ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

SIMFORPAS

Simulador para RPAS

Realizado por

MANUEL MATEOS GUTIÉRREZ
32076954-G

Dirigido por

IRENE ALEJO TEISSIÈRE

PABLO TRINIDAD MARTÍN-ARROYO

Departamento

LENGUAJES Y SISTEMAS INFORMÁTICOS

Sevilla, Mayo de 2014

Agradecimientos

Agradecimientos

Índice general

Índice de figuras

Parte I

Introducción

Introducción

AUNQUE el concepto de aviones no tripulados o UAV's (Unmanned Aerial Vehicles) es bastante antiguo, puesto que se ha hecho uso de ellos desde la primera guerra mundial, cada día oímos más hablar sobre ellos en los medios de comunicación, esto se debe al gran crecimiento que está sufriendo el sector de la aeronáutica en torno a estos dispositivos, tanto para uso militar, como los famosos "drones"de Estados Unidos, como civil.

Su uso es amplio y variado, desde rodaje de planos aéreos en películas de cine hasta control de incendios, control de costas, recogida de información, ayuda en operaciones de rescate, control de multitudes...

Figura 1: Soporte aéreo en el control de incendios.

A finales del siglo XX fue cuando los UAV's empiezan a operar con todas las características de autonomía. Esto nos provee de muchas ventajas, por ejemplo, presencia en lugares de difícil acceso sin necesidad de llevar al terreno a un piloto de UAV, reducción del riesgo humano en determinadas situaciones, disminución del la incursión humana sobre parques naturales y zonas protegidas... . Poco a poco los UAV's tienden a prescindir de la presencia de un piloto que tenga la obligación de estar visualizando el avión y a implementar sistemas de control remoto mediante estaciones de control de tierra o GCS's (Ground Control Stations) y de vuelo automatizado, lo que nos llevará a no depender del factor humano.

Las GCS son controladas por operadores expertos en estos dispositivos que se encargan de diseñar e implementar las misiones que realizarán los aviones, así como llevar el control del curso de la misma, conocer las características de la aeronave, deben saber

interpretar los indicadores de telemetría, estar familiarizados con el protocolo de comunicación y saber reaccionar ante posibles fallos durante la misión para salvaguardar en todo momento la seguridad tanto del vehículo aéreo como del entorno en el que se mueve.

Estos operadores requieren de una formación en profundidad y fiable ya que tienen la responsabilidad sobre las acciones que realice la aeronave, por ello se debe exigir un entrenamiento concienzudo. Si este entrenamiento es realizado con dispositivos reales corremos el riesgo de que frente a cualquier fallo, error humano o de carácter informático, haya una pérdida en algún componente del sistema, ya sea que se estrelle la aeronave, que dañe alguna estructura o a alguna persona, lo que resultaría en una importante pérdida económica y/o humana.

El proyecto SIMFORPAS pretende dar una solución a este problema presentando un entorno de simulación de vuelo de UAV's para operadores de GCS en formación, que proveerá de un contexto de vuelo seguro e idéntico a una situación real de control de misión de un UAV.

0.1. Motivación

Según un estudio publicado por el medio online *Update Defense* y realizado por la firma de investigación de mercados *ICD Research* durante la próxima década el sector de la aviación no tripulada tendrá un aumento anual del 4,08 % lo que supondrá que en 2021 alcance alrededor de los 10.500 millones de dólares. El gran incremento de la demanda se traducirá en una mayor necesidad de infraestructuras y tecnologías en torno a estos dispositivos.

En vistas de estas espectativas resulta interesante implicarse de una forma activa en un mercado en auge que supondrá la aceptación de un gran número de nuevas tecnologías y traerá nuevos desafíos en cuanto a investigación y desarrollo.

Uno de los requisitos que tendrá esta etapa será pa de disponer de personal cualificado para la manipulación de los dispositivos de pilotage remotos de aeronaves no tripuladas, el proyecto SIMFORPAS pretende ocupar ese hueco proveyendo de un entorno seguro y fiable que permita conceder una certificación avanzada a operadores de GCS de forma que se aseguren los conocimientos técnicos necesarios en una situación real de pilotage.

En este proyecto propondremos una solución usando una serie de tecnologías que nos proveerán de las herramientas necesarias para crear el sistema necesario para la consecución de nuestro objetivo.

0.2. OBJETIVOS DEL PROYECTO

EL objetivo de este proyecto será el de crear una plataforma de simulación para la formación y entrenamiento de pilotos de RPAS (Remotely Piloted Aircraft System) ligeros que se comporte exactamente como lo haría el avión real. Que el sistema permita hacer uso de una GCS homologada para el manejo de UAV's usando un protocolo de comunicaciones para aviones no tripulados de menos de 25 Kg y usando un modelo de avión real.

También se requerirá de una herramienta que permita a un instructor ser capaz de controlar la simulación permitiéndole manejar su curso e introducir errores en el sistema. La simulación se deberá hacer en tiempo real.

Para garantizar la correcta consecución de los objetivos generales del proyecto se utilizará la herramienta de organización SCRUM junto a otras metodologías de desarrollo ágil.

0.2.1. OBJETIVOS ORIENTADOS A LA METODOLOGÍA

Los objetivos que se tienen en mente al realizar esta aplicación con respecto a las metodologías usadas son los siguientes:

- Aplicar el marco de trabajo Scrum, usando para ello los conocimientos adquiridos al trabajar en empresas que utilizan dicha metodología y cursos. También se dispone del apoyo bibliográfico de libros como *Agile Samurai* y *Agile Software Development with Scrum*
- Aplicar metodologías de programación en pareja para agilizar el desarrollo y evitar errores en el código.
- Aplicar los principios S.O.L.I.D. como base de un código robusto, limpio y sujeto a cambios.
- Aplicar metodologías de eXtreme Programming para asegurar que el código acepte cambios de manera sencilla e intuitiva.
- Comprobar los beneficios colaterales a la realización de estas prácticas como, por ejemplo, la facilidad de añadir nuevas tareas durante el proceso de desarrollo.

0.2.2. OBJETIVOS ORIENTADOS A LA TÉCNICA

Los objetivos que se han querido validar al realizar esta aplicación son los siguientes:

- Aprender y utilizar tecnologías que garanticen una comunicación y procesado de datos en tiempo real como pueden ser C++ y DDS.

- Aprender y utilizar para el puesto de instructor herramientas que ayuden al desarrollo de una plataforma web para el control del simulador como Maven, Struts2, Spring4 e Hibernate4.
- Aprender y utilizar entornos de testeo de código como Google test, Google mock, Junit y Jmock para asegurar que el código funciona en todas las fases de desarrollo y modificación del software.
- Aplicar correctamente cada una de las metodologías estudiadas y sacar conclusiones de su uso.

