitudes del proveedor y responderlas de forma automática. Por ejemplo, un Servlet puede utilizar los datos de un formulario *HTML* de introducción de pedidos para actualizar la B.D. de pedidos de una empresa.

Se ejecutan en un Servidor Web dentro de un Contenedor de Servlets y construyen Página Webs, como se aprecia en la figura:

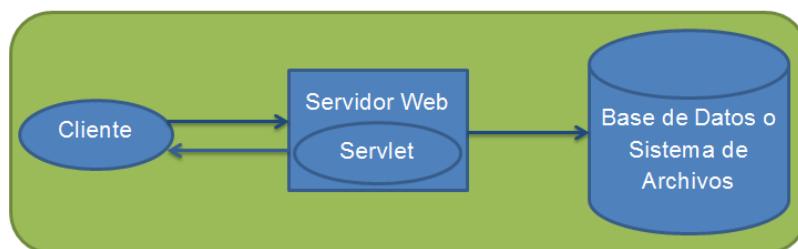


Figura 5.18: Ejecución de Servlets.

El Contenedor Web carga, ejecuta y administra el Servlet, siguiendo el proceso que se describe a continuación:

- El cliente envía una solicitud de página al Contenedor de Servlets.
- En caso de que el Servlet no se encuentre cargado, el Contenedor de Servlets lo carga. Una vez cargado tras la primera solicitud, permanece en este es-

tado hasta que el Contenedor de Servlets decide descargarlo.

- El Contenedor de Servlets envía la información de la solicitud al Servlet, creando un nuevo hilo para cada solicitud a ejecutar.
- El Servlet procesa la solicitud, construye una respuesta y la transmite al Contenedor de Servlets.
- El Contenedor de Servlets envía la respuesta de nuevo hasta el cliente.

5.15.1. Contenedor de Servlets

El Contenedor de Servlets es uno de los elementos más importantes del Servidor Web, ya que se encarga de cargar y de inicializar el Servlet. Puede procesar varias instancias de un Servlet.

Su función es determinar el Servlet que debe recibir una determinada solicitud, comprueba que se devuelve la respuesta al cliente y, por último, elimina el Servlet una vez terminada su vida operativa.

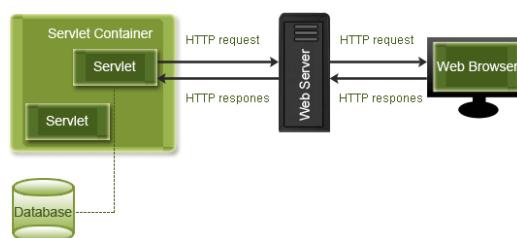


Figura 5.19: Procesamiento de peticiones con un contenedor de Servlets.

Apache Tomcat

Apache Tomcat[48] funciona como un Contenedor de Servlets desarrollado bajo el proyecto Jakarta en la Apache Software Foundation. Implementa las especificaciones de los Servlets y de JavaServer Pages (JSP).

Se constituye como una jerarquía anidada de componentes, donde un mismo tipo de componente puede aparecer en diversos puntos de la jerarquía. Es importante entender esta jerarquía a la hora de configurar y administrar el Servidor Web.

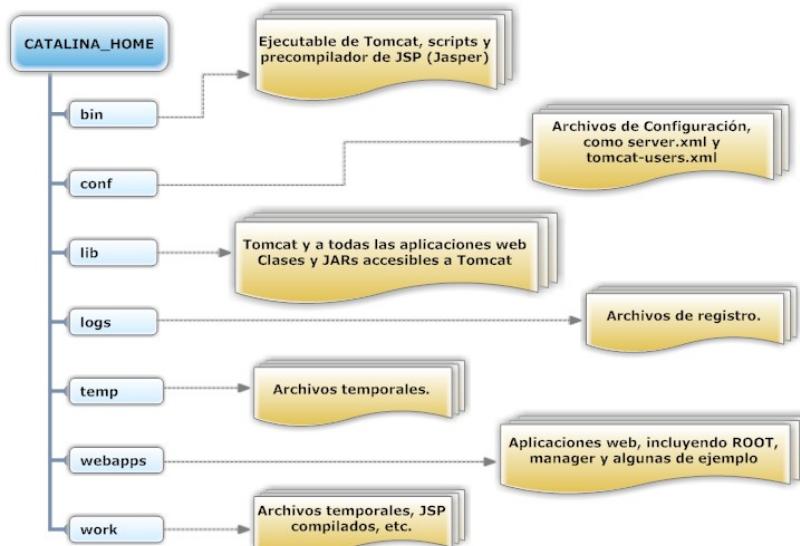


Figura 5.20: Arquitectura de componentes de Tomcat.

El conjunto de todos los Servlets, JSP y demás ficheros que estén relacionados lógicamente constituye una Aplicación Web, empaquetada en un archivo de extensión .war. La especificación Servlet define una jerarquía de directorio estándar para los archivos de extensión .war. Se describe en la siguiente figura:

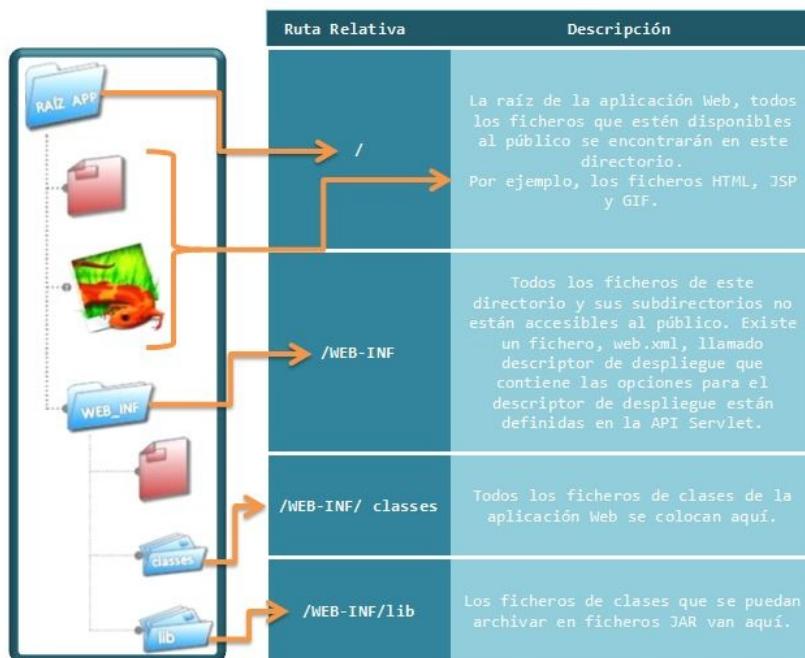


Figura 5.21: Jerarquía estándar de Tomcat.

Un archivo de extensión .war se construye mediante el comando *jar* de Java, ya sea de forma directa o mediante una herramienta de construcción como *Maven*.

Todos los Servlet Container deben utilizar la misma jerarquía de directorios para los archivos de extensión .war. Es más, la localización y características del descriptor de despliegue se establecen en la especificación, por lo que las Web Apps necesitan configurarse solo una vez y serán compatibles con cualquier Contenedor de Servlets.

El descriptor de Servlets define opciones como el orden en que los Servlets se cargan en el Contenedor de Servlets, los parámetros que se pasan a los Servlets al iniciarse, restricciones de seguridad, etc. Esto significa que los desarrolladores sólo tienen que crear una Aplicación Web una sola vez. Por tanto, distribuir y desplegar Web Apps es muy simple, incluso si se cambia de Contenedor de Servlets.

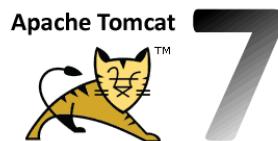


Figura 5.22: Logo de Tomcat.

5.16. Struts 2

Cuando se desarrollan Web Apps es importante que éstas puedan ser creadas de forma rápida y eficiente. Para ello, el Patrón de Diseño M.V.C. nos ofrece la capa **Controlador**, la cual se encarga de hacer la mayor parte del trabajo tedioso y repetitivo.



Figura 5.23: Logo de Struts 2.

Struts2^[21] es un *Framework* que implementa directamente la capa **Controlador** y su objetivo es muy sencillo: hacer que el desarrollo de Web Apps sea simple para los desarrolladores.

Struts2 proporciona también algunos componentes para la capa de **Vista**. Además proporciona una integración perfecta con otros *Frameworks* para implementar la capa del **Modelo** (como Spring e Hibernate^[22], explicado brevemente en el Capítulo Anexo 11).

Para hacer más fácil presentar datos dinámicos, el *Framework* incluye una librería de etiquetas Web. Las etiquetas interactúan con las validaciones y las características de internacionalización del *Framework*, para asegurar que las entradas son válidas, y las salidas están localizadas.

Además, mediante el uso de plugins se pueden agregar más funcionalidades de forma transparente, ya que no tienen que ser declarados ni configurados de ninguna forma.

5.16.1. Componentes

El núcleo de Struts2 es un filtro, conocido como el *FilterDispatcher*. Éste es el punto de entrada del *Framework*, desde donde se lanza la ejecución de todas las peticiones que involucran al mismo.

Las principales responsabilidades del *FilterDispatcher* son:

- Ejecutar los *Actions*, que son los manejadores de las peticiones.
- Comenzar la ejecución de la cadena de *Interceptors*.
- Limpiar el *ActionContext*, para evitar fugas de memoria.

Las peticiones se procesan usando tres elementos principales: *Interceptors*, *Actions* y *Results*.

Interceptors

Los *Interceptors* son clases que siguen el Patrón de Diseño *Interceptor* (véase Sección 5.5.7). Estos permiten que se implementen funcionalidades cruzadas o comunes para todos los *Actions*, pero que se ejecuten fuera del *Action* (por

ejemplo validaciones de datos, conversiones de tipos, población de datos, etc).

Éstos realizan tareas antes y después de la ejecución de un *Action* y también pueden evitar que un *Action* se ejecute (por ejemplo si se está haciendo alguna validación que no se ha cumplido). También sirven para ejecutar algún proceso particular que se quiere aplicar a un conjunto de *Actions*. De hecho muchas de las características con que cuenta Struts2 son proporcionadas por los *Interceptors*.

Si alguna funcionalidad que necesitamos no se encuentra en los *Interceptors* de Struts2 podemos crear nuestro propio *Interceptor* y agregarlo a la cadena que se ejecuta por defecto. De la misma forma, podemos modificar la cadena de *Interceptor* de Struts2, por ejemplo para quitar un *Interceptor* o modificar su orden de ejecución.

La siguiente tabla muestra solo algunos de los *Interceptors* más importantes que vienen integrados y pre-configurados en Struts2:

Interceptor	Nombre	Descripción del Interceptor
Alias	alias	Permite que los parámetros tengan distintos nombres entre peticiones.
Chaining	chaining	Permite que las propiedades del <i>Action</i> ejecutado previamente estén disponibles en el <i>Action</i> actual.
Checkbox	checkbox	Ayuda en el manejo de checkboxes agregando un parámetro con el valor <i>false</i> para checkboxes que no están marcados.
Conversion Error	conversionError	Coloca información de los errores convirtiendo cadenas a los tipos de parámetros adecuados para los campos del <i>Action</i> .
Create Session	createSession	Crea de forma automática una sesión HTTP si es que aún no existe una.
Execute and Wait	executeAndWait	Envía al usuario a una página de espera intermedia mientras el <i>Action</i> se ejecuta en background.
File Upload	fileUpload	Hace que la carga de archivos sea más fácil de realizar.
Logging	logger	Proporciona un logging simple, mostrando el nombre del <i>Action</i> que se está ejecutando.
Parameters	params	Establece los parámetros de la petición en el <i>Action</i> .
Prepare	prepare	Llama al método <i>prepare</i> en los <i>Actions</i> que implementan la Interfaz <i>Preparable</i> .
Servlet Configuration	servletConfig	Proporciona al <i>Action</i> acceso a información basada en Servlets.
Roles	roles	Permite que el <i>Action</i> sea ejecutado sólo si el usuario tiene uno de los roles configurados.
Timer	timer	Proporciona una información sencilla de cuánto tiempo tarde el <i>Action</i> en ejecutarse.
Validation	validation	Proporciona a los <i>Actions</i> soporte para validaciones de datos.
Workflow	workflow	Redirige el objeto tipo <i>Result INPUT</i> sin ejecutar el <i>Action</i> cuando una validación falla.

Tabla 5.1: Tabla de los *Interceptors* más conocidos.

Cada *Interceptor* proporciona una característica distinta al *Action*. Para sacar la mayor ventaja posible de los *Interceptors*, un *Action* permite que se aplique más de un *Interceptor*. Para lograr esto Struts2 permite crear pilas o stacks de *Interceptors* y aplicarlas a los *Actions*. Cada *Interceptor* es aplicado en el orden en el que aparece en la pila. También podemos formar pilas de *Interceptors* en base a otras pilas.

Actions

Las Acciones o *Actions* son clases encargadas de realizar la lógica para servir una petición. Cada URL es mapeada a una *Action* específica, la cual proporciona la lógica necesaria para servir a cada petición hecha por el usuario. Estrictamente hablando, las *Actions* no necesitan implementar una interfaz o extender de alguna clase base. El único requisito para que una clase sea considerada un *Action* es que debe tener un método que no reciba argumentos y que devuelva un objeto *String* o un objeto de tipo *Result*. Por defecto el nombre de este método debe ser *execute* aunque podemos ponerle el nombre que queramos y posteriormente indicarlo en el archivo de configuración de Struts2.

Cuando el resultado es un *String*, el objeto tipo *Result* correspondiente se obtiene de la configuración del *Action*. Esto se usa para generar una respuesta para el usuario.

Los *Actions* pueden ser P.O.J.O.s que cumplan con el requisito anterior, aunque por lo general, también pueden implementar la Interfaz "com.opensymphony.xwork2.Action" o extender una clase base que proporciona Struts2: "com.opensymphony.xwork2.ActionSupport", lo cual hace más sencilla su creación y manejo.

La clase *ActionSupport* implementa la interfaz *Action* y contiene una implementación del método *execute()* que devuelve el valor *SUCCESS*. Además proporciona unos cuantos métodos para establecer mensajes, tanto de error como informativos, que pueden ser mostrados al usuario.

```
public interface Action
{
```

```
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

    public String execute() throws Exception;
}
```

Results

Después de que un *Action* haya sido procesado, se debe enviar la respuesta de regreso al usuario, esto se realiza usando *Results*. Este proceso tiene dos componentes, el tipo del objeto *Result* y el resultado mismo.

El tipo del *Result* indica cómo debe ser tratado el resultado que se le devolverá al cliente. Por ejemplo un tipo de *Result* puede enviar al usuario de vuelta un objeto JSP mientras que otro puede redirigirlo hacia otro sitio.

Un *Action* puede tener más de un objeto tipo *Result* asociado. Esto nos permitirá enviar al usuario a una **Vista** distinta dependiendo del resultado de la ejecución del *Action*. Por ejemplo, en caso de que todo salga bien, enviaremos al usuario al objeto tipo *Result sucess*, si algo sale mal lo enviaremos al objeto tipo *Result error*, o si no tiene permisos lo enviaremos al objeto tipo *Result denied*.

Funcionamiento

Una vez estudiados a grandes rasgos los componentes que forman Struts2 se puede entender cómo son procesadas las peticiones por cualquier aplicación desarrollada con este *Framework*.

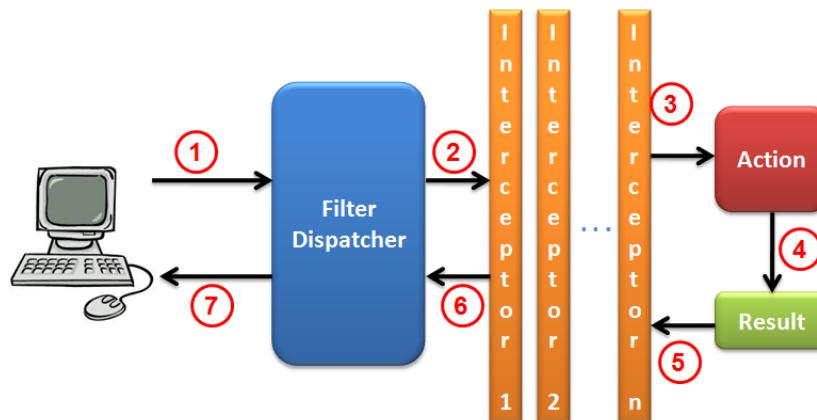


Figura 5.24: Secuencia de pasos de Struts2.

Los pasos que sigue una petición son:

- 1.- El navegador Web hace una petición para un recurso de la aplicación (index.action, reporte.pdf, etc). El filtro o filtros de Struts2 revisa la petición y determina el *Action* apropiado para servirla.
- 2.- Se aplican los *Interceptors*, los cuales realizan algunas funciones como validaciones, flujos de trabajo, manejo de la subida de archivos, etc.
- 3.- Se ejecuta el método adecuado del *Action* (por defecto el método execute), este método normalmente almacena o devuelve alguna información referente al proceso.
- 4.- El *Action* indica cuál objeto tipo *Result* debe ser aplicado. El objeto tipo *Result* genera la salida apropiada dependiendo del resultado del proceso.
- 5.- Se aplican al resultado los mismos *Interceptors* que se aplicaron a la petición, pero en orden inverso.

- 6.- El resultado vuelve a pasar por el *FilterDispatcher* aunque este ya no hace ningún proceso sobre el resultado (por definición de la especificación de Servlets, si una petición pasa por un filtro, su respuesta asociada pasa también por el mismo filtro).
- 7.- El resultado es enviado al usuario y éste lo visualiza.

Estos pasos no siempre siguen el mismo orden. Por ejemplo, si el *Interceptor* de validación de datos detecta algún problema, el *Action* no se ejecutará, y será el mismo *Interceptor* el que se encargará de enviar un objeto tipo *Result* al usuario.

5.17. Spring

Spring[23] es un *Framework Open Source*[24], que surge fundamentalmente para simplificar la programación de aplicaciones Java de tipo empresarial, que hasta entonces se estaban llevando a cabo con *Frameworks* como *EJB* (*Enterprise Java Beans*).

Está diseñado para no ser intrusivo, esto significa que no es necesario que la aplicación extienda o implemente alguna clase o Interfaz de Spring, por lo que el código de lógica quedará libre y completamente reutilizable para un proyecto sin Spring. Gracias a esto es posible usar un P.O.J.O. o un objeto Java para hacer cosas que antes sólo podían hacerse con *EJBs*. Sin embargo su uso no es solo para el desarrollo de *Web Apps*, cualquier aplicación Java puede beneficiarse de su uso.

El núcleo de Spring está basado en un Patrón de Diseño llamado Inversión de Control (IoC). De esta forma, las aplicaciones pueden usar archivos de configuración gXML o anotaciones en el código para describir las dependencias entre sus componentes. Con IoC, la aplicación no controla su estructura, sino que permite que sea el *Framework* de Spring quien lo haga.

Se muestra un ejemplo, donde se crea una clase *AlmacenUsuario*, que depende de una instancia de una clase *UsuariosDAO* para realizar su tarea. *AlmacenUsuario* crea una instancia de *UsuariosDAO* usando el operador *new*. Usando la técnica de Inversión de Control (IoC)[30], una instancia de *UsuariosDAO*, o una subclase de esta, es proporcionada a *AlmacenUsuario* en tiempo de ejecución por el motor de Spring. En este caso *UsuariosDAO* también podría ser una Interfaz y Spring se encargará de proporcionar una instancia de una clase que implemente esa Interfaz. Este proceso ha sido posible gracias a la Inyección de Dependencias.

5.17.1. Inyección de Dependencias en Spring

El uso de interfaces y D.I. son mutuamente benéficos, ya que hace más flexible y robusta cualquier aplicación y es mucho más fácil realizar Pruebas Unitarias. La clave está en que los componentes no saben qué implementación concreta de otros componentes están usando; solo ven sus interfaces.

Spring proporciona las instancias de las clases a la aplicación de forma no intrusiva y automática. Para ello, lo único que hay que hacer es crear un archivo de configuración que describa las dependencias; Spring se encargará del resto. Su funcionamiento es el de un contenedor, ya que no solo crea los componentes de la aplicación, sino que contiene y maneja al ciclo de vida y configuración de estos componentes. De esta forma, se puede declarar cómo debe ser creado cada uno de los objetos de nuestra aplicación, cómo deben ser configurados, y cómo deben asociarse con los demás.

Con la implementación de D.I. se consigue Acoplamiento Débil(véase Sección 5.3). Esto implica que los componentes de la aplicación deben asumir lo menos posible acerca de otros componentes. La forma más fácil de lograr este comportamiento en Java es mediante el uso de Interfaces. Como cada componente de la aplicación solo es consciente de la Interfaz de otros componentes, se puede cambiar la implementación del alguno de ellos sin afectar al resto.

Ventajas del uso de D.I.:

- **Se reduce el *código pegamento*:** esto quiere decir que se reduce drásticamente la cantidad de código que se debe escribir para unir los distintos componentes una aplicación, proporcionando búsquedas automáticas para instanciar objetos remotos.
- **Se externalizan dependencias:** al ser posible colocar la configuración de dependencias en archivos gXML, se puede realizar una reconfiguración fácilmente, sin necesidad de recompilar el código. De la misma forma, es posible realizar el cambio de la implementación de una dependencia a otra.
- **Las dependencias se manejan en un solo lugar:** toda la información de dependencias es responsabilidad de un sólo componente, el Contenedor de IoC de Spring, proporcionando un manejo de dependencias más simple y menos propenso a errores.
- **Hace que las pruebas sean más fáciles:** como las clases serán diseñadas para hacer fácil el reemplazo de dependencias, se podrán proporcionar objetos simulados que regresen datos de prueba, de servicios o cualquier dependencia que necesite el componente que estamos probando.

5.17.2. Programación Orientada a Aspectos

La P.O.A. es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de responsabilidades.

El principal objetivo de la P.O.A. es la separación de las funcionalidades dentro del sistema, por un lado funcionalidades comunes utilizadas a lo largo de la aplicación, y por otro lado, las funcionalidades propias de cada módulo. Cada funcionalidad común se encapsulará en una entidad.

Gracias a esto se pueden encapsular los diferentes conceptos que componen

una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reutilizables.

En Spring esto se configura mediante Beans y un archivo de configuración de P.O.A., buscando encapsular todo el código reutilizable. Con estas sencillas condiciones, el código se vuelve más sencillo, limpio, y por lógica, comprensible para cualquiera que lo lea.

5.17.3. Componentes de Spring

Spring está dividido en alrededor de 20 módulos y colocados en los siguientes grupos:

- **IoC Container:** los componentes no crean, o buscan las referencias a otros componentes que necesiten para realizar su trabajo, sino que simplemente declaran qué dependencias tienen, y el IoC Container les proporciona automáticamente estas dependencias, usando para ello *IoC + D.I.*.
- **Acceso a Datos / Integración:** en este grupo, Spring mapea las *SQLException* a excepciones específicas y automatiza la gestión de conexiones. Se declara una fuente de datos y Spring la gestiona.
- **WEB:** En este grupo se encuentran las herramientas para implementar el Patrón de Diseño Modelo Vista Controlador y el acceso a componentes remotos. Esto facilita el desarrollo de aplicaciones distribuidas y reduce el código que se necesita para exponer un Beans como servicio o para acceder desde un Beans a un servicio remoto. También unifica distintas A.P.I. para la gestión de transacciones.
- **Programación Orientada a Aspectos:** Permite combinar cierto código

con otro para añadir cierta funcionalidad al original sin necesidad de modificarlo, facilitando así la implementación de funcionalidades transversales de una aplicación.

- **Instrumentación:** Este módulo facilita las herramientas para la medida de rendimiento y métricas de utilizaciones de recursos, así como estadísticas generales para las operaciones y la gestión de transacciones.
- **Pruebas:** Soporte de clases para desarrollo de Pruebas Unitarias y Pruebas de Integración.

Estos grupos se muestran en la siguiente imagen:

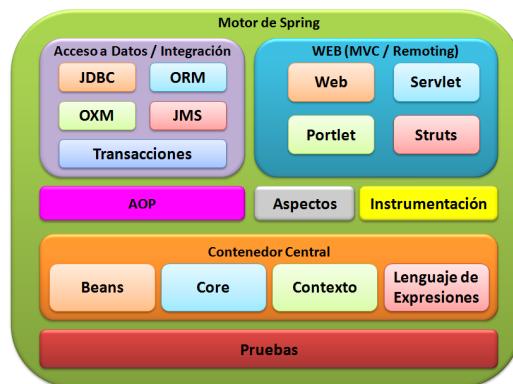


Figura 5.25: Módulos de Spring.

En este proyecto nos centraremos en el grupo Web, con el que implementaremos el Patrón de Diseño Modelo Vista Controlador junto con el Framework Struts2.

5.18. Maven

Maven^[39] es una herramienta de software para la gestión y construcción de proyectos Java.

Maven utiliza un POM^[41] para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. El motor incluido en su núcleo puede dinámicamente descargar plugins de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos Open Source^[24] en Java de Apache y otras organizaciones y desarrolladores. Una caché local de *artefactos* actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

5.18.1. El POM

El Project Object Model (POM) no es más que la abstracción usada por *Maven* para definir los proyectos, como tal, contiene los atributos de estos y las instrucciones para construirlo.

Un proyecto en *Maven* se define mediante este archivo, en el cual se definen cosas como las instrucciones para compilar el proyecto, las librerías necesarias, etc.

La ejecución de un archivo POM siempre genera un *artefacto*. Este *artefacto* puede ser cualquier cosa: un archivo jar, un swf de flash, un archivo zip o el mismo archivo POM. *Maven* trabaja modularizando los proyectos, de esta forma se tienen varios módulos que conforman un sólo proyecto. Para denotar esta relación en *Maven*, se crea un proyecto padre de tipo POM y los módulos se definen como otros archivos POM que heredan del primero. Esta organización

sirve para centralizar en el POM padre las variables (como el nombre del proyecto o el número de versión), las dependencias, los repositorios, etc. que son comunes a los módulos, eliminando duplicidad de código.

Para crear un proyecto en *Maven*, lo más sencillo es usar los *Maven Archetypes*. Los arquetipos son *artefactos* especiales de *Maven* que sirven como plantillas para crear proyectos. *Maven* cuenta con algunos predefinidos y terceros han hecho los suyos para crear proyectos con tecnologías específicas, como proyectos Web con *Spring* o proyectos con *Adobe Flex*[40].

Componentes de un arquetipo:

- **groupId:** Nombre de la empresa u organización, ya que conceptualmente todos los proyectos con ese groupId pertenecen a una sola empresa.
- **artifactId:** Es el nombre del *artefacto*.
- **version:** Número de versión del *artefacto*.
- **package:** Paquete base donde irá el código del *artefacto*.

Ejemplo de definición de un POM:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Ejemplo de definición de un Arquetipo:

```
<groupId>my.groupId</groupId>
<artifactId>my-archetype-id</artifactId>
<version>1.0-SNAPSHOT</version>
```

```
<packaging>jar</packaging>
```

Maven descarga sus dependencias y las dependencias de los proyectos de un repositorio central. Por defecto, usa el repositorio central de *Maven*, aunque puede usar otros. Estas dependencias las almacena en un repositorio local, que no es otra cosa que una carpeta local con el fin de no tener que volver a descargarlas otra vez. Dentro de este repositorio, *Maven* coloca los *artefactos* en carpetas de acuerdo a su *groupId*, *artifactId* y *version*.

Aunque podríamos hablar mucho más acerca del POM, ya tenemos los conocimientos básicos necesarios para entender su funcionamiento. Por último, aclarar que en este proyecto hemos utilizado el plugin de eclipse m2e[42] , el cual nos permite trabajar en nuestro Entorno de Desarrollo Integrado con las herramientas de *Maven*.



Figura 5.26: Logo de Maven.



Capítulo 6. Control de Versiones

En cualquier proyecto desarrollado es muy recomendable tener un control de versiones para poder revertir cambios, volver a un estado anterior del código, asignar etiquetas a esos cambios e incluso para tener una copia del mismo en la nube para evitar posibles desastres.

Por definición, se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación. Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión, usando repositorios.

Un repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas.

6.1. Arquitecturas de almacenamiento

Podemos clasificar los sistemas de Control de Versiones atendiendo a la arquitectura utilizada para el almacenamiento del código:

- **Centralizados:** existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS[36] y Subversion[33].
- **Distribuidos:** cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Algunos ejemplos son Git[34] y Mercurial[35].

Ventajas de sistemas distribuidos:

- Necesita menos veces estar conectado a la red para hacer operaciones. Esto produce una mayor autonomía y una mayor rapidez.
- Aunque se caiga el repositorio remoto la gente puede seguir trabajando.
- Al hacer los distintos repositorios una réplica local de la información de los repositorios remotos a los que se conectan, la información está muy replicada y por tanto el sistema tiene menos problemas en recuperarse si por ejemplo se quema la máquina que tiene el repositorio remoto. Por tanto hay menos necesidad de backups, aunque siguen siendo necesarios para resolver situaciones en las que cierta información todavía no haya sido replicada.

- Permite mantener repositorios centrales más limpios en el sentido de que un usuario puede decidir si ciertos cambios realizados por él en el repositorio local no son relevantes para el resto de usuarios y por tanto no permite que esa información sea accesible de forma pública.
- El servidor remoto requiere menos recursos que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los repositorios locales.
- Al ser los sistemas distribuidos más recientes que los sistemas centralizados, y al tener más flexibilidad por tener un repositorio local y otro/s remotos, estos sistemas han sido diseñados para hacer fácil el uso de ramas (creación, evolución y fusión), y poder aprovechar al máximo su potencial. Por ejemplo se pueden crear ramas en el repositorio remoto para corregir errores o crear funcionalidades nuevas. Pero también se pueden crear ramas en los repositorio locales para que los usuarios puedan hacer pruebas y dependiendo de los resultados fusionarlos con el desarrollo principal o no. Las ramas dan una gran flexibilidad en la forma de trabajo.

Ventajas de sistemas centralizados

- En los sistemas distribuidos hay menos control a la hora de trabajar en equipo ya que no se tiene una versión centralizada de todo lo que se está haciendo en el proyecto.
- En los sistemas centralizados las versiones vienen identificadas por un número de versión. Sin embargo en los sistemas de control de versiones distribuidos no hay números de versión, ya que cada repositorio tendría sus propios números de revisión dependiendo de los cambios. En lugar de eso cada versión tiene un identificador al que se le puede asociar una etiqueta.

6.1.1. Mercurial

Mercurial[35] está implementado principalmente haciendo uso del lenguaje de programación **Python**, pero incluye una implementación binaria de *diff* escrita en **C**. Aunque fue escrito originalmente para funcionar sobre **Linux**, ha sido adaptado para **Windows**, **Mac OS X** y la mayoría de otros sistemas tipo **Unix**.

Mercurial[35] es, sobre todo, un programa para la línea de comandos. Todas sus operaciones se invocan como opciones dadas a su programa motor, *Hg* (cuyo nombre hace referencia al símbolo químico del mercurio).

Las principales metas de desarrollo de Mercurial incluyen un gran rendimiento y escalabilidad; desarrollo completamente distribuido, sin necesidad de un Servidor Web; gestión robusta de archivos tanto de texto como binarios; y capacidades avanzadas de ramificación e integración, todo ello manteniendo sencillez conceptual.

Como *GUI* o *Interfaz Gráfica de Usuario*, hemos utilizado TortoiseHg[37] en este proyecto, que enlaza con un servicio de alojamiento de código en la nube con *Bitbucket*[38].



Figura 6.1: Logos de *Mercurial* y *TortoiseHg*.

Parte IV

Desarrollo del Proyecto



Capítulo 7. Planificación

Se procede a desarrollar un proyecto de investigación, en el que la planificación temporal no cumple un papel tan importante, ya que en el proceso de investigación es difícil poner límites temporales, ya que no se puede estimar exactamente en cuanto tiempo el investigador va a aprender un tema determinado. Además, no es necesario tener un entregable listo a cada fin de *Sprint*, ni disponemos de un cliente que requiera completar ciertos plazos. Aún así si se deben de cumplir ciertos hitos dentro del proyecto PLANET, como por ejemplo establecer una fecha de integración con los demás socios, donde la funcionalidad de cada uno debe quedar probada. Estos hitos son fijados por la figura del Product Owner y deben de cumplirse sin excepción.

El desarrollo inicial del proyecto se establece en seis meses. En la elaboración del mismo se usarán dos métodos de desarrollo.

En primer lugar se usará Scrum[17] que, junto con Pair Programming ayudarán al programador inexperto a adaptarse a la dinámica de trabajo y a valerse por sí mismo. Esto ocupará los tres primeros meses, dividiendo cada mes en *Sprints* de dos semanas de duración cada uno, donde se llevará a cabo cada una de las Historias de Usuario contenidas en ellos.

Una vez adquirida la experiencia suficiente, se usará Kanban, donde el desarrollo será algo más flexible. De esta forma desaparecen las Historias de Usuario propiamente dichas y se transforman en tareas a realizar conforme se vaya terminando la anterior o vaya surgiendo en el desarrollo.

7.1. Pasos a realizar

En primer lugar, como *Sprint 0*, tendrá lugar la preparación y adaptación del entorno de trabajo, a partir de este punto, los *Sprints* se sucederán conforme se vayan alcanzando los objetivos, con una duración de dos semanas cada uno. Las reuniones de *Scrum* se realizarán cada mañana, con una duración de 5 minutos, donde el Programador Junior expondrá al Scrum Master los avances realizados y las posibles dudas surgidas durante el desarrollo.

El siguiente paso será dar forma a la Red de Petri, obteniendo como producto final una Red de Petri, funcional. Para dar una mayor versatilidad a la Red de Petri, se hará uso de un Parseador de XML pudiendo así crear cómodamente una red desde un archivo de XML.

Una vez terminada por completo, se procederá a integrar el envío y recepción de señales por la red, a través de la plataforma de mensajes DDS RTI suministrada por la UDE. Por último, se desarrollará una Aplicación Web donde visualizar todos y cada uno de los estados de la Red de Petri, así como interactuar con ella mediante las señales.

Para el desarrollo de cada una de las partes se harán las correspondientes Pruebas Unitarias, y en la medida de lo posible, cuando se vaya terminando cada una, se harán las oportunas Pruebas de Integración, todo esto lo veremos al final del desarrollo de cada elemento.

7.2. Historias de Usuario

Las Historias de Usuario se definen en la primera reunión que se realiza con el cliente, y se genera el *Backlog*. Esta reunión es esencial y una de las más importante puesto que el objetivo es detallar las Historias de Usuario. En esta reunión se tiene que obtener del cliente toda la información necesaria para

comenzar a trabajar. Se escuchan sus peticiones y se recogen usando el lenguaje más sencillo posible, escribiendo frases simples y evitando ambigüedades.

Dada la naturaleza de este proyecto, no existe la figura del cliente, con lo que las Historias de Usuario se definen conforme a las necesidades del sistema a desarrollar. Éstas son las Historias de Usuario desarrolladas en un primer momento, en el que no se contempla el uso de un Middleware ni de la tecnología DDS RTI, que aún están por definir:

9990 - **Montaje del Entorno:** los desarrolladores quieren un entorno de desarrollo en donde poder trabajar.