0.2.3. 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. Los desarrolladores tienen la motivación de aprender nuevas técnicas que mejoren la calidad del software.
- Confianza: Conseguir conocimientos que me permitan en un futuro abatir exitosamente un proyecto software.
- Experiencia: Adquirir aptitudes para solucionar problemas y añadir valor a un grupo de trabajo en un entorno laboral.
- Conocimientos Técnicos: Trabajar y sintetizar nuevas tecnologías que me sirvan en el futuro para completarme como profesional en mi campo.

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.

0.3. ESTRUCTURA DEL DOCUMENTO

Éste documento se estructura en las siguientes partes:

PREFACIO: En éste capítulo se introduce el proyecto creando el contexto de su implementación, explicando en qué consiste, la motivación que nos ha llevado a desarrollarlo y los objetivos que se quieren cumplir en el mismo, así como éste mismo apartado de estructura del proyecto en el que explicamos qué vamos a encontrarnos en ésta memoria y cómo está distribuida y un índice de contenidos y de figuras.

CONCEPTOS BÁSICOS: Introducimos las metodologías que hemos usado durante el desarrollo del proyecto así como los conceptos básicos necesarios para comprender todo el documento, una enumeración de tecnologías usadas y un glosario de terminología, también se explicará el método de desarrollo iterativo e incremental que hemos llevado a cabo y en el que se basa la documentación del proyecto.

SISTEMA A DESARROLLAR: Aquí se enumerarán cada una de las etapas de desarrollo que ha ido sufriendo el proyecto SIMFORPAS desarrollando la planificación para cada iteración y cada problema que ha ido surgiendo durante el mismo, también se aportará el diagrama de Burndown para monitorizar en todo momento el estado del proyecto.

CONCLUSIONES: Realizaremos una retrospectiva final del proyecto analizando su estado final, la consecución de los objetivos, los cambios con respecto a la planificación inicial que se han realizado y las posibles mejoras y futuro del proyecto SIMFORPAS.

APENDICES: Para finalizar añadiremos un apéndice de definiciones, un manual de usuario y la bibliografía usada durante el desarrollo del proyecto y la memoria.

Parte II

Conceptos básicos

Capítulo 1. Metodologías usadas

En éste capítulo empezaremos haciendo una introducción a las metodologías ágiles, explicando su filosofía y el por qué de su existencia así como una serie de técnicas para implementar este tipo de metodologías a nuestro proyecto software y qué beneficio nos aporta.

El proyecto fue desarrollado haciendo uso del sistema SCRUM de desarrollo iterativo, se explicará en qué consiste éste método y como ha sido aplicado a SIMFORPAS.

1.1. Metodologías ágiles

El proceso normal afianzado hasta ahora en el desarrollo software sigue unas pautas de rigidez que evita que el producto esté sometido a cambios ya que cuanto más avanzado está el desarrollo del proyecto más difícil y costoso resulta la introducción de modificaciones, para ello se definen unos requisitos que debe cumplir el producto final y antes de empezar el proyecto se decide las tecnologías a usar y la planificación del desarrollo, el cliente no toma parte en el proceso de implementación sino que cuando llega la fecha indicada para la finalización se le presenta y se evalúa si se ha conseguido el resultado que él esperaba.

Como pueden imaginar en la mayoría de los casos debido al desconocimiento real del problema no se definen correctamente los requisitos o las tecnologías usadas y surgían problemas imprevistos en la planificación que retrasan la fecha de entrega o acortan el tiempo de desarrollo obligando al equipo a dedicar más horas repercutiendo todo esto negativamente en el resultado final.

En otros casos la entrega se hace a tiempo pero debido a la ausencia del cliente durante el proceso de desarrollo el producto final no responde a lo que él imaginaba que se iba a desarrollar causando descontento por parte de nuestro cliente y afectando a futuros contratos que podamos hacer con él mismo.

Para hacer frente a esta serie de problemas en torno al desarrollo software nacen las ”Metodologías Ágiles”, En 2001 un grupo de desarrolladores se reúne en Utah para discutir los *métodos de peso ligero* de desarrollo software y publicaron el *Manifiesto ágil*, un documento que resume la filosofía ágil y establece cuatro valores y doce principios.

1.1.1. MANIFIESTO ÁGIL

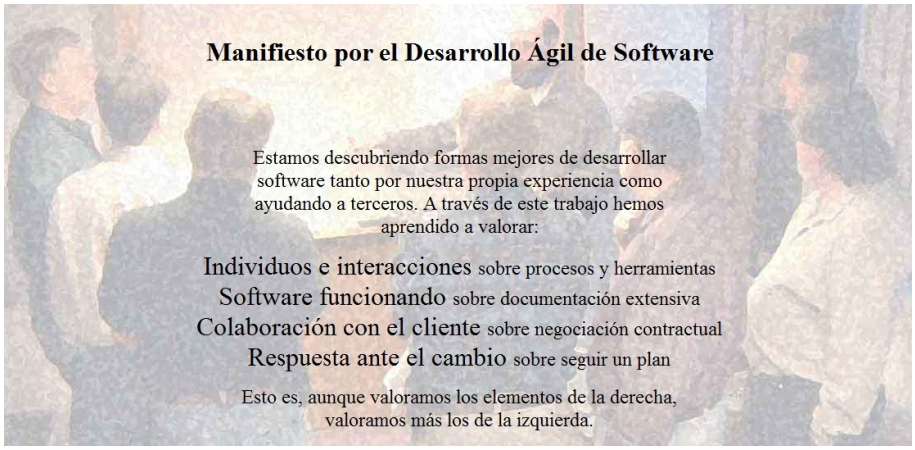

Figura 1.1: Manifiesto ágil.

VALORES:

- **Valorar más a los individuos y su interacción que a los procesos y las herramientas:** Este es posiblemente el principio más importante del manifiesto. Por supuesto que los procesos ayudan al trabajo. Son una guía de operación. Las herramientas mejoran la eficiencia, pero sin personas con conocimiento técnico y actitud adecuada, no producen resultados.
- **Valorar más el software que funciona que la documentación exhaustiva:** La documentación siempre será una medida importante pero no como guía para entender un código sino como complemento de un código claro y autoexplicativo. Al final lo que se debe valorar es un código ordenado y que funciona, que le da valor a un proyecto, por encima de una documentación que aporta datos y no información.
- **Valorar más la colaboración con el cliente que la negociación contractual:** Las prácticas ágiles están especialmente indicadas para productos difíciles de definir con detalle en el principio, o que si se definieran así tendrían al final menos valor que si se van enriqueciendo con retro-información continua durante el desarrollo. También para los casos en los que los requisitos van a ser muy inestables por la velocidad del entorno de negocio. En el desarrollo ágil el cliente es un miembro más del equipo, que se integra y colabora en el grupo de trabajo. Los modelos de contrato por obra no encajan.
- **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.