9080 -**Canal de comunicaciones para el estado de la Red de Petri:** es necesario que el AMS sea capaz de comunicar su estado en cada momento.

9070 -**Canal de comunicaciones para el Radar:** es necesario que el AMS sea capaz de comunicarse con el Radar, y que esta comunicación pueda ser utilizada directamente desde la Red de Petri.

9060 -**Canal de comunicaciones para la estación meteorológica:** es necesario que el AMS sea capaz de comunicarse con la estación meteorológica, y que esta comunicación pueda ser utilizada directamente desde la Red de Petri.

9050 -**Canal de comunicaciones para los UAV:** es necesario que el AMS sea capaz de comunicarse con un UAV, y que esta comunicación pueda ser utilizada directamente desde la Red de Petri.

9040 -**Canal de comunicaciones para los UGV:** es necesario que el AMS sea capaz de comunicarse con un UGV, y que esta comunicación pueda ser utilizada directamente desde la Red de Petri.

9030 -**Visualización del estado del AMS:** disponer de una vista que represente el estado completo del aeropuerto.

9020 -**Visualización del Radar:** disponer de una vista que represente el estado del Radar.

Antes de comenzar el primer *Sprint* y tras varias reuniones con el resto de los socios y comprender mejor las especificaciones del proyecto, queda contemplado ya el uso de DDS RTI como método para recibir y enviar las señales, por lo que se procede a cambiar las Historias de Usuario a partir del segundo *Sprint*. Además se dividen mejor algunas de ellas que anteriormente eran demasiado amplias (épicas, en el lenguaje de Scrum). Varios Sprints después, se decidirá usar Kanban para el desarrollo de la Aplicación Web debido a las particularidades del proyecto en esta fase. Hay que tener en cuenta que, en la parte de desarrollo usando Kanban, aunque se han definido historias de usuario no se han reflejado en los diagramas de burndown, ya que, al ser dos metodologías distintas, desvirtuarían las mediciones realizadas hasta el momento; por ese motivo no se van a contemplar en este apartado.

A continuación, se va a mostrar cómo quedaron finalmente definidas las historias de usuario para la primera parte del desarrollo, es decir, la parte en la que se ha aplicado Scrum) :

9990 - **Montaje del Entorno:** los desarrolladores quieren un entorno de desarrollo en donde poder trabajar.

9690 - **Intérprete de Red de Petri en Java:** como operador del aeropuerto, quiero poder definir una Red de Petri que defina la lógica del AMS, tal que pueda implementar diferentes estrategias de operación.

9950 - **Analizador sintáctico de expresiones:** Como desarrollador quiero un Analizador sintáctico de expresiones que lleguen en forma de cadena y sea capaz de evaluar las condiciones que están definidas en ella.

9740 - **Framework de ejecución de la Red de Petri:** Como operador, quiero que la Red de Petri definida se ejecute paso a paso a una frecuencia (Hz) determinada, enviando las señales de salida de los *Lugares* que estén activados, comprobando el estado de las *Transiciones* mediante la evaluación de una

expresión y cambiando el estado de la Red de Petri si se cumplen las condiciones.

9730 - **Configurador Red de Petri:** como operador, quiero poder definir la Red de Petri en un XML sencillo y con él obtener una Red de Petri funcional.

9720 - **Integración de DDS RTI en en proyecto:** los desarrolladores necesitan disponer de las librerías de DDS RTI para poder recibir y enviar mensajes por la red.

9690 - **Crear una capa de comunicación:** esta capa es necesaria para enlazar la Red de Petri con el [word]gMw de comunicación DDS RTI.

7.3. Tareas

Una vez definida toda la lógica necesaria para hacer funcionar una Red de Petri, el siguiente paso es mostrar y hacer cambiar su estado en una Aplicación Web. Para esta parte se deja Scrum a un lado y se procede al desarrollo con Kanban.

Aunque se use esta metodología para gestionar la carga de trabajo, no se hará demasiado hincapié en la entrega *justo a tiempo*, ya que como sabemos, es un proyecto de investigación, y no se dispondrá de un cliente esperando resultados.

El *trabajo en curso* se dividirá en tareas, añadiéndose a la pizarra conforme vayan surgiendo nuevas necesidades en el desarrollo o se acaben las ya planificadas. Sin embargo sólo se podrá desarrollar una de ellas cada vez, y pasar a otra tarea cuando se ha completado la anterior.

Se muestran algunas de las tareas desarrolladas con esta metodología:

- 1.-** Adaptar el entorno de trabajo para el desarrollo *Web Apps*.
- 2.-** Crear vista principal.
- 3.-** Crear vista del aeropuerto.
- 4.-** Gestionar horas de vuelo.
- 5.-** Crear vista de controles extendidos.
- 6.-** Conectar la Red de Petri con las vistas.
- 7.-** Visualizar correctamente las señales recibidas y enviadas.
- 8.-** Adaptar las vistas para el funcionamiento con un sólo UAV.
- 9.-** Apadtar la vista para un obstáculo.
- 10.-** Poder inicializar y parar la Red de Petri cuando sea necesario.
- 11.-** Adaptar las vistas para el funcionamiento con N UAVs.

Capítulo 8. Diseño e Implementación

Una vez estudiadas las tecnologías y metodologías usadas en el proyecto, pasamos a definir cómo está construido y donde utilizaremos cada una de ellas.

El objetivo principal de este proyecto es visualizar el estado actual de una Red de Petri y poder interactuar con ella a través de una Aplicación Web. Estas son las bases en las que se asienta este proyecto.

Por un lado, utilizando Scrum[17], se desarrollará la capa de Lógica de Negocio, donde se implementa una Red de Petri enlazada con un paresador de XML. Ésta recibe y envía señales por la red mediante DDS RTI, a través de una plataforma externa desarrollada por la University of Duisburg-Essen (UDE). A continuación se detalla la Red de Petri que se implementará:

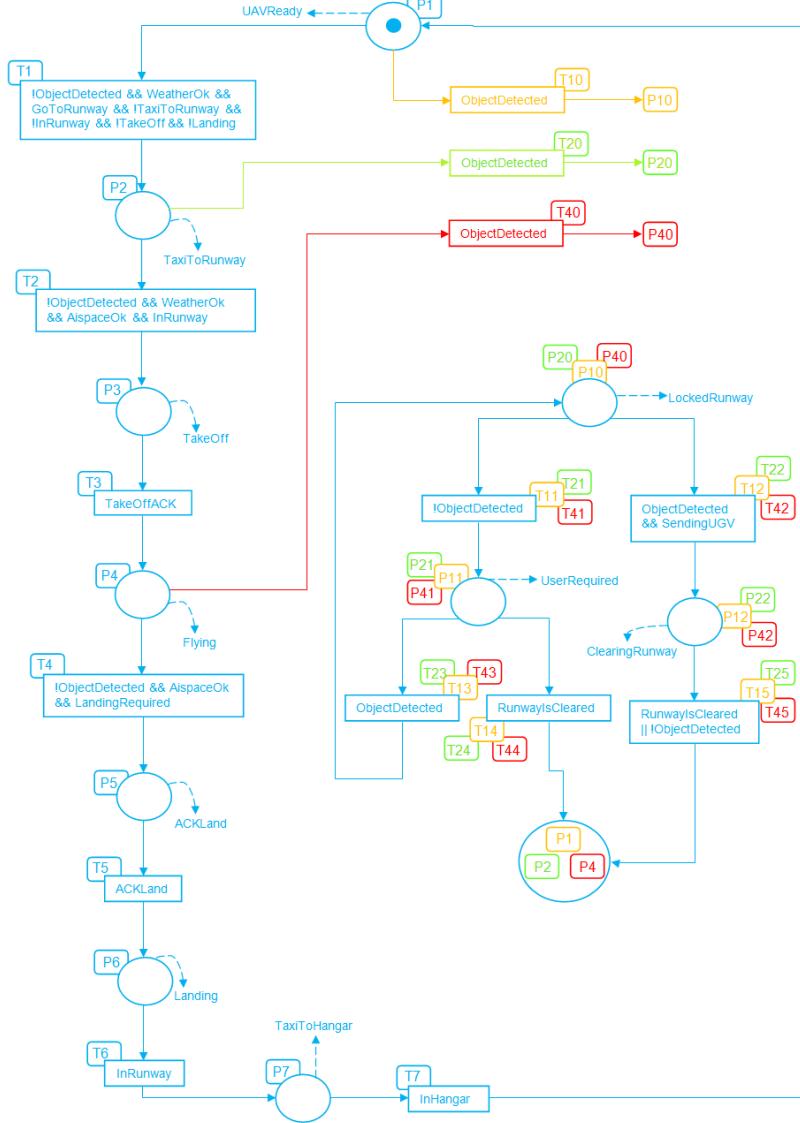


Figura 8.1: Red de Petri a implementar.

Por otro lado, utilizando Kanban esta vez, se utilizará el Patrón de Diseño M.V.C., para desarrollar la Aplicación Web. Este Patrón, como se ha estudiado anteriormente, está sostenido en tres pilares, cada uno igual de importante que el otro. En el **Modelo** quedará implementada la Lógica de Negocio, que

estará desarrollada íntegramente en Java. El **Controlador**, que responde a los eventos del usuario en esta, estará implementado con Struts2 y Spring. La **Vista** por su parte, será representada con JSPs^[46].

Como este proyecto es de ámbito internacional, todas las clases y variables del código, así como los comentarios, estarán escritos únicamente en inglés, siendo además esta medida una ayuda para evitar tener un código escrito en *Spanglish*, apostando por un único idioma para su mejor comprensión y evitar pérdida de información o malentendidos.

8.1. Montaje del entorno

Éste es el primer y obligatorio paso a dar a la hora de crear un proyecto. Anteriormente se han estudiado las tecnologías a usar en este proyecto (véase Capítulo 5).

Para el desarrollo en Java del proyecto en **Ubuntu**^[52] como sistema operativo principal, se ha usado OpenJDK^[53], versión libre de la plataforma Java e implementación por defecto instalado en el S.O. citado.

Se necesitará un Entorno de Desarrollo Integrado que soporte el resto de tecnologías Web que necesitamos, para ello usaremos eclipse^[51], un IDE con soporte para Java y desarrollo de Web Apps, entre otras muchas cosas. Además, dispone de un plugin para trabajar con *Maven* denominado m2e^[42]. De la misma forma, también será necesario instalar el plugin para trabajar con JUnit.

Por último, para mantener el control de versiones, se utilizará ^[35], y como Interfaz Gráfica de Usuario, *TortoiseHg*^[37], que enlaza con un servicio de alojamiento de código en la nube con *Bitbucket*^[38]. Para más información, volver a la Sección 6.

8.1.1.Historia de usuario y Diagrama de BurnDown

Con la instalación de todos los elementos necesarios para empezar a desarrollar, se completa la primera historia descrita:

9990 - Montaje del Entorno.

En este diagrama de *BurnDown* inicial, podemos apreciar cómo ha afectado el cambio de las Historias de Usuario iniciales, al incluir el tratamiento de señales por DDS RTI en el desarrollo y la división de varias historias de usuario:

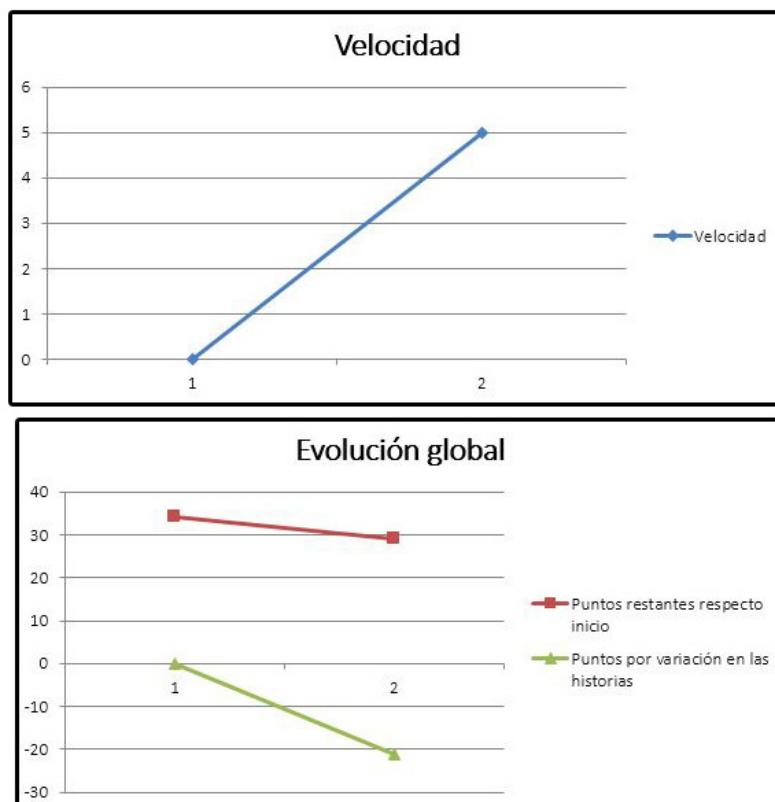


Figura 8.2: Diagrama de BurnDown para Sprint 2.

Este diagrama pertenece al *Sprint* dos, ya que como se puede ver, en el primero no se completó ninguna historia de usuario, esto es así porque se dedicó precisamente a organizarlas y preparar conocimientos a aplicar a partir del segundo *Sprint*.

8.2. Desarrollo de la Red de Petri

Para representar los distintos posibles estados del aeropuerto y poder usar un sistema de envío y recepción de señales que encaje perfectamente con la representación de cada uno de los estados, se ha considerado desarrollar una Red de Petri (véase Sección 5.4).

En primer lugar se definen los tipos más básicos de una Red de Petri, los objetos Transición (*Transition*) y Lugar (*Place*), los cuales tienen una Id única que los identifica.

8.2.1. Objeto Transition

Una *Transition* contiene una lista de *Places* de entrada y otra de *Places* de salida. Para facilitar la construcción del grafo de la Red de Petri se define un *enumerado* con dos estados, *IN* o *OUT*. Al agregar un *Place* a una de las transiciones, se especificará su *arquetipo enumerado*, y se añadirá a la lista correspondiente. De la misma forma, se podrá eliminar un *Place*, según su *arquetipo enumerado* y obtener la lista de *Places* de entrada o la de *Places* de salida.

Una *Transition*, por su naturaleza, puede estar habilitada o inhabilitada, por lo tanto dispondrá de los métodos necesarios para habilitarla, deshabilitarla y obtener su estado actual. Así pues, también se podrá saber si está sensibilizada o no, esto es, cuando todos sus *Places* de entrada están marcados.

8.2.2. Objeto Place

Un *Place* tiene un número limitado de marcas, por defecto es una, pero se puede consultar y modificar el número máximo de marcas que puede tener un *Place*, así como marcar y desmarcar un *Place* determinado. Nunca se podrá marcar más del número máximo de marcas asignado.

8.2.3. Objeto PetriNet

Un objeto PetriNet almacenará un conjunto de *Transitions*, las cuales se almacenarán como un conjunto clave-valor, donde la clave será la *Id* de la transición y el valor el objeto *Transition* en sí mismo. De la misma forma también dispondrá de un conjunto clave-valor para los *Places*. A parte de esta funcionalidad básica, podremos obtener una lista de los *Places* que están marcados y una lista de las *Transitions* habilitadas. Para hacer funcionar la PetriNet hay dos métodos importantes que desarrollar, los cuales son claves para su funcionamiento:

```
firetransition(T transitionId)
```

Este método dispara la *Transition* indicada, comprobando primero que ésta está sensibilizada y habilitada. Recordemos que esto sucede cuando su expresión asociada se cumple. Al dispararse la *Transition*, los siguientes *Place-OUT* de la *Transition* quedan marcados, desmarcando a su vez los *Place-IN* marcados en el paso anterior. Por último, al desactivar un *Place*, queda desactivada también su señal asociada mediante el objeto *Manejador de señales*, que se verá más adelante. Veamos su definición:

```
public boolean fireTransition(T transitionId) throws  
OverflowException {  
    ITransition<T> transitionTofire = transitions.get(  
        transitionId);  
  
    if (transitionTofire.isSensitized() && transitionTofire.  
        isEnabled()) {
```

```

        List<IPlace<T>> placeList = transitionTofire.
            getPlaces(ArcType.OUT);
        for (IPlace<T> place : placeList) {
            place.mark();
            log.debug("Marking the place " + place.getId());
        }
        placeList = transitionTofire.getPlaces(ArcType.IN);
        for (IPlace<T> place : placeList) {
            place.unmark();
            signalManager.deactivateSignalFromPlace(
                place.getId());
            log.debug("Unmarking the place " + place.
                getId());
        }
        return true;
    }
    return false;
}

```

nextStep()

Este otro método es el que se encarga de gestionar cada paso en la PetriNet. En primer lugar, guarda el estado actual de las señales, para evitar fallos de concurrencia, y envía las señales de cada *Place* marcado mediante el *Manejador de señales*. Seguidamente, comprueba todas las *Transition* mediante el objeto *PetriNetTransitionManager*, y obtiene las *Transitions* habilitadas en ese momento. Para cada una de ellas, llama al método *firetransition(T transitionId)*, que comprobará si se puede disparar o no. Si se ha podido ejecutar el franqueo de alguna *Transition*, el método devolverá **true**, indicando que la *PetriNet* ha pasado a un nuevo estado.

```

public boolean nextStep() {
    boolean result = false;
    signalManager.saveSignals();
    try {
        for (IPlace<T> place : this.getMarkedPlaces()) {
            signalManager.sendSignalsFromPlace(place.
                getId());
        }

        transitionManager.checkAllTransitionsExpressions();

        for (ITransition<T> transition : transitionManager.
            getEnabledTransitions()) {

            result = fireTransition(transition.getId());
            if (result) {
                log.debug("The transition " +
                    transition.getId() + " has been
                    fired succesfully");
            }
        }
    }
}

```

```
                break;
            }
        }

    } catch (EmptyException | InvocationMethodException |
OverflowException e) {
    log.error("Error during the execution of the
              iteration: " + e.getMessage());
    result = false;
}
return result;
}
```

8.2.4. Objeto PetriNetTransitionManager

Este objeto se encarga de otra importante tarea en una *PetriNet*, comprobar qué *Transition* puede activarse y desactivarse, pudiendo obtener en cada momento una lista con las *Transition* habilitadas y otra con las deshabilitadas.

Para llevar a cabo esta tarea, dispondrá de un objeto *PetriNetTransitionMatcher*, que veremos más adelante, y que nos facilita las relaciones de las *Transition* con sus respectivas expresiones. Para cada expresión, se hará uso del objeto *ExpressionParser*, que también se estudiará más adelante, cuya función es evaluar cada una de ellas y devolver cierto o falso como resultado. Si la evaluación de la expresión ha resultado positiva, se añadirá a la lista de *Transition* habilitadas, o a la lista de las inhabilitadas en caso contrario, eliminado también a la viceversa en cada caso.

```
public void checkAllTransitionsExpressions() {
    transitionsMap = petriNetTransitionMatcher.getPairs();
    boolean parsedResult;

    for (ITransition<T> transition : transitionsMap.keySet()) {
        String condition = transitionsMap.get(transition);
        try {
            parsedResult = parser.parse(condition);
        } catch (InvocationMethodException e) {
            parsedResult = false;
        }
        if (parsedResult) {
            transition.enable();

            enabledTransitions.add(transition);
        } else {
            transition.disable();

            disabledTransitions.add(transition);
        }
    }
}
```

```
        if (disabledTransitions.contains(transition)
            )
            disabledTransitions.remove(
                disabledTransitions.indexOf(
                    transition));
    } else {
        transition.disable();

        disabledTransitions.add(transition);
        if (enabledTransitions.contains(transition))
        {
            enabledTransitions.remove(
                enabledTransitions.indexOf(
                    transition));
        }
    }
}
```

8.2.5. Objeto PetriNetTransitionMatcher

Este objeto es el encargado de enlazar las *Transition* con sus respectivas expresiones. Para ello dispondrá de una colección clave-valor, donde a cada clave *Transition* le será asignada un valor expresión. En él podremos agregar un conjunto *Transition*-expresión o eliminarlo, pasando la *Transition* como clave. De la misma forma, podremos obtener la expresión enlazada a una *Transition* u obtener una colección con todas las uniones realizadas.

Cabe destacar que, para crear eliminar u obtener una unión *Transition*-expresión, el objeto *PetriNetTransitionMatcher* comprobará debidamente que ninguno de los participantes es nulo o vacío, de lo contrario lanzará una excepción, evitando así un mal funcionamiento de la Red de Petri. Además, a la hora de agregar una expresión, se hará uso del objeto *RegularExpressionValidator*, que comprobará adecuadamente si la expresión asignada está bien formada, para que pueda ser interpretada correctamente por nuestro programa.

8.2.6. Objeto PetriNetGenerator

Para repartir mejor las dependencias, se necesitará una clase generadora de PetriNet, que obtenga los objetos necesarios y con ellos forme la PetriNet, ya que crear un objeto de esta clase no es tarea sencilla.

En este objeto, como padre de la PetriNet, se podrá:

- **Añadir:** un objeto Place, un objeto Transition, una relación de Place-Transition-Arquetipo, una o varias marcas a un objeto Place, una relación Place-Signal o una relación Transition-Expression.
- **Añadir un conjunto de:** relaciones de Places-Transition-Arquetipo, objetos Place a los que añadir una marca o de señales asignadas a un Place.
- **Eliminar:** un objeto Place, un objeto Transition o una relación Place-Transition-Arquetipo
- **Modificar:** el número máximo de marcas para un Place.
- **Obtener un conjunto de:** objetos Place, objetos Transition o de señales relacionadas a un Place.
- **Obtener:** una expresión relacionada a una Transition y cómo no, una PetriNet completamente formada.

Casuística a tener en cuenta: A la hora de añadir una expresión a una *Transition*, es importante y necesario reemplazar en la expresión los caracteres *AND* por *&&* y *OR* por *||*. Esto es así debido a que XML no ve con buenos ojos los caracteres *&* y *|*, y para las expresiones, como veremos más adelante, estos caracteres son indispensables para su evaluación.

Por otra parte, cuando añadimos un *Place*, una *Transition* o una relación *Place- Transition*, se comprueban debidamente si ya existían anteriormente, si es así no podrán añadirse, ya que no pueden existir dos *Place* o dos *Transition* con la misma Id.

Finalmente, antes de devolver un objeto *PetriNet* completo, el objeto *PetriNetGenerator* debe comprobar que está correctamente definida. Para ello comproba que cada *Transition* tiene asignado al menos un *Place* de entrada y otro de salida, y que cada uno de estos *Places* estan debidamente definidos en el objeto *PetriNetGenerator*. Si todo es correcto, invoca al objeto *PetriNet* y le pasa cada uno de los elementos necesarios para su creación.

```

public IPetriNet<T> getPetrinet() {

    PetriNet<T> petrinet = new PetriNet<T>(new
        PetriNetTransitionManager<T>(this.matcher, signals) {
    }, signalsManager);
    if (!validatePetriNet())
        return null;

    petrinet.setTransitions(transitions);
    petrinet.setPlaces(places);

    return petrinet;
}

/**
 * This method checks if actual PetriNet is built correctly.
 *
 * @return True if the PetriNet is right, false if not.
 */
private boolean validatePetriNet() {
    List<IPlace<T>> placesWithTransitions = new ArrayList<>();

    for (ITransition<T> transition : transitions.values()) {
        if (transition.getPlaces(ArcType.IN) == null ||
            transition.getPlaces(ArcType.IN).size() <= 0 ||
            transition.getPlaces(ArcType.OUT) == null ||
            transition.getPlaces(ArcType.OUT).size() <= 0)
            return false;
        placesWithTransitions.addAll(transition.getPlaces(
            ArcType.IN));
        placesWithTransitions.addAll(transition.getPlaces(
            ArcType.OUT));
    }

    for (IPlace<T> place : places.values()) {
        if (!placesWithTransitions.contains(place))
            return false;
    }
    return true;
}

```

8.2.7. Objeto PetriNetFramework

Una Red de Petri no es un objeto estático, todo lo contrario, es dinámico, y cambia con el paso del tiempo, y gracias a la interactuación de las señales. Es por esto que debe mantener un hilo siempre funcionando en background.

Este Objeto, básicamente, extiende las funciones de *Thread*, adecuándolas a nuestro propósito. Contiene un método *run()* que mantiene el hilo vivo, podríamos decir que es como *el corazón* de la PetriNet, funciona a una frecuencia establecida mientras no se le indique que pare, con el método *stop()*. La frecuencia se establece al crear el objeto, y se puede cambiar con el método *setFrequency(int hertz)*. Éste valor establece la frecuencia, en hercios, a la que se llama el método *nextStep()* de la propia PetriNet, que ejecuta, como hemos visto anteriormente, las operaciones correspondientes, sensibilizando transiciones y enviando señales cuando sea posible.

8.2.8. Historias de usuario y pruebas realizadas

Para completar este desarrollo, se han necesitado los *Sprints* que van del tercero al quinto, aunque en el cuarto se ha pausado para realizar el Analizador sintáctico necesario para completar el funcionamiento de la Red de Petri. Por este motivo, el Diagrama de Burndown de estos tres *Sprints* se verá en el siguiente apartado donde se podrá apreciar mejor el resultado de una manera global.

En el segundo *Sprint* se empieza a desarrollar la Historia de usuario:

9690 – Intérprete de Red de Petri en Java.

Algunas de las Pruebas Unitarias desarrolladas para la Red de Petri son:

- Poder instanciar correctamente un objeto *Place*.
- Poder instanciar correctamente un objeto *Transition*.
- Definir una Red de Petri en el formato acordado.
- Las condiciones de disparo de cada *Transition* se pueden definir con operaciones lógicas de diferentes parámetros.
- Acceder a los diferentes parámetros de los objetos creados.

En el cuarto *Sprint* se completó la Historia de usuario:

9740 – Framework de ejecución de Red de Petri en Java.

Algunas de las Pruebas Unitarias desarrolladas para el Objeto *PetriNetFramework* son:

- El *Framework* debe poder configurar los *Hz* a los que se quiere refrescar.
- Se debe de poder pasar una Red de Petri funcional que es la que se va a ejecutar.
- Debe de enviar las señales de los *Places* activos a quien corresponda.
- Debe de evaluar la expresión de las transiciones sensibilizadas (sus *Places* están activos), para determinar si se puede o no lanzar.
- Si una *Transition* es lanzada, se debe cambiar el estado de la máquina al

siguiente paso.

- Se desea una prueba de integración que compruebe todo esto, imprimiendo el estado de la Red de Petri en cada refresco y permitiendo entrada de valores del teclado que cambien el estado de la máquina.

Por último, como Pruebas de Integración, se asegura que la Red de Petri funciona correctamente, integrando todos los componentes desarrollados y manejados desde el objeto *PetriNetFramework*. Esta prueba es crítica, ya que integra por primera vez los componentes de la Red de Petri.

8.3. Intérprete de expresiones

Cada objeto *Transition* tiene ligado a él una expresión regular que, tras ser evaluada a cada paso de la *PetriNet*, la habilita o deshabilita convenientemente.

8.3.1. Expresiones regulares

Una expresión regular es una secuencia de caracteres que forma un patrón de búsqueda, principalmente utilizada para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones. Por ejemplo, las cadenas Casa, Cosa y Cesa se describen mediante el patrón "C(a|o|e)s". El uso de éstas nos provee una manera muy flexible de buscar o reconocer cadenas de texto, por lo que es la solución perfecta para nuestro propósito.

Las expresiones regulares se rigen por una serie de normas, y hay una construcción para cualquier patrón de caracteres. Pueden contener, aparte de letras y números, varios caracteres especiales, que agrupan y cuantifican las expresiones, los cuales veremos en las siguientes tablas:

Exp. Regular	Descripción de la expresión regular
.	Concuerda con cualquier carácter, 1 o más concordancias.
^	Concuerda en el principio de la línea.
\$	Concuerda en el final de la línea.
[]	Conjunto de caracteres.
[-]	Rango de caracteres.
[^]	Concuerda con cualquier carácter excepto el conjunto de caracteres contenidos en el conjunto.
()	Subexpresión o grupo.
	Permite una alternativa para elegir entre dos expresiones.

Tabla 8.1: Símbolos comunes de concordancia.

Los siguientes metacaracteres tienen un significado predefinido, y hacen que crear algunos de los patrones más comunes sean más fáciles de usar:

Exp. Regular	Descripción de la expresión regular
\d	Concuerda con cualquier número, entre 0 y 9.
\D	Concuerda con cualquier carácter que no sea un número.
\s	Concuerda con cualquier carácter de espaciado, [\t\n\r\f].
\S	Concuerda con cualquier carácter que no sea de espaciado.
\w	Rango de caracteres que forma una palabra, [a-zA-Z_0-9].
\W	Concuerda con cualquier carácter que no sea una palabra.
\S+	Concuerda con varios caracteres de espaciado consecutivos.

Tabla 8.2: Conjunto de metacaracteres disponibles.

Los cuantificadores definen con qué frecuencia puede ocurrir un elemento para cierto patrón.

Exp. Regular	Descripción de la expresión regular
*	Concuerda con cualquier carácter, 0 o más veces.
+	Concuerda con cualquier carácter, 1 o más veces.
?	Concuerda con cualquier carácter, 1 o ninguna vez.
{X}	Concuerda con cualquier carácter X veces.
{X,Y}	Concuerda con cualquier carácter entre X e Y veces.
*?	El símbolo ? después de un cuantificador hace que sea un cuantificador reticente, trata de encontrar la combinación más pequeña.
	Permite una alternativa para elegir entre dos expresiones.

Tabla 8.3: Conjunto de cuantificadores disponibles.

8.3.2. Construcción de expresiones regulares en Java

Para construir y validar expresiones regulares, en Java[20] (véase Sección 5.6), disponemos del paquete `java.util.regex`[50]. Aunque el objeto `String` nos proporciona métodos como `cadena.matches("regex")`, que comprueba si cadena concuerda con la cadena "regex" entera, o `cadena.split("regex")`, que crea un array dividiendo cadena en dos cada vez que concuerda "regex", o `cadena.replace("regex", "replacement")`, que sustituye en cadena la palabra "regex" por "replacement", pero todo esto carece de la potencia y facilidades que nos ofrece el citado paquete.

Para trabajar con expresiones regulares, hay que construir primero un objeto `Pattern`, donde se define la expresión regular en sí llamando a su método `compile("regex")`, obteniendo una estancia del mismo, con la expresión regular asignada. Este objeto, a su vez, no permite obtener otro del tipo `Matcher`, llamando al método `matcher("cadena")`. En su interior, el `Matcher` comprueba si la expresión regular creada en el `Pattern` y la cadena pasada concuerdan, para ello llamamos al método `matches()`, que devuelve `true` en caso de concordancia o `false` en caso contrario.

A continuación, se muestra un ejemplo sencillo para afianzar conocimientos. En este caso se comprobará que la *expression* pasada al *Matcher* contiene xx en su interior, entre al menos, una letra a cada lado, además incluiremos al final un ejemplo de un patrón mal expresado:

```
public static void main(String[] args) {
    Pattern pattern = Pattern.compile(".xx.");
    Matcher matcher = pattern.matcher("MxxY");
    System.out.println("Input String matches regex - "+matcher.matches());

    // Expresión regular mal formada:
    //pattern = Pattern.compile("*xx*");
    // esta expresión permite dejar las xx solas, y que remos
    // que estén rodeadas

}
```

Tanto *Pattern* como *Matcher* y el paquete entero de *regex*[50], contienen mucha más funcionalidad de la descrita, pero para cumplir los objetivos del proyecto, estos son los conocimientos necesarios.

8.3.3. Objeto RegularExpressionValidator

Las expresiones ligadas a las *Transitions* son cadenas de texto, estas son sus características:

- Una expresión válida podrá contener un nombre, al que se le llamará "nombre clave", compuesto por letras y números.
- Este "nombre clave" podrá estar negado o no, usando para la negación el símbolo !.
- Cada "nombre clave" podrá tener contiguo una pareja de caracteres & y |, seguidos de otro «nombre clave», repitiéndose cuanto sea necesario.

- Cada expresión puede estar formada a su vez de otras subexpresiones, que podrán aislar entre ellas usando paréntesis ().

A continuación se verá cómo se han implementado estas condiciones usando expresiones regulares:

```

        // and finally it ends with
        // another parentheses.
        "(((\&&|\\"|\\"|)\\"s*!{0,1}\\"s
         *\\"(\{0,1}(\\"s*!{0,1}(\\"s
         *([a-zA-Z]+\\"d{0,3}?)\\"s
         *)\{1,\})?)\\"{0,1}\\"s*)"
        + "|\(=\\"s*!{0,1}\\"s
         *\\"(\{0,1}(\\"s*!{0,1}(\\"s
         *([a-zA-Z]+\\"d{0,3}?)\\"s
         *)\{1,\})?)\\"{0,1}\\"s*))*"
        );
    }

    Pattern patternParentheses = Pattern.compile("([^\\"(\\")]*(\\"s*\\"s*[^\\\"(\\")]*\\"s*\\"))*)");

    Pattern patternNoDelimiters = Pattern.compile("[a-zA-Z]+\\"d
        {0,3}\\"s+[a-zA-Z]+\\"d{0,3}");

    Matcher matcherExpressionContent = patternExpressionContent.
        matcher(expression);
    Matcher matcherEquals = patternEquals.matcher(expression);
    Matcher matcherParentheses = patternParentheses.matcher(
        expression);
    Matcher matcherNoDelimiters = patternNoDelimiters.matcher(
        expression);

    if (matcherEquals.matches()) {
        return false;
    } else {
        if (!matcherParentheses.matches()) {
            return false;
        } else {
            if (matcherNoDelimiters.matches()) {
                return false;
            } else {
                return matcherExpressionContent.
                    matches();
            }
        }
    }
}

```

8.3.4. Objeto ExpressionParser

Este objeto es el encargado de recibir las expresiones desde la Red de Petri y validar cada una de ellas. Es un objeto muy simple y complejo a la vez. Simple, porque sólo tiene un método público, *parse(String expression)*, que evalúa la expresión y devuelve su resultado. Complejo debido a que, aunque sólo disponga

de un método de cara al público, por dentro realiza múltiples operaciones.

Su cometido es ir dividiendo las expresiones conforme a los delimitadores permitidos, hasta obtener el nombre de las señales por las que se pregunta. Una vez obtenido el nombre de la señal, se preguntará por su estado al objeto Signal-Handler. A continuación, se procederá a la evaluación de la expresión conforme a cada uno de los delimitadores encontrados en ellas. Finalmente, obtenemos el resultado final de la expresión.