PRINCIPIOS:

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 diariamente a lo largo del proyecto.

5.- Construir el proyecto en torno a individuos motivados. Darles el entorno y apoyo que necesiten y confiar a 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.

1.1.2. DESARROLLO ITERATIVO INCREMENTAL CON SCRUM

INTRODUCCIÓN

PARA abordar la realización del proyecto SIMFORPAS se hará uso del modelo organizativo de trabajo Scrum. Una de las características de éste modelo es la búsqueda de una serie de beneficios, como por ejemplo, la capacidad de aceptación de nuevos cambios durante el desarrollo ya sean requeridos por el cliente como por el mercado, esto nos asegura que el cliente al finalizar el proyecto va a tener el producto que satisface a sus necesidades ya que de otro modo los requisitos pueden haber cambiado desde la definición inicial.

El equipo de trabajo se auto asignará las tareas a realizar, esto provoca que cada integrante se mantenga motivado ya que él mismo se ha puesto su objetivo. El desarrollo iterativo exige tener una versión funcional o una serie de resultados presentables al finalizar cada etapa del desarrollo, esto se traduce en una mayor calidad del software.

Utilizando herramientas como la gráfica de burn down es posible observar la velocidad que está llevando el equipo de desarrollo, esto es útil para detectar posibles problemas de rendimiento que haya que solucionar entre todo el equipo o una reorganización de la planificación así como para poder estimar el tiempo de duración del proyecto.

ROLES

- **Product owner:** El Product Owner representa la voz del cliente. Se asegura de que el equipo Scrum trabaje de forma adecuada desde la perspectiva del negocio. El Product Owner escribe historias de usuario, las prioriza, y las coloca en el Product Backlog.
- **ScrumMaster:** El Scrum es facilitado por un ScrumMaster, cuyo trabajo primario es eliminar los obstáculos que impiden que el equipo alcance el objetivo del sprint. El ScrumMaster no es el líder del equipo (porque ellos se auto-organizan), sino que actúa como una protección entre el equipo y cualquier influencia que le distraiga. El ScrumMaster se asegura de que el proceso Scrum se utiliza como es debido. El ScrumMaster es el que hace que las reglas se cumplan.
- **Equipo de desarrollo:** El equipo tiene la responsabilidad de entregar el producto. Un pequeño equipo de 3 a 9 personas con las habilidades transversales necesarias para realizar el trabajo (análisis, diseño, desarrollo, pruebas, documentación, etc).

Existen otros roles auxiliares como pueden ser proveedores, clientes, vendedores... sólo participarán directamente durante las revisiones de sprint.

DESARROLLO DEL PROYECTO CON SCRUM

Al principio se tiene una reunión con el cliente donde se recoge el objetivo del proyecto, los requisitos y las tareas que se llevarán a cabo para realizarlos, todo ello mediante historias de usuario en las que se asocia el rol a la necesidad del proyecto como se puede observar en el siguiente ejemplo: *Como desarrollador quiero un módulo central capaz de cambiar y monitorizar el estado del modelo*, de esta forma se deciden las tareas a realizar. Luego el equipo y el cliente discuten la prioridad en las tareas hasta llegar a un consenso de qué es más importante desarrollar primero y qué dejar para más adelante, de esta forma nos aseguramos de ir cumpliendo las necesidades más importantes para poder tener cuanto antes una versión funcional del proyecto. También se valorarán según la dificultad de cada una de ellas, facilitando de esta forma la elección de qué se realizará antes, las acciones que sean esenciales y fáciles se harán primero, y las difíciles y poco necesarias se dejarán para el final, el tiempo de realización de cada tarea se hará en función a la dificultad de la misma.

Una vez definidas las historias de usuario se define el tamaño de los *sprints*, normalmente un sprint es un espacio de tiempo de entre una y cuatro semanas en las que se desarrollarán determinadas historias de usuario. Cuando se define el primer sprint se colocan en una pizarra las historias de usuario, para cada historia se definirán unos test de aceptación que asegurarán una vez cumplidos que la tarea está finalizada y se colocarán

pequeñas sub-tareas necesarias para la realización de la historia de usuario. La pizarra tendrá varios *pools* que indicarán las sub-tareas a realizar, las que están en proceso y las que ya se han realizado. Éstas sub-tareas se irán cambiando de posición según sea su estado. La morfología de la pizarra de Scrum se muestra en la siguiente figura.

Durante el sprint se hará una reunión diaria entre el equipo y el ScrumMaster en la que se hablará del estado del proyecto, qué se realizó el día anterior, qué se realizará en ese día y qué problemas han surgido para buscar entre todos soluciones y que ningún miembro del equipo se quede estancado en una tarea.

Una vez finalizado el sprint se organiza una reunión de retrospectiva, a la que acudirá el equipo, el ScrumMaster y el product owner en la que se presentará el estado del proyecto, qué es lo que se ha llevado a cabo durante el sprint, qué problemas han surgido, que se podría modificar/mejorar. El cliente dará el visto bueno y hará las peticiones que vea necesarias, se hará la gráfica de burn down para documentar el estado de esa fase del proyecto y se organizarán las tareas para el siguiente sprint.

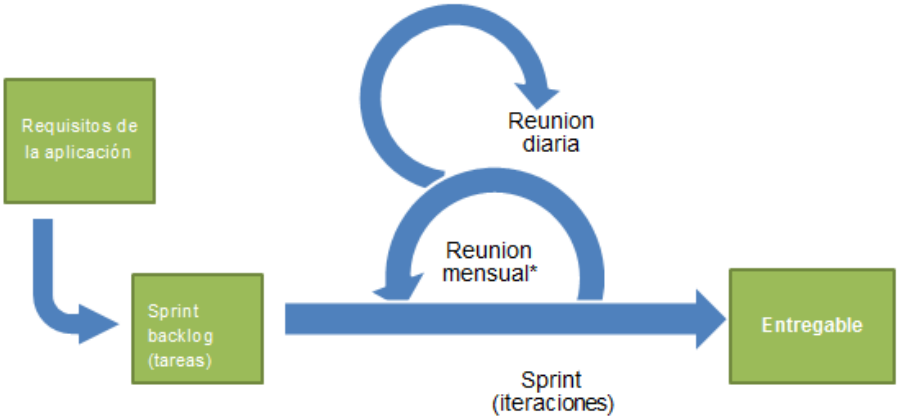

Figura 1.2: Ciclo metodología Scrum.