A modo de ilustración a lo anteriormente explicado, se mostrará la implementación de cómo divide la expresión en cada situación, invocando al método correspondiente para que haga el tratamiento adecuado:

```
public boolean parse(String expression) throws InvocationMethodException {
    boolean result = false;

    if (expression.contains("||")) {
        String terms[] = expression.split("\\|\\|");
        List<Object> objectsToCompare = invokeMethods(terms);
        result = compareIfOrsTerms(objectsToCompare);
    } else {
        if (expression.contains("(") && expression.contains(")")) {
            String terms[] = expression.split("\\(");
            result = invokeParenthesesMethods(terms);
        } else {
            if (expression.contains("&&")) {
                String terms[] = expression.split("&&");
                List<Object> objectsToCompare = invokeMethods(terms)
                    ;
                result = compareIfAmpersandsTerms(objectsToCompare);
            } else {
                if (expression.contains("=")) {
                    String terms[] = expression.split("=");
                    List<Object> objectsToCompare = invokeMethods
                        (terms);
                    result = compareIfEqualsTerms(
                        objectsToCompare);
                } else {
                    if (expression.contains("!")) {
                        String terms[] = expression.split("!");
                        ;
                        List<Object> objectsToCompare =
                            invokeNegationMethods(terms);
                        result = compareIfNegateTerms(
                            objectsToCompare);
                    } else {
                        result = (Boolean) invokeMethod(
                            expression);
                    }
                }
            }
        }
    }
}
```

```
        }
    }
    return result;
}
```

Como se puede observar, tras cada división se hace una llamada a `getSignals()`, que obtendrá las señales pertinentes y las tratará conforme la operación que lleven entre ellas.

8.3.5. Historias de usuario y Pruebas realizadas

En el cuarto *Sprint* la funcionalidad desarrollada queda a expensas de terminar el desarrollo de la Red de Petri en el próximo *Sprint*. Como hemos comentado en secciones anteriores, será en los resultados del cuarto sprint donde se obtenga una perspectiva global del desarrollo de la primera parte de la Red de Petri.

Las historia iniciada es:

9690 – Intérprete de expresiones para la Red de Petri.

Algunas de las Pruebas Unitarias desarrolladas para el Intérprete de Expresiones son:

- Analizar las expresiones lógicas con *AND* (`&&`).
- Analizar las expresiones lógicas con *OR* (`||`).
- Analizar las expresiones lógicas con *NOT* (`!`).

- Analizar las expresiones lógicas con *EQUALS* (=).
- Analizar las expresiones lógicas con uso de paréntesis.
- Las condiciones de disparo de cada *Transition* se pueden definir con operaciones lógicas de diferentes parámetros.
- Evaluar expresiones del tipo: "weatherCondition"="Sunny" && "takeoff" y devolver si se cumple o no.

8.3.6. Diagrama de BurnDown

En el diagrama de *BurnDown* tras el quinto *Sprint* de desarrollo se puede apreciar cómo en el *Sprint* apenas se avanza, ya que aunque la Red de Petri está siendo definida, sin el intérprete de expresiones, no se puede probar apenas su funcionalidad. Lo mismo pasa en el siguiente *Sprint*, donde se desarrolla sólo el intérprete de expresiones, sin probar aún más funcionalidad con la Red de Petri:

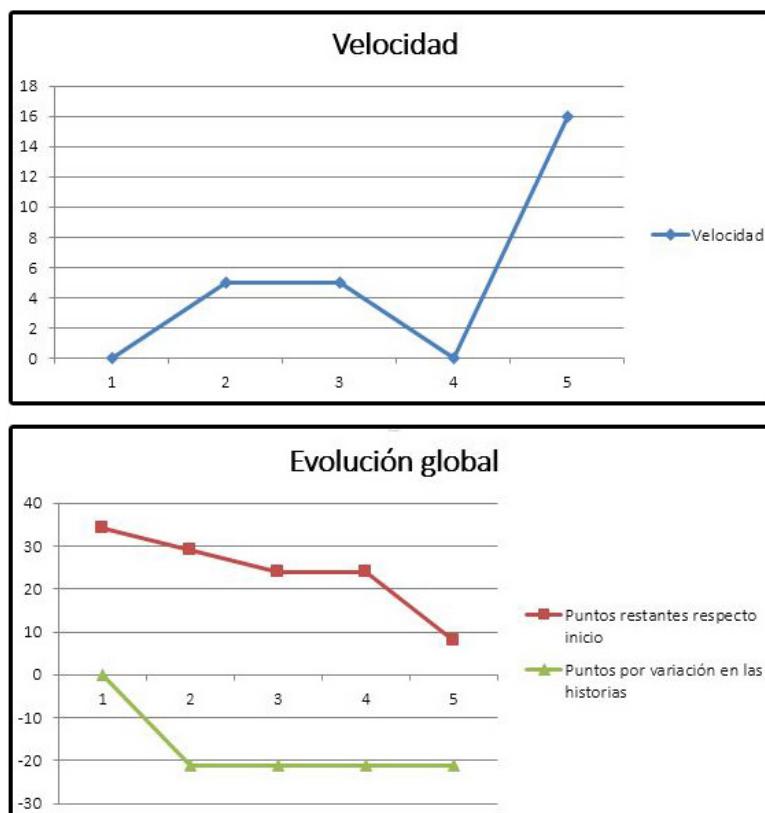


Figura 8.3: Diagrama de BurnDown para Sprint 5.

Finalmente es en el cuarto *Sprint* cuando todo queda integrado y las historias de usuario que no estaban completas y quedaron pendientes de anteriores sprints quedan superadas. Esto enseña que había un fuerte acomplamiento entre historias de usuario, cuestión que debe mejorarse en futuras iteraciones.

8.4. Parseador de XML

El parseador de XML es necesario para dotar a la Red de Petri de facilidad de creación y modificación, ya que nos permite definir en un archivo estructurado cada uno de sus componentes.

Fucionamiento

Para la lectura del archivo XML haremos uso de las librerías de XStream. Estas nos proporciona los métodos necesarios para cargar y leer el archivo XML. Una vez cargado, XStream[54] transforma la información leída a objetos que han sido diseñados con una estructura similar a la especificada en el XML.

Se muestra un ejemplo de XML generador de Red de Petri:

```
<?xml version="1.0" encoding="UTF-8"?>

<petrinet>
    <transitions>
        <transition idTransition="One" expression="BUENTIEMPO AND
PISTADESPEJADA AND DESPEGA" />
        <transition idTransition="Two" expression="BUENTIEMPO AND
QUIEREATERRIZAR" />
    </transitions>
    <places>
        <place idPlace="One" signal="DESPEGA"/>
        <place idPlace="Two" signal="ATERRIZA"/>
    </places>
    <relations>
        <relation transitionID="One" placeID="One" type="IN"/>
        <relation transitionID="One" placeID="Two" type="OUT"/>
        <relation transitionID="Two" placeID="One" type="OUT"/>
        <relation transitionID="Two" placeID="Two" type="IN"/>
    </relations>
    <initialStatus>
        <placeStatus placeID="One" maxMarks="3" marks="1"/>
    </initialStatus>
</petrinet>
```

Usando XStream[54], esta especificación crea un objeto *petrinet*, que contiene un objeto *transitions*, un objeto *places* y un objeto *initialStatus*. A su vez, el objeto *transitions* contiene dos objetos *transition* distintos, que tienen una *id* y una expresión asignada; el objeto *places* contiene dos objetos *place*, que tienen una *id* y una señal asignada; y el objeto *initialStatus* contiene un sólo objeto *placeStatus* que tiene una *id*, un número máximo de marcas y un número de marcas por defecto.

Teniendo los objetos creados a partir del XML, sólo necesitamos de un objeto *PetriNetXMLTransformer* que construirá cada uno de los elementos de

la Red de Petri ayudado por el *PetriNetGenerator* estudiado anteriormente. Una vez que el objeto *PetriNetGenerator* ha obtenido todos los datos necesarios, se obtiene una Red de Petri completamente funcional lista para ser inicializada por el objeto *PetriNetFramework*.

8.4.1. Historias de usuario y Pruebas realizadas

Para completar este desarrollo, se ha necesitado un sólo *Sprint*, el sexto, añadiendo la funcionalidad a la ya desarrollada Red de Petri en los *Sprints* anteriores.

La historia completada es:

9690 - Configurador Red de Petri mediante un XML.

Algunas de las Pruebas Unitarias desarrolladas para el Intérprete de Expresiones son:

- Poder definir los nodos tipo *Place*, sus señales y sus *Transitions* tanto de entrada como de salida.
- Poder definir los nodos tipo *Transitions* y sus expresiones de activación.
- Poder definir el estado inicial de la Red de Petri, indicando qué lugares tienen activos y cuál es el valor inicial de los valores de entrada para las expresiones.
- Las condiciones de disparo de cada *Transition* se pueden definir con operaciones lógicas de diferentes parámetros.

- Si una Red de Petri no está definida de manera congruente, debe fallar: hay *Places* sin *Transitions*, no hay un *Place* marcado inicialmente, hay *Transitions* que no tienen expresiones, hay *Transitions* huérfanas, etc.
- Si no se definen los valores de entrada de las variables usadas en las expresiones de las *Transitions*, se ponen los valores por defecto dependiendo del tipo.
- El resultado debe devolver una Red de Petri funcional y lista para usar Poder definir la Red de Petri en un XML sencillo.

Por último, como Pruebas de Integración, se requiere una prueba que genere una Red de Petri a partir de un XML dado. Y también es necesario probar en los test de integración que lo desarrollado en la historia de usuario **9740** funcione junto con lo desarrollado en este sprint, por lo que se cargará la Red de Petri de un XML y se ejecutará, iteración a iteración, el Marco de Trabajo de la Red de Petri, mostrándose el resultado en pantalla.

8.4.2. Diagrama de BurnDown

Estado del diagrama de *BurnDown* tras el sexto *Sprint* de desarrollo:

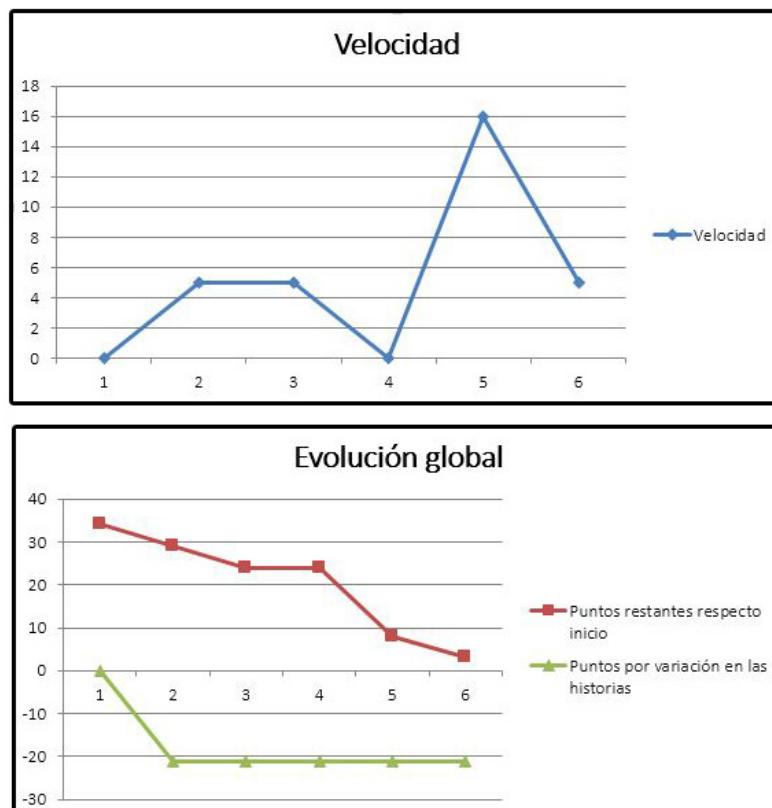


Figura 8.4: Diagrama de BurnDown para Sprint 6.

8.5. Señales y DDS

Cada objeto *Place* en la Red de Petri desarrollada tiene asignada una señal, que se deberá activar y publicar si está dicho *Place* marcado, o desactivar y dejar de publicar si está desmarcado. Estas señales, a su vez, son consultadas en cada iteración por el objeto *PetriNetTransitionManager*, que disparará o no las *Transitions* en función de sus expresiones y el estado de las señales.

Las señales van cambiando con el tiempo, ya sea por la actuación de la Red de Petri o por la recepción de la misma a través de la red local. Para ello se

utilizarán las librerías DDS RTI, que nos permitirá tanto recibir como publicar un mensaje asignado a una señal.

8.5.1.DDS Middleware

Anteriormente se ha estudiado con detenimiento DDS RTI (véase la Sección 5.8), para el manejo de señales será necesario hacer uso de él mediante el Middleware de PLANET.

Para poder usar el Middleware será necesario instalar software DDS RTI desde de la web de RTI^[8]. Seguidamente, se procederá a instalar la versión 2.4.1 de Google Protocol Buffer, ya que versiones más actuales no están soportadas por el propio Middleware. Por último, serán necesarias las librerías Boost^[45].

Una vez instalado todo esto, se enlaza el proyecto con las dependencias necesarias del Middleware y se utilizan los objetos *Builder* proporcionados(véase el Patrón de Diseño en la Sección 5.5.3. Con ellos se podrán crear, recibir y enviar mensajes a través de su plataforma.

8.5.2.Capa de comunicación interna

Un mensaje en DDS RTI dispone de distintas posibilidades de configuración. Para ello el Middleware de PLANET proporciona un *Builder* que se encarga de configurarlo de una forma sencilla.

Una vez preparado el mensaje a enviar, sólo será necesario llamar al método que lo enviará a través de la red. Sólo será necesario indicar el mensaje y canal de comunicación por el que deberá ir.

A la hora de publicar mensajes, es necesaria la creación del objeto *AMSPublisher*, el cual dispone de un *switch* que gestiona cada una de las posibles señales que pueden ser publicadas por un objeto *Place*. Esto es así ya que cada señal es distinta y habrá señales que apenas necesiten tratamiento, y otras que necesiten crear mensajes específicos para ser enviados por la red, informando así a otros módulos externos desarrollados por otros socios, como, por ejemplo, el despegue de un *UAV* gestionado por el módulo de control en tierra desarrollado por otros compañeros.

Finalmente, para cada mensaje recibido, también será necesario crear un objeto que lo trate y lo traduzca a la posible activación o desactivación de señales en la Red de Petri. Estos objetos son llamados "Procesadores de mensajes asociados a un tópico determinado".

8.5.3. Manejador de señales

Las señales propiamente dichas se crean en un archivo de configuración, indicando el nombre correspondiente de cada una. La capa de comunicación se encargará de obtener estos nombres y almacenarlos en un conjunto clave-valor en el objeto *Signals*. Este conjunto clave-valor dispondrá del nombre de cada señal y su estado actual correspondiente, **true** o **false**.

Para manejar las señales desde cualquier punto de la Red de Petri se crea el objeto *SignalHandler*. Desde este objeto se podrá activar, desactivar o reseñear el valor de cada una de las señales, sólo con indicar su nombre. Además dispondrá del método *boolean getSignalStatus(String signalID)* que devolverá el estado guardado de la señal.

Por último, este objeto también se encargará, de una forma muy sencilla, de publicar las señales por la red, pues sólo habrá que indicar su nombre. Esto es posible debido a que contiene un objeto publicador de mensajes de DDS RTI, ahorrando muchas líneas de código.

Indicar que este objeto se ha diseñado como un *Singleton*. Ya que este objeto será utilizado por distintas partes de la Red de Petri simultáneamente, se ha considerado requisito indispensable que se disponga solamente de una única instancia, de esta forma se evitará confusiones a la hora de leer el estado de las señales por posibles solapamientos, siendo además un objeto *thread-safe*, que garantiza la exclusión mutua y permite que el código sea concurrente.

8.5.4. Historias de usuario y Pruebas realizadas

Para completar este desarrollo, se ha necesitado un *Sprints*, el séptimo que define la integración de los mensajes DDS RTI en la Red de Petri la capa de comunicación que enlaza la Red de Petri con el [word]gMw de comunicación DDS RTI.

Las historias completadas son:

9720 – Integración de DDS en en proyecto.

Algunas de las Pruebas Unitarias desarrolladas para el Middleware DDS RTI son:

- Integrar DDS RTI-RTI en el proyecto.
- Construir una capa de abstracción para los mensaje suscritos.
- Enviar y recibir mensajes de otros ordenadores o elementos comunicados por la red.

9690 – Crear una capa de comunicación.

Algunas de las Pruebas Unitarias desarrolladas para capa de comunicación son:

- Crear una capa de comunicación que enlace el envío y recepción de señales con el envío y recepción de mensajes DDS RTI.
- Crear una capa para enviar señales cómodamente.

Por último, como Pruebas de Integración, se requiere una prueba que haga uso de la capa de comunicación desde la Red de Petri y envíe mensajes usando el Middleware DDS RTI. De la misma forma, las señales externas serán recibidas y modificarán el estado de la Red de Petri.

La última de las Pruebas de Integración necesarias es la más importante, es la que prueba todo el conjunto global desarrollado, comprobando que la comunicación entre las distintas capas es correcta y la interpretación de las señales se hace debidamente.

8.5.5. Diagrama de BurnDown

Estado del diagrama de *BurnDown* tras el séptimo *Sprint* de desarrollo:

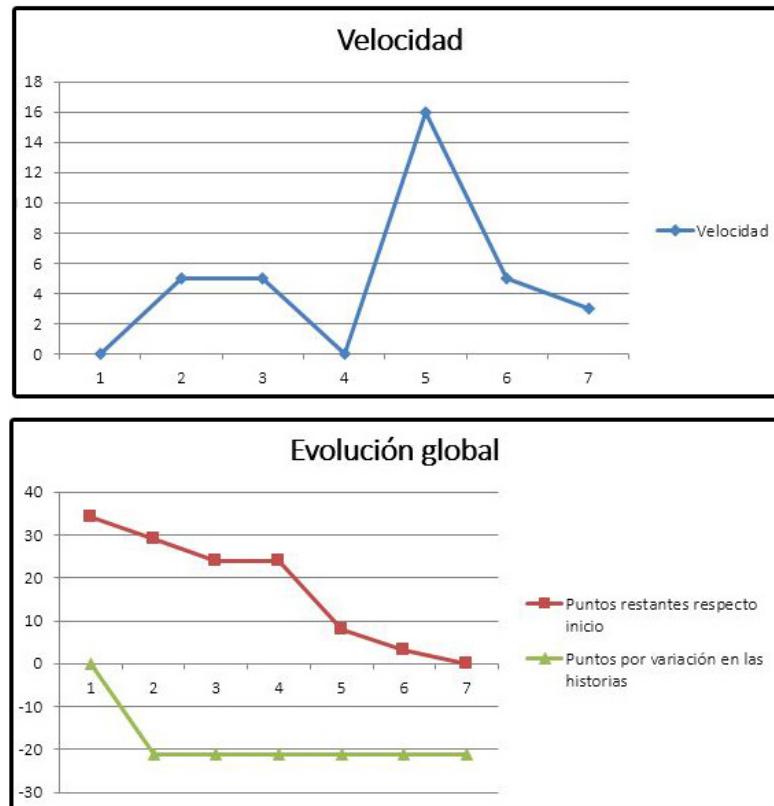


Figura 8.5: Diagrama de BurnDown para Sprint 7.

Podemos apreciar cómo las capas encajan unas con otras y consiguen hacer un sistema completo con la Red de Petri como nexo de unión entre todas.

8.6. Aplicación Web

En esta sección se procede a describir el proceso de desarrollo de la Web Page utilizando el Patrón de Diseño M.V.C., véase Sección 5.5.9 para más información. Como se ha estudiado, este desarrollo estará dividido en tres grandes partes, **Modelo, Vista y Controlador**.

8.6.1. Modelo

Se puede decir que esta capa es como el cerebro de la Web Page, ya que es donde toda la Lógica de Negocio es desarrollada. Esto quiere decir que todo lo que ocurra en esta capa será invisible para el usuario y que además, será el encargado de cambiar el estado del sistema y hacer que lo mostrado en la capa **Vista** pueda ser posible.

En este proyecto, esta capa será ocupada por la ya desarrollada Red de Petri, de la cual se podrá obtener toda la información necesaria. El **Controlador** usará toda esta información para presentarla adecuadamente al usuario enviándola a la capa de **Vista**.

8.6.2. Vista

La Vista es la capa con la que interactúa directamente el usuario, por lo que debe ser fácil de utilizar y agradable en su uso.

Para lograr una interacción completa con la Red de Petri, necesitaremos varias vistas distintas, desarrolladas con páginas JSP y maquetadas debidamente con CSS[55].

Página Principal

La página principal de la Web Page dará la bienvenida al usuario, desde donde se podrá acceder a la reserva de una hora de vuelo, a la vista del aeropuerto y a la vista de controles avanzados. Además incluye varios textos que dan información al usuario acerca del AMS.

La reserva de la hora está ligada directamente con el número de UAVs disponibles. Esto significa que si sólo hay un UAV, sólo se podrá reservar una hora, y no se podrá volver a reservar otro vuelo hasta que éste vuelva a estar desocupado. Para el caso en el que se dispongan de más UAVs, se podrá reservar tantos vuelos como UAVs disponibles. Un UAV se considera disponible cuando está ocioso, es decir, no tiene programado un vuelo o no está volando.



Figura 8.6: Vista de la página principal del AMS.

Vista del Aeropuerto

En esta vista es donde el usuario puede visualizar todo lo que está ocurriendo en el aeropuerto, así como interactuar con él cuando sea necesario. Se mostrará qué cosas se podrán ver y hacer:

- Visualizar el número de UAVs estacionados en el Hangar.
- Visualizar el número de UAVs en misión, es decir, volando.
- Visualizar los datos de la estación meteorológica en tiempo real.

- Visualizar un UAV cuando hace Taxi a la pista.
- Visualizar un UAV cuando hace Taxi al hangar.
- Visualizar cuando un UAV despega o aterriza.
- Visualizar obstaculización de la pista y su despeje.
- Visualizar estado del espacio aéreo de la pista.
- Visualizar posición del obstáculo.
- Visualizar horas de vuelo reservadas.
- Poder enviar la orden de limpieza de pista a un UGV a una posición guardada u otra concreta.
- Informar al AMS de que la pista está despejada.
- Informar al AMS de que el espacio aéreo está ocupado o despejado.



Figura 8.7: Vista del Aeropuerto en el AMS.

Controles avanzados

En la vista de controles avanzados se dejan a un lado los efectos visuales y se centra en la visualización de todos los estados actuales de cada una de las señales disponibles en el AMS. Además permitirá simular mensajes recibidos por la Red de Petri, para que así pueda cambiar de estado y poder probar su funcionalidad. Éste es principalmente su propósito, las pruebas, ya que al ser un proyecto que depende de componentes externos con un UAVs una estación meteorológica, no estarán accesible en cualquier momento.



Figura 8.8: Vista de la página principal del AMS.

8.6.3. Controlador

Esta capa es la que se encarga de traducir las peticiones de la capa **Vista** y obtener las respuestas pertinentes de la capa **Modelo**, impulsada por la Red de Petri. De la misma forma, irá informando pertinentemente a la **Vista** de los cambios realizados e ella.

Estas tareas de linkado se han dividido en dos partes. Por un lado, la gestión de eventos por Struts2[21] que responderá a las peticiones de la **Vista**, y por otro, la gestión de señales internas y datos necesarios también para la capa **Vista**.

Gestión de eventos

Uno de los aspectos más importantes a tener en cuenta para trabajar con Struts2[21] es la configuración del archivo XML de configuración, donde se construye la funcionalidad de cada elemento.

Para el desarrollo de la **Vista** se han desarrollado tres páginas distintas, las cuales serán enlazadas con los correspondientes *Actions*, los cuales se ocuparán de obtener o modificar los datos necesarios de la Red de Petri, como por ejemplo preguntar cuántos UAVs se encuentran volando, obtener los datos de la estación meteorológica, enviar un UGV a limpiar la pista, preguntar por el estado de cada una de las señales para la vista de controles avanzados o comprobar si un UAV puede ir a la pista.

Cabe destacar también que se utilizará un objeto *Interceptor*, que se encargará de inicializar la Red de Petri, suscribir al AMS a cada uno de los mensajes que recibirá por el Middleware DDS RTI, comprobar si alguna hora de vuelo reservada se ha cumplido e informar al AMS si se ha dejado de percibir que hay un obstáculo en la pista.

Gestión de Datos y señales

Para la gestión de los datos creados y accedidos desde la web, se creará un objeto *Singleton*(véase Sección 5.5.1). De nuevo utilizaremos este Patrón de Diseño para poder guardar y suministrar los datos necesarios de cuantas llamadas sean necesarias, sin necesidad de utilizar una base de datos y garantizando la concurrencia de accesos a un mismo dato sin corromper la información. Para lograr este propósito se ha usado el Patrón de Diseño *Command* estudiado anteriormente.

La reserva de horas de vuelos, tendrá un *Action* específico, el cual dispondrá también de un objeto *Singleton* con todas las horas reservadas. Éste a

su vez se encargará de la validación de las horas, como el formato en el que se introducen o el intervalo de tiempo necesario para poder reservar un vuelo detrás de otro, que será de diez minutos.

Para la suscripción de mensajes DDS RTI que harán que la Red de Petri cambie de estado, se dispondrá de un objeto *Container* que realizará esta tarea de forma sencilla y con una sola llamada. De la misma forma, para publicar mensajes Data Distribution Service por la red, se dispondrá de un objeto *Publisher* que creará los mensajes con las características necesarias y los publicará a través del Middleware.

Parte V

Conclusiones



Capítulo 9. Valoraciones y objetivos

En este capítulo se procederá a numerar las conclusiones obtenidas y los objetivos completados tras el desarrollo del proyecto.

9.1. Valoraciones

A continuación se enumeran las valoraciones obtenidas tras el desarrollo:

- 1.- El uso de metodologías ágiles se adapta muy bien al desarrollo de un proyecto fin de carrera.
- 2.- Las metodologías ágiles aumentan el compromiso de los desarrolladores al ser ellos los que se comprometen a tener terminadas las tareas para cada sprint.
- 3.- Demostrada una gran adaptabilidad a los cambios y motivación por el uso de nuevas tecnologías desconocidas.
- 4.- Se considera importante la experiencia de haber trabajado en proyecto real en un entorno empresarial.

- 5.- El dueño de producto y tutor ha quedado muy convencido de la utilidad de las metodologías ágiles.
- 6.- El proyecto ha conseguido cumplir con todos los objetivos.

Como conclusión, reconocer que el resultado ha sido satisfactorio y se considera que el trabajo realizado cumple con los objetivos establecidos.

9.2. Objetivos Cumplidos

Tras las valoraciones obtenidas tras finalizar el proyecto, se procede a repasar los objetivos cumplidos en el desarrollo.

9.2.1. Objetivos orientados a la metodología

Una vez introducidos los conocimientos necesarios de Scrum, la puesta en práctica de los mismos ha sido bastante intuitiva. Scrum ha resultado ser un Marco de Trabajo muy flexible, con el que se ha podido comprobar la facilidad para agregar o modificar las tareas durante el desarrollo.

Los resultados de Kanban también han sido muy positivos, pudiendo ser aplicado después de Scrum de forma muy sencilla, donde se ha podido comprobar también la comodidad y sencillez a la hora de agregar o eliminar tareas.

De la misma forma, las técnicas de eXtreme Programming, como el Programación por parejas y T.D.D. han resultado muy positivas, mejorando muchísimo la calidad del código desarrollado.

Siendo así, se consideran estos objetivos cumplidos y podemos concluir que usar Metodologías Ágiles ha sido una experiencia muy grata, obteniendo un muy buen resultado del desarrollo del proyecto.

9.2.2. Objetivos orientados a la técnica

Aplicar cada uno de los conocimientos de las Metodologías Ágiles ha sido sencillo y muy intuitivo. El desarrollo de cada una de las tecnologías ha sido un trabajo duro aliviado con el Programación por parejas, pero que ha dado sus frutos, obteniendo como resultado la conclusión de este proyecto. Con lo que se considera que los objetivos han sido cumplidos.

9.2.3. Objetivos personales

La experiencia de trabajo final ha sido muy positiva. Las metodologías usadas han hecho del desarrollo del proyecto algo ameno y fácil de llevar. Los problemas que han ido surgiendo a lo largo del desarrollo se han ido dividiendo en pequeñas tareas aprovechando la flexibilidad que nos proporcionan estas metodologías, con lo que han sido mucho más fáciles de solucionar llegado el momento.

Como punto final, reconocer la seguridad que genera tener el código en un repositorio en la red y cubierto por montones de tests, que elimina toda posible excusa y miedo para hacer una refactorización del código para añadir nuevas funcionalidades.



Parte VI

Apéndices



Capítulo 10. Ejemplos de Uso

En este capítulo se expondrán, paso a paso, varios ejemplos de uso de la Red de Petri. Para mantener una referencia de cómo se tá comportando el AMS, debemos tener presente cómo está definida la Red de Petri:

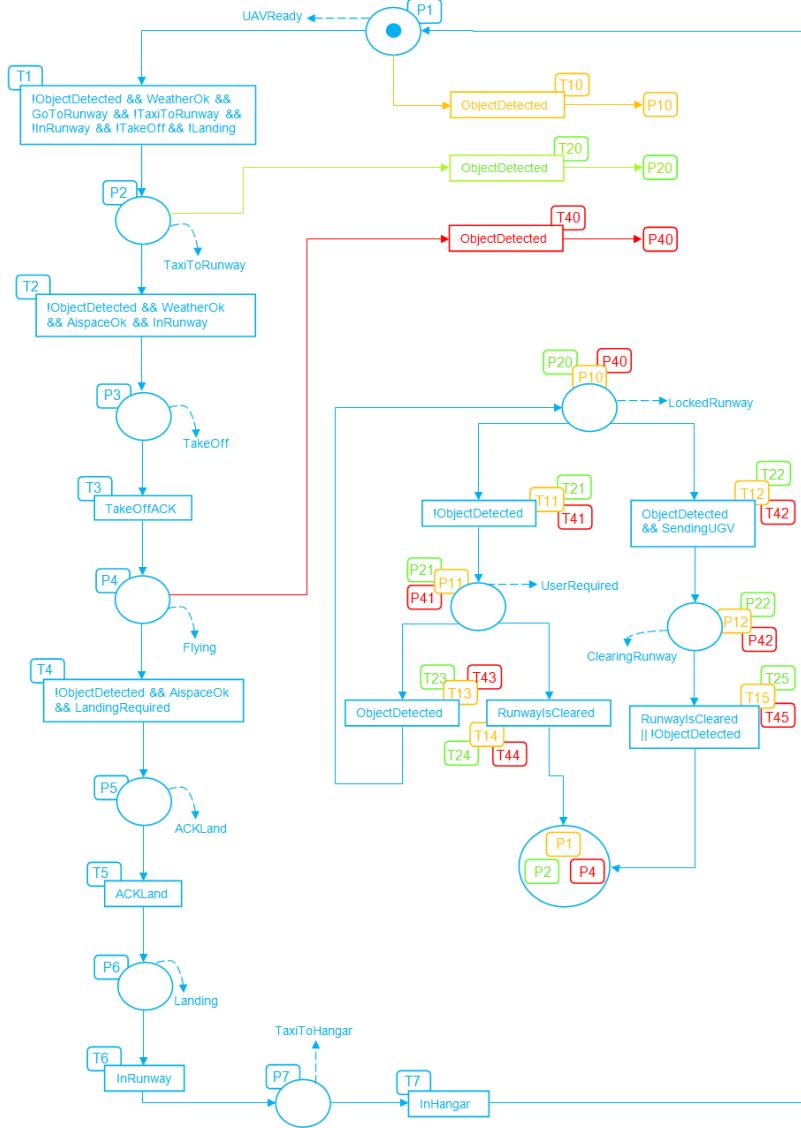


Figura 10.1: Red de Petri a implementar.

Además de estos ejemplos, se disponen de varios vídeos suministrados junto con este documento que muestran el desarrollo de estos ejemplos.

10.1. Reserva de vuelo, despegue y aterrizaje para un UAV

Lo primero que se debe de hacer para ver en marcha el AMS es reservar una hora para un vuelo:



Figura 10.2: Reservando un vuelo.

Tras esto, se puede ver cómo queda reservado el vuelo en las distintas vistas del AMS:



Figura 10.3: Vuelo reservado en la vista principal.

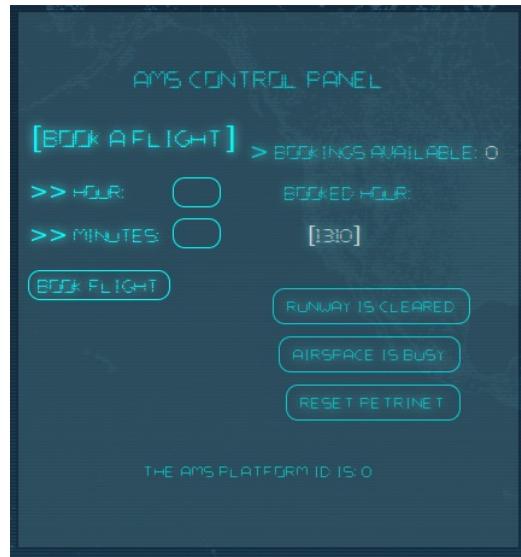


Figura 10.4: Vuelo reservado en la vista de control.



Figura 10.5: Vuelo reservado en la vista del aeropuerto.

Una vez se alcanza en el sistema la hora reservada, el UAV estará preparado para desplazarse hasta la pista de despegue. Se puede ver esto en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.6: Hora del despegue en la pantalla del aeropuerto.

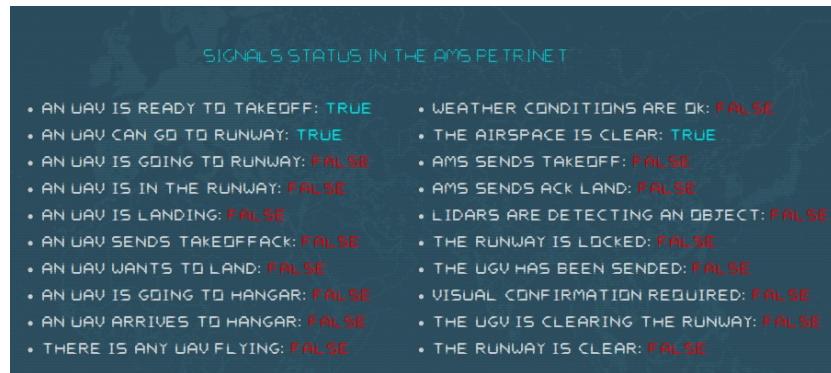


Figura 10.7: Señales en la pantalla de control.

El AMS, tras comprobar que las condiciones meteorológicas son aceptables y que ningún objeto obstaculiza la pista, da permiso para que el UAV se desplace hasta la pista. Además de esto, el contador de UAVs disponibles en el hangar se decrementará en uno. Se puede ver en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.8: El UAV se dirige a la pista.