Esta forma de trabajo se repetirá hasta la finalización del proyecto, nos asegurará que el cliente está implicado en el desarrollo y que conocerá de antemano el producto que va a comprar y podrá interceder en su morfología. Con este método el equipo de desarrollo trabaja de una forma más relajada evitando la acumulación de trabajo a última hora y los estancamientos ya que entre el equipo debe fluir la comunicación y el problema que tenga uno en la realización de su parte se convierte en problema de todos. El cliente podrá añadir cambios o complementos al proyecto a sabiendas de que esos cambios vendrán con el sacrificio de otras historias de usuario programadas o de un incremento del tiempo de desarrollo y del coste del proyecto.

1.1.3. TEST DRIVEN DEVELOPMENT

EL desarrollo guiado por pruebas o TDD por sus siglas en inglés consiste en una práctica de programación que implica a su vez otras dos prácticas: Escribir las pruebas antes que el código y refactorizar. Una vez habiendo definido la funcionalidad de la parte del código que vamos a escribir hacemos un test que pruebe esa funcionalidad y una vez definido éste test y fallando nos disponemos a codificar la solución que lo resuelva, de esta forma nos aseguramos de que no perdemos ninguna función al programar el código. Si se hace al revés el test se ve afectado por la forma que tiene el código y tendemos a probar lo que sabemos que va a ocurrir dejándonos muchos casos sin resolver que pueden afectar más tarde al correcto funcionamiento de nuestro programa. Con esta técnica también nos aseguramos que en el momento en que se realice un cambio en el programa nada deja de funcionar, puesto que en todo momento el código debe pasar los test asegurando que no se pierde ninguna funcionalidad debida al nuevo cambio.

Una vez se ha escrito el test, se ha comprobado que fallaba y se ha resuelto viene la hora de refactorizar el código, como no sabemos a priori cómo va a ser el código final debemos probablemente el código que hemos escrito para pasar ese test sea memorable, por eso tenemos que estudiar la forma correcta de escribir esa parte del código, esto se hace mediante una refactorización. Una vez realizada ésta refactorización se vuelven a pasar los test, si no pasan habría que repasar el código para que se solucione el error y luego volver a repasar la refactorización.

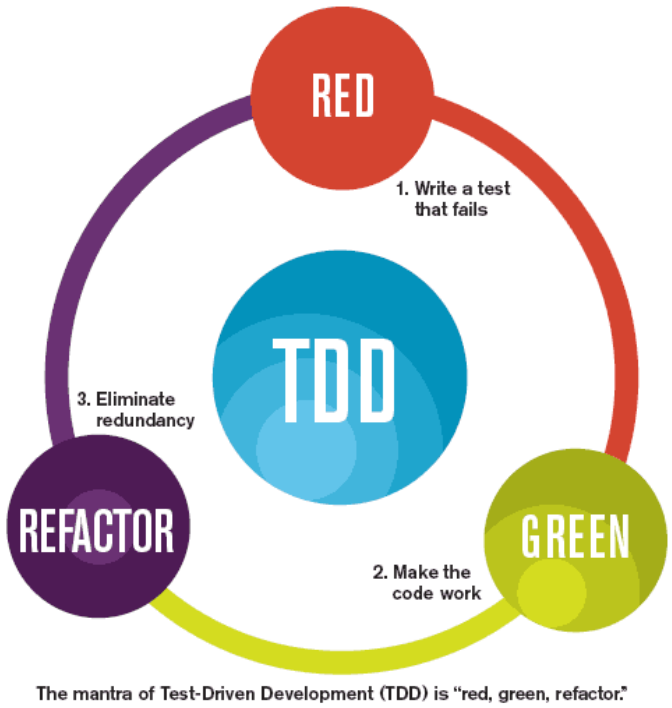

Figura 1.3: Ciclo TDD.

Mediante ésta técnica nos aseguramos un código limpio, bien estructurado y libre de errores. Otra funcionalidad de los test es explicar de qué manera se usa el código que estamos programando, ya que para probar nuestros métodos y clases debemos hacer uso de ellas, y este uso queda reflejado en el test.

1.1.4. PAIR PROGRAMMING

A la hora de programar es muy común perder mucho tiempo con errores al codificar así como en tomar decisiones correctas sobre qué forma darle al código, una técnica que evita estas situaciones es la programación por parejas, consiste en unir a dos desarrolladores para que programen juntos en el mismo puesto de trabajo, de forma que mientras uno programa el otro vigila que no tenga errores. También deciden entre los dos cómo hacer las cosas de una forma objetiva, siempre es bueno tener una segunda opinión y discutir cuál es la mejor solución a un problema.

A priori puede parecer que éste método hace que dos personas estén haciendo el trabajo de una, pero a la larga esto acelera el tiempo de desarrollo. También es muy útil a la hora de transmitir conocimientos a una nueva incorporación al equipo o para enseñar a programadores junior.

Cada cierto tiempo se pueden intercambiar los papeles lo que les permitirá a las dos partes coger soltura y ver el código con perspectiva de forma que puedan abstraerse y tener una visión global. Ésta técnica puede combinarse con la programación guiada por tests de forma que uno de los dos escribe el test y el otro tiene que escribir el código que lo resuelve para que el otro tenga la obligación de pensar de qué forma podría fallar y así tener una mayor cobertura frente a fallos.

Figura 1.4: Pair programming.

1.2. TECNOLOGÍAS

ANTES de iniciar el desarrollo del proyecto debemos decidir qué tecnologías son las más adecuadas a la hora de realizar ciertas tareas, como por ejemplo, unos de los requisitos para el simulador de UAV's es que las comunicaciones deben ocurrir en tiempo real, así como el procesado de datos, por tanto necesitamos un lenguaje que nos ofrezca esta velocidad como podría ser c++.

A continuación enumeraremos y daremos una breve explicación de cada una de las tecnologías usadas durante el desarrollo del proyecto SIMFORPAS.

1.2.1. PROGRAMACIÓN ORIENTADA A OBJETOS

LA programación orientada a objetos es un paradigma de programación en el que las funciones las realizan los *objetos*. Está basado en varias técnicas como la *herencia*, la *cohesión*, *abstracción*, *polimorfismo*, *acoplamiento* y *encapsulamiento*. La principal característica de éste paradigma es que relaciona el sistema con el mundo real, en el que cada entidad que cumple una función está representada por un objeto en el código, así podemos encontrar objetos controladores, fábricas, etc...

La herencia y el polimorfismo son técnicas que nos permiten crear un código mucho más legible, limpio, y fácil de mantener debido al encapsulamiento de responsabilidades que hace que cuando necesitemos encontrar una función de nuestro código sepamos en qué lugar buscar y no tengamos que depurar todas las líneas como ocurre en paradigmas como la programación estructural.

Entre las muchas ventajas de la programación orientada a objetos podemos encontrar la robustez del código debido a que una clase que no funcione correctamente no debe afectar al resto del código, es capaz de abstraer entidades del mundo real haciendo mucho más fácil manejarlas en nuestro programa, facilita el desarrollo del software y el trabajo en equipo ya que dos personas serán capaces de trabajar sobre distintas partes del código sin interferir una en el trabajo de la otra.

CARACTERÍSTICAS DE LA POO

- **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, y çomunicarseçon otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos, y, cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. 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. La abstracción es clave en el proceso de análisis y diseño

orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

- **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. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

- **Modularidad:** Se denomina modularidad a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas.

- **Principio de ocultación:** Cada objeto está aislado 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. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

- **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. O, 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. Algunos lenguajes proporcionan medios más estáticos (en ”tiempo de compilación”) de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

- **Herencia:** Las clases no se encuentran 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 polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos 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; siendo de alta complejidad técnica por lo cual suele recurrirse a la herencia virtual para evitar la duplicación de datos.

- **Recolección de basura:** La recolección de basura o garbage collection 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. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse expresamente.

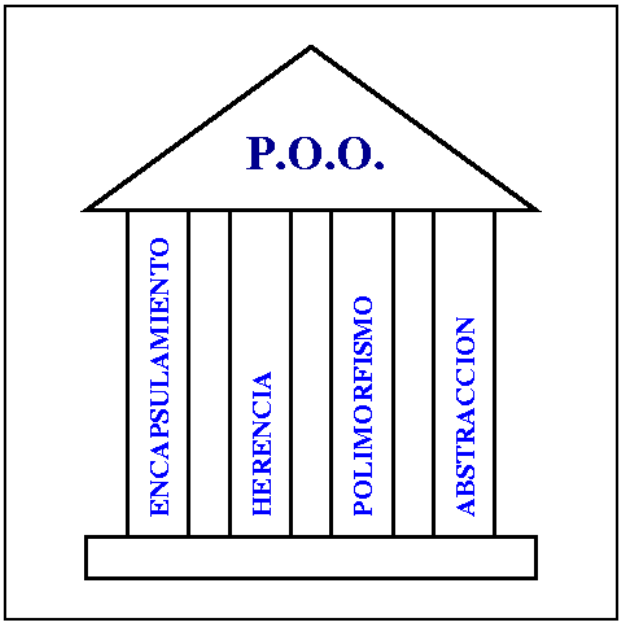

Figura 1.5: Pilares de la POO.

La manera que usaremos para sacar el mayor partido a la programación orientada a objetos será haciendo uso de los principios *S.O.L.I.D.* como guía de buenas formas a la hora de programar.

PRINCIPIOS S.O.L.I.D.

REPRESENTAN cinco principios básicos del uso de la POO y del diseño software. Éstos conceptos fueron recogidos por Robert C. Martin en torno al año 2000, los principios S.O.L.I.D. son una guía que ayuda al programador a elaborar un código más limpio, legible y fácil de mantener y extender. Su uso se adapta a dos conceptos que hemos visto anteriormente como son el de TDD y Refactorización, ya que estos dos tienen como finalidad buscar un código que cumpla siempre con estas directrices.

El acrónimo S.O.L.I.D. responde a las siguientes definiciones:

- **S**ingle responsibility principle: El principio de única responsabilidad dice que una clase sólo debería tener una única responsabilidad, de esa forma solo tendría una única razón para cambiar y así se contiene la propagación de cualquier cambio que realicemos sobre ella sin que afecte a otra parte del código que no tiene nada que ver con dicha responsabilidad.
- **O**pen/close principle: Nos explica la importancia de que el código esté abierto a su extensión pero cerrado a su modificación, esto nos permite modificar la funcionalidad de una clase sin necesidad de tocar su código, lo que requeriría revisiones, pruebas y comprobaciones de que todo sigue funcionando correctamente.
- **L**iskov substitution principle: Es una definición particular de una relación de subtipificación, llamada tipificación del comportamiento, esto quiere decir que en un código puede usarse cualquier clase hija del mismo padre sin que esto altere las propiedades de ese programa.
- **I**nterface segregation principle: Es una reflexión que apunta que es mejor tener muchas interfaces específicas a una genérica, de esta forma evitamos que el cliente haga uso de propiedades de la interfaz que no necesita o a las cuales no debería tener acceso, también es una buena herramienta de documentación de la funcionalidad del programa y así se define mejor la funcionalidad de pasa clase.
- **D**dependency inversión principle: Éste principio apunta que las dependencias entre partes del código deben hacerse sobre abstracciones no sobre implementaciones, de esta forma una clase que haga uso de otra no dependerá del código que se haya escrito para la segunda y si en algún momento éste cambiara no afectaría a la primera. Esto ayuda a mejorar el mantenimiento del código y lo prepara para futuras modificaciones.

1.2.2.C++

PARA la consecución de nuestro objetivo de rapidez a la hora del procesado de datos necesitamos un lenguaje que nos ofrezca esta característica, debido a la estructura del lenguaje de programación orientado a objetos *Java*, éste resulta lento y pesado a la hora de ejecutarse lo que supondría retrasos en el intercambio de datos, cosa que no nos podemos permitir cuando una aeronave depende de la comunicación que mantengamos con ella. C, al ser un lenguaje a más bajo nivel nos ofrece ésta característica, pero al ser un lenguaje estructural nos priva de la ventaja de los lenguajes orientados a objetos. Una buena combinación de estas dos características es el lenguaje C++, éste lenguaje es una extensión de C que nos permite la manipulación de objetos, desde el punto de vista de la programación orientada a objetos, éste es un lenguaje híbrido.

Figura 1.6: Ilustración c++11.

Debido a su base C de bajo nivel, teniendo que gestionar la memoria del programa, nos permite tener la velocidad requerida, por eso es la mejor opción para el sistema que queremos desarrollar. En 2011 se actualizó la biblioteca estándar de C++ con la nueva versión C++11, que ofrece nuevas funcionalidad para facilitar la labor del programador.