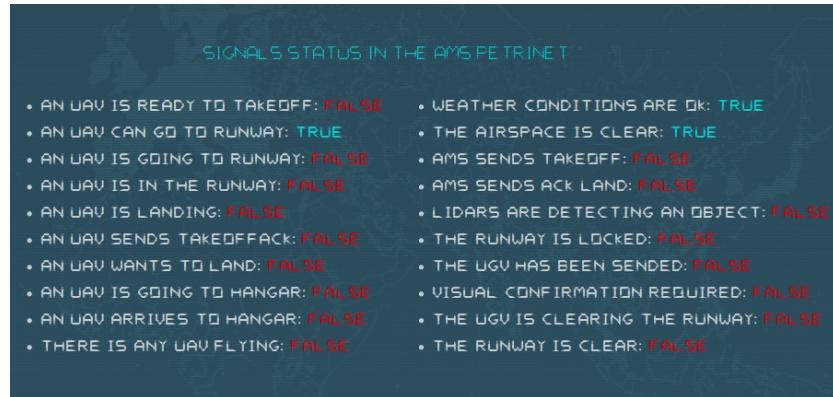


Figura 10.9: Estado de las señales en el control.

Cuando el AMS detecte por telemetría que el UAV ha llegado a la pista, comprueba de nuevo que las condiciones meteorológicas son aceptables, que ningún objeto obstaculiza la pista y que el espacio aéreo de la pista está despejado. Siendo así, se procede al despegue, cerrando el espacio aéreo del aeropuerto. Se pueden ver los cambios en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.10: El UAV está en la pista.



Figura 10.11: El UAV puede despegar.

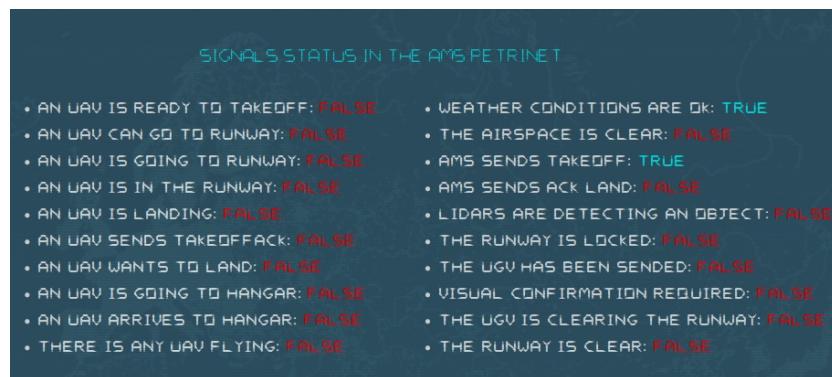


Figura 10.12: Señales en el despegue.

Una vez que el UAV está volando, el espacio aéreo del aeropuerto queda

abierto de nuevo, y el contador de UAVs en misión aumenta en uno. Se puede comprobar en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.13: El UAV está volando.

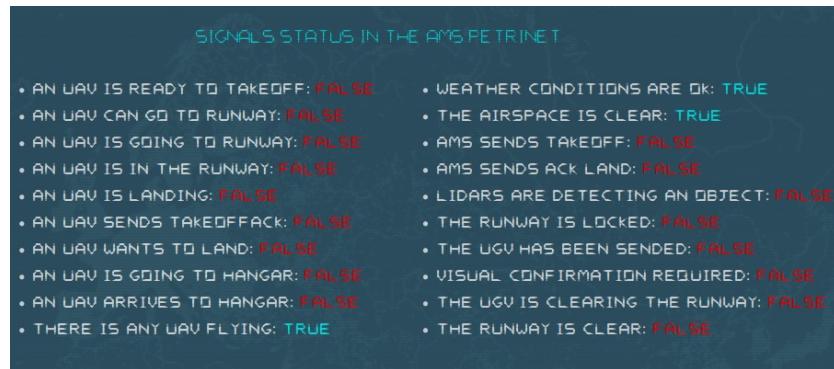


Figura 10.14: Señales tras el despegue.

Tras un tiempo de vuelo, el UAV enviará una señal de fin de misión al AMS. Éste, tras comprobar que ningún objeto obstaculiza la pista y que el espacio aéreo de la pista está despejado, le concede permiso para aterrizar, cerrando de nuevo el espacio aéreo de la pista. Se puede ver esto en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.15: El UAV puede aterrizar.

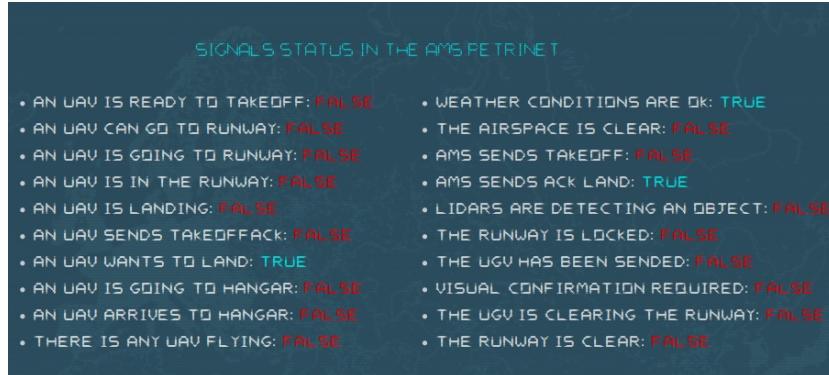


Figura 10.16: Señales al aterrizar.

En el momento que el UAV toma tierra, el AMS lo detecta por telemetría, y envía automáticamente la señal para realizar el Taxi hasta el hangar, dejando de nuevo libre el espacio aéreo de la pista. Se puede apreciar en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.17: El UAV se dirige al hangar.

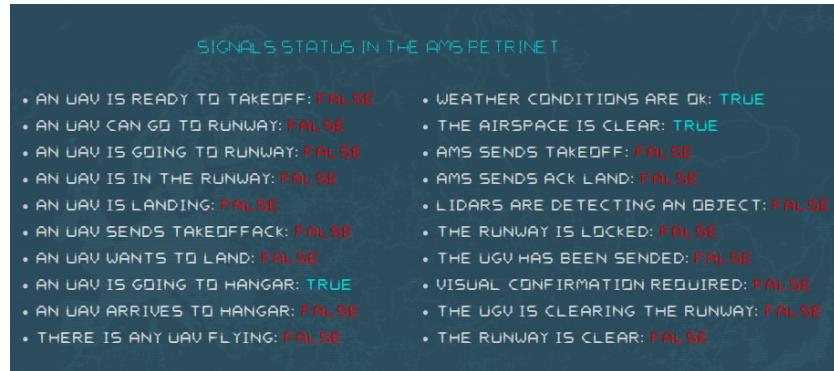


Figura 10.18: Vista de las señales en la pantalla de controles avanzados.

Finalmente el UAV llega al hangar, donde estará de nuevo preparado para realizar otro despegue, sólo se tendrá que reservar otro vuelo. Se puede ver esto en la pantalla de aeropuerto y en la de controles avanzados:



Figura 10.19: El UAV vuelve al hangar.

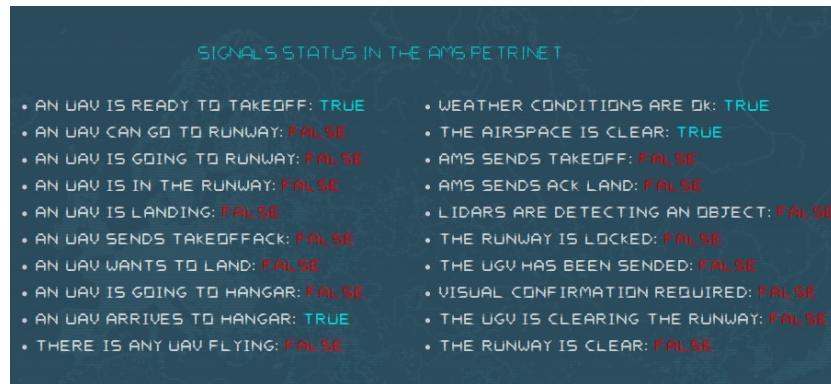


Figura 10.20: Señales tras el aterrizaje.

Es necesario aclarar que un UAV pasa a estar *en misión* cuando está volando fuera del espacio aéreo de la pista, y dejará de estarlo cuando aterrice en la pista.

10.2. Reserva de vuelo, despegue y aterrizaje para tres UAVs

En este apartado veremos las diferencias que se pueden observar en el AMS al disponer de tres UAVs distintos para volar.



Figura 10.21: Horas de despegue en vista principal.



Figura 10.22: UAVs disponibles en el aeropuerto.

En primer lugar se procede a la reserva del vuelo. Como se puede ver, al reservar un vuelo, se actualiza el número de UAVs disponibles para reservas:



Figura 10.23: Nueva disponibilidad de reservas.

Una vez el UAV ha despegado, se sigue manteniendo el número de vuelos para reservar disponibles, ya que hasta que el UAV no termine su misión, no se le podrá asignar otra hora de vuelo. De la misma forma, el contador de UAVs disponibles y en misión se actualizará en la pantalla del aeropuerto en cada momento.



Figura 10.24: Ya no hay hora reservada.



Figura 10.25: Contador de UAVs actualizado.



Figura 10.26: Contador actualizado tras despegue.

Estas son las diferencias de la vista con respecto a un sólo UAV en el hangar, el resto de animaciones serán las mismas.

10.3. Detección y retirada de un obstáculo

Antes que el UAV toque la pista, ya sea para despegar o para aterrizar, el AMS comprobará que está despejada y que ningún obstáculo le impide hacerlo. Si en alguna de estas comprobaciones se detecta un obstáculo, el AMS no otorgará los permisos necesarios al UAV para que pueda maniobrar en la pista. Se muestra qué pasa cuando el obstáculo es detectado:



Figura 10.27: Detección de un obstáculo.



Figura 10.28: Señales en la pantalla de control.

Una vez detectado, el usuario decide enviar un UGV para retirar el obstáculo:



Figura 10.29: UGV enviado.

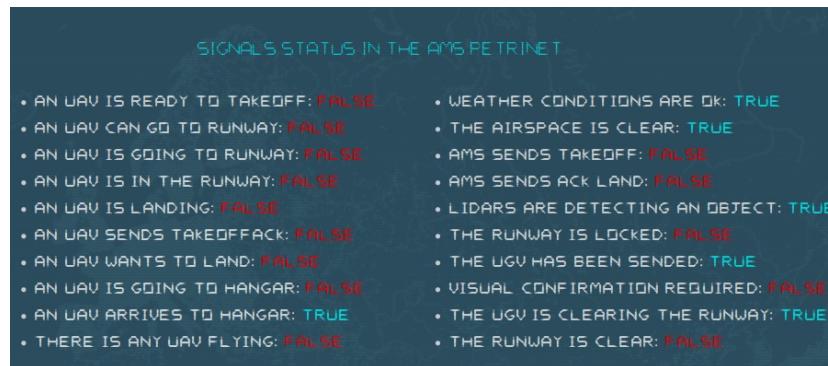


Figura 10.30: Estado de las señales.

Tras ser retirado el obstáculo por el UGV, el AMS vuelve al estado anterior y estará preparado para gestionar un nuevo despegue:



Figura 10.31: Pista despejada de nuevo.



Capítulo 11. Hibernate

Aunque se ha procedido a su estudio y valoración, en este proyecto finalmente no ha sido necesario el uso de esta herramienta, ya que no se dispone de B.D.. De todas formas, como anexo se adjunta una introducción a esta tecnología.

11.1. Introducción a Hibernate

Hibernate[22] es una herramienta ORM para la plataforma Java (también para .Net con el nombre de NHibernate), que facilita el mapeo de atributos entre una Base de Datos Relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los Beans de las entidades que permiten establecer estas relaciones.

En pocas palabras, es un *Framework* que agiliza la relación entre la aplicación y la B.D..



Figura 11.1: Logo de Hibernate.

11.2. ORM

El Mapeo Objeto-Relacional (ORM) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje P.O.O. y el utilizado en una Base de Datos Relacional, utilizando un D.B.M.S.[\[14\]](#). Esto posibilita el uso de las características propias de la Programación Orientada a Objetos.

Con sólo configurar un ORM, se obtienen todas las típicas tareas repetitivas realizadas y el desarrollador solo tendrá que preocuparse por aquellas consultas fuera de lo normal.

11.3. Funcionamiento de Hibernate

Hibernate funciona asociando a cada tabla de la B.D. un P.O.J.O..

```
public class Cat {  
    private String id;  
    private String name;  
    private char sex;  
    private float weight;  
  
    public Cat() {}  
  
    public String getId() {  
        return id;  
    }  
  
    private void setId(String id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public char getSex() {  
        return sex;  
    }  
}
```

```

        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}

```

Para poder asociar el P.O.J.O. a su tabla correspondiente en la B.D., Hibernate usa los ficheros *hbm.xml*.

Para la clase Cat se usa el fichero *Cat.hbm.xml* para mapearlo con la B.D.. En este fichero se declaran las propiedades del P.O.J.O. y sus correspondientes nombres de columna en la B.D., asociación de tipos de datos, referencias, relaciones n a n con otras tablas, etc.

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A Cat has to have a name, but it shouldn't be too long. -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

```

De esta forma, se podrá usar el siguiente código para comunicarnos con la B.D.:

```
sessionFactory = new Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

session.close();
```

Hibernate tiene la ventaja de hacer totalmente transparente el uso de la B.D., pudiendo cambiarla sin necesidad de cambiar ni una sola línea del código. En su lugar basta con modificar los ficheros de configuración.

Glosario

Aircraft Management System (AMS) *Sistema de Gestión de Aeronaves*, es el sistema encargado de garantizar el aterrizaje y despegue de un UAV, atendiendo a distintos factores, como la meteorología y los posibles obstáculos en la pista de despegue y aterrizaje., pág. 4.

Analizador sintáctico (Parseador) el análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos., pág. 6.

Aplicación Web (Web App) es una aplicación software que se codifica en un lenguaje soportado por los navegadores Web en la que se confía la ejecución al navegador a la cual accede el usuario a través de Internet o de una intranet, pág. 6.

Application Programming Interface (A.P.I.) *Interfaz de programación de aplicaciones*, es el conjunto de funciones y procedimientos o métodos que ofrece una librería para ser utilizado por otro software como una capa de abstracción., pág. 63.

Bárbara Liskov (Liskov) destacada científica de la computación estadounidense., pág. 32.

Base de Datos Relacional (B.D.) es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso., pág. 32.

Código abierto (Open Source) es el término con el que se conoce al software distribuido y desarrollado libremente.[24], pág. 88.

Capability Maturity Model Integration (C.M.M.I.) *Integración de modelos de madurez de capacidades*, es un modelo para la mejora y evaluación de procesos para el desarrollo, mantenimiento y operación de sistemas de software[28]., pág. 18.

Common Gateway Interface (CGI) *Interfaz de entrada común*, especifica un estándar para transferir datos entre el cliente y el programa a través de un navegador Web., pág. 183.

Contenedor de IoC (IoC Container) es el contenedor que inyecta las dependencias necesarias al objeto cuando es creado., pág. 90.

Contenedor de Servlets (Servlet Container) se encarga de realizar todo el trabajo de las conexiones y gestionar la vida del Servlet. Véase su funcionamiento detallado en la Sección: 5.15, pág. 76.

Contenedor Web (Web container) es la herramienta del Servidor Web que toma los Servlets y las JSPs[46], les da el trato necesario y se lo pasa como respuesta al mismo Servidor Web donde está configurado., pág. 77.

Data Distribution Service (DDS RTI) *Sistema de Distribución de Datos para sistemas en tiempo real*, es la especificación para un middleware

de tipo publish-subscribe en sistemas distribuidos. Basa su funcionamiento en el patrón de diseño Observer, el cual está especificado en la Sección: 5.5.6, véase su funcionamiento detallado en la Sección: 5.8.[8], pág. 4.

DeadLine fecha límite o fin de plazo para terminar una tarea., pág. 36.

Desarrollo ágil de software (Metodologías Ágiles) son métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requerimientos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios. Existen muchos métodos de desarrollo ágil; la mayoría minimiza riesgos desarrollando software en lapsos cortos.[16], pág. XI.

Dinamic Web App (Página Web Dinámica) son aquellas en las que la información presentada se genera a partir de una petición del usuario de la página. Contrariamente a lo que ocurre con las páginas estáticas, en las que su contenido se encuentra predeterminado, en las páginas dinámicas la información aparece inmediatamente después de una solicitud echo por el usuario., pág. 74.

Divide y Vencerás este algoritmo hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original., pág. 13.

Dueño del producto (Product Owner) persona o personas que conocen y marcan las prioridades del proyecto o producto en Scrum., pág. 25.

Entidad Abstracta (Abstracción) permite que dispongamos de las características esenciales de un objeto que necesitemos, las cuales lo distinguen de los demás., pág. 33.

Entorno de Desarrollo Integrado (IDE) *Integrated Development Environment*, es un entorno de programación que ha sido empaquetado como un programa de aplicación; es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica, pág. 64.

eXtensible Markup Language (XML) *Lenguaje de Marcas eXtensible*, es un lenguaje de marcas desarrollado por el W3C utilizado para almacenar datos en forma fácilmente legible.[19] Para más información, consultar la Sección detallada: 5.11, pág. 6.

eXtreme Programming (X.P.) Es un enfoque de la ingeniería de software formulado por Kent Beck, que pertenece al Desarrollo ágil de software. Véase detallado en la Sección: 2.1, pág. 6.

FADA-CATEC (CATEC) La Fundación Andaluza para el Desarrollo Aeroespacial, FADA, tiene como fin el desarrollo y la promoción de actividades de I+D+i susceptibles de fomentar el desarrollo económico del sector Aeroespacial en Andalucía y promover la generación y explotación de nuevos conocimientos y tecnologías. Es por ello que gestiona el Centro Avanzado de Tecnologías Aeroespaciales, el cual tiene como objetivo la mejora de la competitividad de las empresas del sector, mediante el impulso de la creación de conocimiento, la gestión de la propiedad intelectual de la I+D+i y la innovación tecnológica.[10], pág. 3.

Gang of Four (GoF) es el nombre con el que se conoce comúnmente a los autores del libro *Design Patterns*[31], referencia en el campo del diseño orientado a objetos., pág. 50.

Hibernate es una herramienta ORM para la plataforma Java originalmente, que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos XML o anotaciones en los Beans de las entidades que permiten establecer estas relaciones.[22], pág. 8.

Historias de Usuario es una representación de un requisito de software escrito en una o dos frases utilizando el lenguaje común del usuario., pág. 26.

Hypertext Transfer Protocol (HTTP) *Protocolo de Transferencia de Hipertexto*, es el protocolo usado en cada transacción de la World Wide Web., pág. 77.

Inversión de Control (IoC) *Inversion of Control*, es un método de programación en el que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos o funciones. Véase información detallada en la Subsección: 5.5.8., pág. 90.

Inyección de Dependencias (D.I.) Patrón de Diseño que deriva de IoC, hace uso de la modularidad y la reutilización[30]. Véase información detallada en la Subsección: 5.5.8., pág. 33.

Java Beans (Beans) son un modelo de componentes creado por Sun Microsystems[13], para la construcción de aplicaciones en Java. Se usan para encapsular varios objetos en otro único (vaina o Bean), para hacer uso de un sólo objeto en lugar de varios más simples.[20], pág. 88.

Java SDK (Java) es un lenguaje de P.O.O. creado por Sun Microsystems, Inc.[13] que permite crear programas que funcionan en cualquier tipo de ordenador y sistema operativo.[20], pág. 6.

JavaServer Pages (JSP) es una tecnología que ayuda a los desarrolladores de software a crear Dinamic Web App[46].., pág. 6.

JUnit es un conjunto de bibliotecas para hacer pruebas unitarias de aplicaciones Java SDK., pág. 23.

Kanban es un método para gestionar la carga de trabajo, haciendo especial hincapié en la entrega justo a tiempo sin sobrecargar de trabajo a los

miembros del equipo.[\[18\]](#) Para más información, consultar la Sección detallada: 3.4, pág. 6.

Kent Beck es un ingeniero de software estadounidense, uno de los creadores de eXtreme Programming (X.P.) y Test-Driven Development (T.D.D.), las llamadas Metodologías Ágiles., pág. 15.

Lógica de Negocio parte del sistema que se encarga de las rutinas que realizan entradas de datos, consultas a los datos, generación de informes y más específicamente todo el procesamiento que se realiza detrás de la aplicación visible para el usuario, pág. 61.

Mapeo Objeto-Relacional (ORM) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje P.O.O. y el utilizado en una Base de Datos Relacional, utilizando un Motor de persistencia., pág. 171.

Marco de Trabajo (Framework) define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar., pág. 6.

Middleware es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, software, redes, hardware o sistemas operativos., pág. 71.

Modelo Vista Controlador (M.V.C.) es un patrón de diseño que separa los datos y la Lógica de Negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones., pág. 61.

Motor de persistencia (D.B.M.S.) *data base management system*, es una aplicación o componente que resuelve la Persistencia pero soporta además un número significativo de características, como Encapsula-

miento del medio persistente, Integridad y escalabilidad.[14], pág. 172.

Patrón Singleton (Singleton) patrón de diseño indicado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Véase información detallada en la Subsección: 5.5.1., pág. 52.

Persistencia de objetos (Persistencia) acción de preservar la información de un objeto de forma permanente (guardar), pero a su vez también se refiere a poder recuperar la información del mismo (leer) para que pueda ser nuevamente utilizada., pág. 180.

Plain Old Java Object (P.O.J.O.) es una instancia de una clase que no extiende ni implementa nada en especial., pág. 44.

PLAtform for the deployment and operation of heterogeneous NETworked cooperating object
Plataforma para el despliegue y operación de objetos heterogéneos y cooperativos en red, este proyecto tiene como objetivo la integración de distintos sistemas inteligentes como redes de sensores, UAVs y UGVs en dos entornos de aplicación: Parque Natural de Doñana y un aeródromo inteligente.[9], pág. 3.

Principios S.O.L.I.D. (S.O.L.I.D.) *Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion.* Éste es un acrónimo mnemónico introducido por Robert C. Martin que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo.[15] Para más información, consultar la Sección detallada: 3.5, pág. 5.

Programación Orientada a Objetos (P.O.O.) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento

y encapsulamiento., pág. 31.

Programación por parejas (Pair Programming) se basa en que dos programadores trabajan juntos en un solo ordenador. Uno de ellos desarrolla mientras que el otro ayuda y revisa el código del compañero. Véase información detallada en la Sección: 3.1, pág. 16.

Project Object Model (POM) es un archivo XML utilizado por Maven[39] que describe el proyecto, sus dependencias, los plugins que utiliza, y otros datos, como la conexión con el sistema de control de versiones, o definición de otros repositorios Maven[39] que usemos en nuestro proyecto para descargar dependencias[41]., pág. 93.

Pruebas de Integración (Integration tests) se refieren a la prueba o pruebas de todos los elementos unitarios que componen un proceso, hecha en conjunto, de una sola vez., pág. 92.

Red de Petri (PetriNet) es una herramienta gráfica y matemática de modelación que se puede aplicar en sistemas que se definen en términos causa-evento. Particularmente son ideales para describir y estudiar sistemas que procesan información, y que tienen características concurrentes, asíncronas, distribuidas, paralelas y no determinísticas. Para más información, consultar la sección detallada: 5.4, pág. 6.

Refactoring (Refactorización) es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo., pág. 15.

repositorio es un sitio centralizado donde se almacena y mantiene información digital., pág. 93.

Scrum Scrum es un marco de trabajo para la gestión y desarrollo de software basada en un proceso iterativo e incremental utilizado comúnmente en entornos basados en el desarrollo ágil de software.[17] Para más

información, consultar la Sección detallada: 3.3, pág. 5.

Scrum Master es la persona que asegura el seguimiento de la metodología guiando las reuniones y ayudando al equipo ante cualquier problema que pueda aparecer. Su responsabilidad es entre otras, la de hacer de paraguas ante las presiones externas., pág. 25.

Servidor Web (Web Server) sirve contenido estático a un navegador, carga un archivo y lo sirve a través de la red al navegador de un usuario., pág. 60.

Servlet son programas escritos en Java que dan una respuesta alternativa a la programación Web con CGI, ampliando su funcionalidad. Se ejecutan en un Servidor Web dentro de un Contenedor de Servlets. Véase su funcionamiento detallado en la Sección: 5.15, pág. 44.

Seventh Framework Programme (FP7) El Séptimo Programa Marco de investigación, que abarca el período 2007-2013, ofrece a la UE la ocasión de poner su política de investigación a la altura de sus ambiciones económicas y sociales mediante la consolidación del Espacio Europeo de la Investigación. Para alcanzar este objetivo, la Comisión desea aumentar el presupuesto anual de la UE en materia de investigación y, de este modo, atraer más inversiones nacionales y privadas. Durante su aplicación, el Séptimo Programa marco también debe responder a las necesidades, en términos de investigación y conocimiento, de la industria y de forma más general de las políticas europeas. El Programa se articula alrededor de cuatro programas principales y se ha simplificado en gran parte para ser más accesible a los investigadores y más eficaz.[1], pág. 3.

Software Process Improvement Capability Determination (SPICE) *Determinación de la Capacidad de Mejora del Proceso de Software*, es un modelo para la mejora, evaluación de los procesos de desarrollo, mantenimiento de sistemas de información y productos de software[29]., pág. 18.

Spring [23] es un Framework para el desarrollo de aplicaciones y contenedor de IoC, de Open Source para la plataforma Java[20]., pág. 6.

Struts2 es un Framework para el desarrollo de Web Apps, el cual hace que la implementación de las mismas sea más sencillo, más rápido, y con menos complicaciones. Además hace que estas sean más robustas y flexibles.[21], pág. 6.

Taxi desplazamiento sobre la pista del aeropuerto de un avión o de un UAV en este caso., pág. 143.

Test-Driven Development (T.D.D.) *Desarrollo guiado por pruebas de software*, es una práctica de programación que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para más información, consultar la Sección detallada: 3.2, pág. 5.

Uniform Resource Identifier (URI) *Identificador Uniforme de Recursos*, es una cadena corta de caracteres que identifica inequívocamente un recurso., pág. 60.

Uniform Resource Locator (URL) *Localizador de Recursos Uniforme*, es una secuencia de caracteres, de acuerdo a un formato modélico y estándar, que se usa para nombrar recursos en Internet para su localización o identificación, pág. 76.

Unit Tests (Pruebas Unitarias) es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión., pág. 16.

University of Duisburg-Essen (UDE) La Universidad de Duisburg-Essen es una universidad pública en Duisburg y Essen, Renania del Norte-

Westfalia, Alemania. Con sus 12 departamentos y más de 39.000 estudiantes, la Universidad de Duisburg-Essen se encuentra entre las 10 mayores universidades alemanas. Muchos estudiantes internacionales que estudian en la Universidad de Duisburg-Essen y dan las ciudades de Duisburg y Essen un ambiente internacional. [2], pág. 3.

Unmanned Aerial Vehicle (UAV) *Vehículo Aéreo no tripulado*, es una aeronave que vuela sin tripulación humana a bordo., pág. 3.

Unmanned Ground Vehicle (UGV) *Vehículo terrestre no tripulado*, es un vehículo que opera mientras está en contacto con el suelo y sin una presencia humana a bordo., pág. 3.

Web Page (Página Web) es el nombre de un documento o información electrónica adaptada para la World Wide Web y que puede ser accedida mediante un navegador., pág. 77.

World Wide Web (Web) *Red Informática Mundial*, es un sistema de distribución de información basado en hipertexto o hipermedios enlazados y accesibles a través de Internet. Con un navegador World Wide Web, un usuario visualiza Página Webs que pueden contener texto, imágenes, vídeos u otros contenidos multimedia, navegando a través de esas páginas usando Web Pages., pág. 64.

World Wide Web Consortium (W3C) es un consorcio internacional que produce recomendaciones para la World Wide Web. Para más información: [12], pág. 71.



Referencias

- [1] Página Web de la unión europea donde se detalla el FP7:
http://europa.eu/legislation_summaries/energy/european_energy_policy/i23022_es.htm
- [2] Página Web de la UDE: <http://www.uni-due.de/>
- [3] *Deutsches Zentrum für Luft- und Raumfahrt e.V.* o Centro Aeroespacial Alemán, centro de investigación nacional para aviación y vuelos espaciales de Alemania y de la Agencia Espacial Alemana.
<http://www.dlr.de/dlr/en/desktopdefault.aspx/tabcid-10002/>
- [4] Empresa de *Finmeccanica*, es un líder internacional en tecnologías electrónicas y de la información de los sistemas de defensa, aeroespacial, datos, infraestructuras, seguridad y protección territorial y soluciones "inteligentes" sostenibles. Para más información: <http://www.selex-es.com/>
- [5] *Boeing Research and Technology Europe*, forma parte de Boeing, en cuya vocación clara por Europa, apuestan por la tecnología del Software como facilitador de los procesos de la industria Aeronáutica. Para más información:
<http://www.boeing.es/>
- [6] El Consejo Superior de Investigaciones Científicas es la mayor institución pública dedicada a la investigación en España y la tercera de Europa. Para más información: <http://www.csic.es/>

- [7] La Reserva Biológica de Doñana está incluida en los límites del Parque Nacional de Doñana en Almonte, Huelva, y está constituida por dos fincas o Reservas Científicas, para más información: <http://www.ebd.csic.es/website1/Zesp/Reserva.aspx>
- [8] RTI es una compañía de software de infraestructura en tiempo real, la cual es uno de los proveedores más conocidos de DDS RTI. para más información: <http://www.rti.com/>.
- [9] PLANET: Plataforma para el despliegue y operación de objetos heterogéneos y cooperativos en red, véase: <http://www.planet-ict.eu/>
- [10] FADA-CATEC, para más información: <http://www.catec.com.es/>
- [11] *Clean Code: A Handbook of Agile Software Craftsmanship*, Robert C. Martin[15]. Véase: <http://www.cleancoders.com/>
- [12] W3C, dirigida por Tim Berners-Lee, véase: <http://www.w3.org/>
- [13] Sun Microsystems, Inc. fue una empresa informática que se dedicaba a vender estaciones de trabajo, Web Servers, componentes informáticos, software y servicios informáticos. Fue adquirida en el año 2009 por Oracle Corporation. Es la creadora de Java (véase Sección 5.6).
- [14] Motores de persistencia: <https://sites.google.com/site/estrategiasdepersistencia/material-teorico/db-persistencia-y-dbms-motor>
- [15] Robert C. Martin, o conocido habitualmente como Tío Bob, promovedor del Código Limpio y promovedor de los Principios S.O.L.I.D. : http://en.wikipedia.org/wiki/Robert_C._Martin
- [16] Manifiesto por el Desarrollo Ágil de Software:

<http://www.agilemanifesto.org/iso/es/manifesto.html>

- [17] Todo lo que necesitamos saber sobre Scrum[17] (véase: 3.3) está en:
<https://www.scrum.org/>
- [18] Todo lo que necesitamos saber sobre Kanban está en:
<http://www.kanbanblog.com/explained/>
- [19] La W3C nos enseña todo lo necesario sobre XML:
<http://www.w3.org/XML/>
- [20] Página Web oficial de Java SDK: <http://www.oracle.com/technetwork/es/java/index.html>
- [21] Página Web oficial de Struts2: <http://struts.apache.org/development/2.x/>
- [22] Página Web oficial de Hibernate: <http://www.hibernate.org/>
- [23] Página Web oficial de Spring: <http://www.springsource.org/>
- [24] Para más información sobre Open Source: <http://opensource.org/>
- [25] *Henrik Kniberg.* Scrum and X.P. from the Trenches. InfoQ, 2007.
- [26] *Ken Schwaber, Mike Beedle.* Agile Software Development with Scrum. Pearson Education, 2001.
- [27] *Kent Beck* . Extreme Programming Explained: Embrace Change. Addison-Wesley, 2004.

- [28] Página Web oficial de Capability Maturity Model Integration: <http://www.sei.cmu.edu/cmmi/>
- [29] Página Web oficial de Software Process Improvement Capability Determination: <http://www.isospice.com/>
- [30] Para más información sobre IoC: <http://martinfowler.com/bliki/InversionOfControl.html>
- [31] *Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- [32] Master-D, empresa especializada en la enseñanza a distancia: <http://www.masterd.es/>
- [33] Subversion (SVN), es un sistema de Control de Versiones (véase Capítulo: 6). Página Web oficial de SVN: <http://subversion.apache.org/>
- [34] Git, es un sistema de Control de Versiones (véase Capítulo: 6). Página Web oficial de Git: <http://git-scm.com/>
- [35] Mercurial (hg), es un sistema de Control de Versiones (véase Capítulo: 6). Página Web oficial de Mercurial: <http://mercurial.selenic.com/>
- [36] Concurrent Versions System (CVS), es un sistema de Control de Versiones (véase Capítulo: 6). Página Web oficial de CVS: <http://cvs.nongnu.org/>
- [37] TortoiseHg es un cliente de Control de Versiones de Mercurial (véase Capítulo: 6). Página Web oficial de TortoiseHg: <http://tortoisehg.bitbucket.org/>
- [38] Bitbucket es un servicio de alojamiento basado en web, para los proyec-

tos que utilizan el sistema de Control de Versiones Mercurial y Git (véase Capítulo: 6). Página Web oficial de Bitbucket: <https://bitbucket.org/>

[39] *Maven* es una herramienta de software para la gestión y construcción de proyectos Java, para más información consulte su Página Web: <http://maven.apache.org/>

[40] Adobe Flex: <http://www.adobe.com/products/flex.html>

[41] Información extendida sobre el POM: <http://www.javaworld.com/javaworld/jw-05-2006/jw-0529-maven.html>

[42] Plugin *Maven* para eclipse: <http://eclipse.org/m2e/>

[43] JUnit es un conjunto de librerías utilizadas en programación para hacer Pruebas Unitarias de aplicaciones Java SDK, para más información: <http://junit.org/>

[44] Página Web oficial de Google Protocol Buffers: <https://code.google.com/p/protobuf/>

[45] Página Web oficial de las librerías Boost: <http://www.boost.org/>

[46] Para más información sobre JSP: <http://www.oracle.com/technetwork/java/index.html>

[47] Para más información sobre JSTL: <http://www.oracle.com/technetwork/java/index-jsp-135995.html>

[48] Para más información sobre Apache Tomcat: <http://tomcat.apache.org/>

- [49] Carl Adam Petri fue un matemático y científico de la computación alemán, para más información: http://es.wikipedia.org/wiki/Carl_Adam_Petri
- [50] Paquete *regex* de Java[20], más información: <http://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html>
- [51] Entorno de Desarrollo Integrado *eclipse*, para más información: <http://www.eclipse.org/>
- [52] Página Web oficial de Ubuntu: <http://www.ubuntu.com/>
- [53] Página Web oficial de OpenJDK: <http://openjdk.java.net/>
- [54] Para más información sobre XStream: <http://xstream.codehaus.org/>
- [55] Las *Cascading Style Sheets* (CSS), hacen referencia a un lenguaje de hojas de estilos usado para describir la presentación semántica (aspecto y formato), de un documento escrito en lenguaje de marcas, como HTML. Para más información: <http://www.w3schools.com/css/>