Una de las bibliotecas principales que usamos en C++ es Qt, una biblioteca multi-plataforma que nos permite, entre otras funcionalidades, crear aplicaciones con interfaz gráfica, el API de la biblioteca también cuenta con métodos para acceder a bases de datos, uso de XML, gestión de hilos y otras muchas funciones útiles a la hora de programar.

Éste lenguaje lo usaremos a la hora de realizar las funciones de simulación y comunicación entre los diferentes módulos del sistema, en el siguiente capítulo haremos hincapié en qué función requerirá el uso de éste lenguaje.

1.2.3. JAVA

EL proyecto podría dividirse en tres partes bien identificadas, por un lado tenemos el simulador de la aeronave, la GCS y el puesto de instructor, los dos primeros como ya hemos explicado requieren una respuesta rápida ya que el vuelo simulado depende de la rapidez en las comunicaciones y para asegurar que el entorno es similar a un vuelo en la vida real necesitamos tratar al simulador como si fuera una aeronave real.

Sin embargo el puesto de instructor no requiere una interacción con el sistema que sea en tiempo real, sino que importa más que la aplicación, al estar separada del resto de módulos, sea multi-plataforma para abstraernos del entorno desde el que se use y sea fácilmente portable, así como que disponga de un método de comunicación compatible con el resto del sistema. Java es un lenguaje totalmente orientado a objetos, se ejecuta sobre una máquina virtual (JVM) adaptada a la mayoría de sistemas operativos, lo que convierte cualquier aplicación java en multi-plataforma.

Figura 1.7: Logo Java.

Al ser un lenguaje orientado a objetos nos da todas las facilidades que ya explicamos antes y debido a la gran comunidad y a lo extendido que está éste lenguaje de programación tenemos un número infinito de herramientas y tecnologías que se pueden implementar con Java, incluido el método de comunicación entre módulos que usaremos para el intercambio de información que usaremos en el proyecto, por tanto es el lenguaje perfecto para lo que necesitamos. También permite la implementación de páginas web de manera rápida y sencilla lo que nos permitirá darle aún más accesibilidad a la aplicación ya que podría instalarse en un servidor y accederse desde cualquier terminal conectado a él.

1.2.4. PATRONES DE DISEÑO

A la hora de programar solemos encontrarnos una gran cantidad de veces con problemas recurrentes de diseño de cuya solución puede depender que nuestro código se alivie o que salga herido. Una mala solución a un problema normalmente acarreará más cambios en el resto del código, en cuanto al diseño del software hay una serie de problemas que son bastante conocidos ya que suelen aparecer frecuentemente, por ejemplo, tenemos que implementar una aplicación que trabaja con una base de datos y una interfaz de usuario gestionada por una clase que se comunica directamente con la base de datos y la interfaz representando estos datos. Si en algún momento se quisiera modificar cualquiera de las partes, base de datos, interfaz o añadir una nuevo funcionalidad o modificar una ya existente este cambio afectaría al conjunto de las partes y prácticamente tendríamos que reescribir parte del código sino el código entero. Para resolver este problema tenemos uno de los patrones de diseño más comunes, el patrón MVC(Modelo Vista Controlador) el cual separa la parte de persistencia de datos o modelo de datos de la interfaz del usuario, el controlador hace de puente entre los dos anteriores y guarda la lógica de negocio asociada a esos datos. La vista es la interfaz con el usuario, el modelo guarda los datos con los que se trabaja y el controlador modifica esos datos.

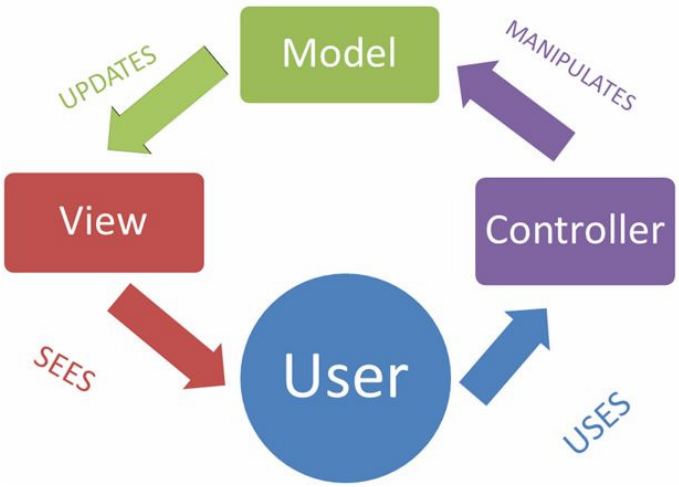

Figura 1.8: Diagrama MVC.

En 1990 el grupo *Gang of Four* publica el libro *Design Patterns*, en el que recogen los 23 patrones de diseño más comunes, en éste libro se recogen una serie de soluciones a problemas habituales en el diseño software. Los problemas que resuelven estos patrones de diseño han sido ampliamente estudiados por lo que podemos asegurarnos de que su uso va a ofrecernos una solución limpia y en la mayoría de los casos óptima sin necesidad de tener que reintentar la rueda. El uso compulsivo de patrones de diseño, sin embargo, es desaconsejable, los patrones son una gran ayuda en los casos en los que necesitamos de ellos, pero retorcer el código con la finalidad de introducir un patrón de diseño para resolver un problema que no necesitaba de ese patrón también podría ser negativo para nuestro programa.

1.2.5. BIBLIOTECAS

CUANDO nos enfrentamos al desarrollo de un sistema complejo que requiera el uso de muchas tecnologías disponemos de herramientas para facilitarnos el trabajo, por ejemplo, una parte importante en el desarrollo de videojuegos es el aspecto gráfico, si deseamos animar una imagen podríamos crearnos un programa que recoja las imágenes que queramos mostrar y crearnos un sistema que secuencie la muestra de estos dibujos creando así la animación, también podríamos implementar un sistema que gestione la física de nuestro juego, esto requeriría que diseñáramos desde cero un motor de juego que nos permita más adelante desarrollar nuestra aplicación. Otra opción es hacer uso de bibliotecas que nos den éstas herramientas ya desarrolladas que nos permitirá, por ejemplo, mediante el archivo que guarda las imágenes y una velocidad de animación que la biblioteca usará para automáticamente gestionar la animación de una forma transparente al programador. Otro ejemplo más sencillo es el de operaciones matemáticas incluidas en la mayoría de bibliotecas estándares de casi todos los lenguajes de programación que nos quitan el peso de tener que implementar ciertas acciones que son muy comunes, si ya está hecho y probada su calidad y buen funcionamiento no es necesario hacerlo otra vez.

Existen bibliotecas específicas para un gran número de utilidades, como por ejemplo la biblioteca *Spring* de Java que nos permite gestionar fácilmente la *inversión de control* en nuestro programa, de ésta biblioteca hablaremos más adelante. También existen bibliotecas que nos permiten acceder a ciertos recursos gráficos, trabajar con mapas, gestionar servicios web, hacer uso de determinadas aplicaciones, dar servicios de comunicación, etcétera...

1.2.6. UBUNTU 12.04

EN la búsqueda de un entorno de desarrollo que respondiese a nuestras necesidades debíamos buscar un sistema operativo que fuera estable y que nos diera libertad de configuración. El sistema que vamos a desarrollar requiere de un gran número de componentes ya sean bibliotecas o módulos adicionales, los cuales pueden inducir al sistema operativo a múltiples errores durante el desarrollo, si no disponemos de un sistema estable esto afectaría a la velocidad del desarrollo y al estado de ánimo del programador. También nos permite manipular la totalidad del sistema de forma que podemos adaptarlo al desarrollo de la forma que mejor nos convenga, sin contar la gran comunidad que tiene detrás al ser un sistema operativo de código libre lo cuál es un motivo para muchos programadores para implementar sus herramientas con compatibilidad para este sistema operativo lo que se convierte en una gran batería de herramientas a nuestro alcance a la hora de desarrollar un proyecto.

Figura 1.9: Logo Ubuntu.

1.2.7. Qt Creator

PARA el código del simulador que como ya explicamos anteriormente hace uso de C++ ya que nos proporciona la velocidad y robustez que necesitamos en técnicas de simulación en tiempo real hemos elegido Qt Creator como entorno de desarrollo integrado. Comenzamos usando *Eclipse for C/C++* pero a la hora de realizar la integración con *CMake*, del cual hablaremos más adelante, surgían muchos problemas y la velocidad de compilación era muy reducida, después de ser aconsejados por una compañera de trabajo decidimos cambiar a Qt Creator, el cual ofrece un entorno de desarrollo para C++ orientado a facilitar el uso de la biblioteca Qt de C++ y con muy buena integración con CMake. Al no depender de la máquina virtual de java la velocidad también aumentó, la interfaz es mucho más reducida que la de Eclipse y más sencilla e intuitiva aportando todas las opciones de configuración que necesitábamos.

También nos provee de una herramienta de diseño de interfaces de usuario con Qt sencilla y un debugger visual. Por todos esos motivos decidimos seguir el desarrollo con éste IDE en el caso de la programación en C++ ya que aumentó el rendimiento y redujo los fallos derivados del manejo del entorno de desarrollo.

Figura 1.10: Logo Qt Creator.

1.2.8. ECLIPSE

EL puesto de instructor está escrito en código Java ya que necesitábamos que fuera multiplataforma, para éste lenguaje de programación, el IDE más extendido es el de la propine empresa que lo diseñó, Sun Microsystems, cuyo nombre es Eclipse. Es un programa compuesto por un conjunto de herramientas de código abierto y multiplataforma orientado al desarrollo de entornos de desarrollo integrados, en nuestro caso, lo usaremos como entorno para programadores Java ya que tiene una gran integración con herramientas necesarias en nuestro proyecto como pueden ser *Maven*, *Junit*, Control de versiones, etcétera... Decidimos hacer uso de él debido al gran número de herramientas de soporte para el desarrollo Java.

Figura 1.11: Logo Eclipse.

1.2.9. CONTROL DE VERSIONES

CUSANDO nos disponemos a afrontar un proyecto software uno de los miedos más comunes es el de tener un código que funciona bien, realizar algún cambio y que todo deje de funcionar y haya que volver a repetir el trabajo ya realizado anteriormente. Éste miedo está ampliamente superado gracias al control de versiones. Ésta técnica nos permite hacer copias de seguridad periódicas de nuestro código en un repositorio externo, de forma que si en algún momento ocurre un accidente que haga que perdamos nuestro código dispongamos de un archivo con todas las versiones anteriores de nuestro programa pudiendo volver a un punto del tiempo en el que nuestro código funcionaba correctamente, evitando así tener que repetir la solución que ya teníamos implementada.

Otro problema muy normal entre los equipos de desarrollo se presenta cuando varios integrantes necesitan tocar el mismo código, sin el control de versiones, dos personas que estén implementando sobre el mismo fichero, sin saber qué está yaciendo el otro tendrán que unir, una vez finalizados sus respectivos cambios, los dos archivos en uno único en el que convivan las modificaciones que haya hecho cada uno, esto es un trabajo bastante tedioso y problemático ya que en muchas ocasiones el código de uno se verá afectado por el de otro, de manera que cuando el segundo vuelva a revisar su código se encuentre

que lo que ya funcionaba ahora no realiza correctamente su función y se le presente una dura tarea de debug para averiguar qué cambios han afectado a su código. El control de versiones nos permite crear varias ramas de desarrollo, de forma que cada programador trabaja en su rama y solo debe unir sus cambios al repositorio cuando se haya bajado el código actual, haya comprobado que no hay con conflictos entre el código principal y el suyo y su código esté probado y funcionando correctamente, de esta forma siempre tendremos una versión del código que funciona correctamente y eliminaremos la situación en que el código escrito por dos personas sobre el mismo fichero se vea en conflicto. Éstos motivos la convierten en una técnica indispensable a la hora de organizar un equipo de desarrollo software.

Los sistemas de control de versiones pueden ser clasificados según la arquitectura que utilizan para el almacenamiento del código:

- **Centralizados:** Hay un único repositorio que almacena todo el código y es gestionado por un administrador o grupo de administradores. Es más sencillo de gestionar ya que para realizar algún cambio como la creación de una nueva rama hay que pedir la aprobación del responsable del repositorio. Algunos ejemplos de repositorios de éste tipo son *Subversion* o *CVS*.

- **Distribuidos:** A diferencia de los anteriores, cada usuario tiene su propia copia del repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Existe también un repositorio principal que sirve para sincronizar el resto de repositorios locales. Entre los repositorios de éste tipo podemos encontrarnos *Mercurial* o *Git*.

  Ventajas de los sistemas distribuídos:

- 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 repositorio 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.

- Permite mantener repositorios centrales más limpios en el sentido de que un usuario puede decidir que 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. Por ejemplo es muy útil se pueden tener versiones inestables o en proceso de codificación o también tags propios del usuario.

- 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.

Para implementar ésta técnica en el proyecto SIMFORPAS decidimos hacer uso de las siguientes herramientas.

MERCURIAL

Haremos uso de un sistema de control de versiones distribuido, ya que su arquitectura de adecua mejor a nuestras necesidades como equipo de desarrollo. Usaremos Mercurial un sistema de control de versiones distribuido multiplataforma, originalmente escrito para trabajar en sistemas Linux como Ubuntu. Es un programa para línea de comandos y ofrece un protocolo de acceso mediante red muy eficiente que persigue reducir el tamaño de los datos así como la gestión de múltiples peticiones y conexiones. Su código se distribuye bajo licencia GNU GPL, lo que lo clasifica como Software Libre.

Figura 1.12: Logo Mercurial.

BITBUCKET

Bitbucket es un servicio web de alojamiento de código que use sistema de control de versiones Mercurial y Git. Ofrece alojamiento gratuito u opcional de pago, permitiendo éste segundo un mayor número de participantes en el repositorio. También nos da la opción de crear repositorios tanto públicos como privados y el manejo de funciones propias de Mercurial como la creación de *Forks* que nos permitan clocar un repositorio en un punto determinado y desarrollar en él, mientras el administrador se encarga de gestionar la adición de los cambios que aporte ese foro al repositorio principal, asegurando que el código principal siempre va a gozar de buena salud.

Figura 1.13: Logo Bitbucket.

TORTOISEHG

Es un cliente de escritorio para Ubuntu de control de versiones Mercurial que nos permite interactuar entre los archivos locales de nuestro código y el repositorio remoto, de forma que nos gestiona nuestro repositorio local y nos permite tenerlo sincronizado con el remoto comprobando si ha habido cambios, pudiendo actualizar nuestro repositorio, el remoto y todas las funciones características del control de versiones como los push, pull, así como una herramienta de solución de conflictos en el código.

1.2.10. PRUEBAS UNITARIAS

EL código escrito por un programador está siempre sujeto a errores inherentes a la condición humana, por éste motivo siempre debemos probar un código después de haberlo implementado. Si tratásemos de probar el código después de haber acabado un proyecto completo seguramente nos encontraríamos con una gran cantidad de fallos difícilmente localizables y requeriría mucho tiempo descubrir qué es lo que funciona mal y cuál es su solución. Una buena técnica para evitar esto es realizar pruebas al código parte a parte, otorgándole a cada clase su propia prueba de forma unitaria,

Entre las ventajas del uso de las pruebas unitarias se encuentran:

Encuentra problemas a tiempo: En TDD, como ya explicamos anteriormente, se intenta definir primero la funcionalidad de una clase escribiendo la condición que tiene que cumplir para que su funcionamiento se de por bueno a través de una prueba unitaria, de esta forma tendremos conocimiento de cuando falla el código inmediatamente después de haberlo escrito. Por tanto, las pruebas unitarias alertan a los desarrolladores de un problema antes de que el producto salga al mercado.

Facilita los cambios: Las pruebas unitarias permiten al programador realizar refactorizaciones comprobando que cada parte individual del código sigue cumpliendo su función. El procedimiento requiere que se escriban pruebas para cada método del programa por lo que cualquier lugar en el que haya ocurrido un error será rápidamente reconocible ya que la prueba nos indicará dónde se ha producido el fallo.

Simplifica la integración: Ayudan a la integración entre diferentes unidades del sistema haciendo más sencillas las pruebas de varias clases en conjunto ya que cuando se vayan a hacer estas no tenemos que probar los fallos referentes al funcionamiento individual de las clases. En las pruebas unitarias de una clase no debe intervenir ninguna otra, esas casuísticas se tratan en las pruebas de integración.

Aporta documentación: Las pruebas aportan documentación práctica sobre el código ya que para implementarlos hemos tenido que utilizar esas clases de forma correcta y queda plasmada en ellos cómo quiere el desarrollado que se use esa clase, de forma que cualquiera que quiera usar ese código o conocer su funcionamiento puede acudir a las pruebas para ver de primera mano qué están haciendo esas clases.

Mejora el diseño: Cuando el software es desarrollado con guiado mediante pruebas, la combinación entre escribir los test pare definir las interfaces con la refactorización del código después de que se valide la prueba correctamente hace que la estructura del código adquiera una forma adecuada y optimizada mejorando así la calidad del mismo y favoreciendo las futuras modificaciones.

GOOGLE TEST

Para la gestión de las pruebas unitarias en los proyectos que estén escritos en el lenguaje de programación C++ usaremos la biblioteca de pruebas de Google *Google Test*. Ésta librería nos ofrece una amplia gama de herramientas para probar nuestro código, dándonos la oportunidad de ejecutar los test por separados o todos a la vez, lo que hace que cubra las necesidades de un amplio espectro de perfiles de desarrolladores.

Google test funciona separando cada test de manera que unos no interfieran sobre la ejecución de los demás lo que nos proporciona fiabilidad y robustez en nuestras pruebas. También ejecuta todas las pruebas definidas en el proyecto de manera que no necesitamos listarlas de manera especial para indicarle qué test tenemos. Otra característica es la búsqueda de ofrecer el mayor número de información sobre el código que se está probando de manera que no para en el primer error que encuentra sino que sigue con las siguientes pruebas para darnos una visión más global del estado de nuestro proyecto. También nos permite reutilizar y compartir código e instancias de objetos entre los diferentes tests, que gestionamos con los *set-ups* y *tear-downs* de manera que los tests serán más rápidos.

JUNIT

En cuanto a la parte del proyecto realizada en código Java haremos uso de las muy extendidas bibliotecas de pruebas para Java *JUnit*. Son un conjunto de clases que nos permiten realizar la ejecución de clases Java de manera controlada para poder evaluar si el funcionamiento de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado, al igual que la biblioteca de pruebas para C++.

Existe integración para Eclipse de JUnit, el cual puede ser obtenido a través del *Market Place* incluido en el mismo programa, el cual nos ofrece un entorno gráfico para la visualización de los resultados de las pruebas. Los tests se definen mediante anotaciones, una técnica muy usada en Java que nos permite añadir metadatos al código fuente para la aplicación en tiempo de ejecución de matera que nos libera de tener que usar una biblioteca y añadir más líneas a nuestro código.

1.2.11. MOCKS

LAS pruebas unitarias requieren que solamente se valide una única clase de forma que no afecte a la prueba ninguna otra. Pero en múltiples ocasiones nos encontramos con que la clase que queremos someter a pruebas depende de una clase externa y su funcionamiento está ligado a la interacción con esta otra clase. Para solucionar esta paradoja se nos ofrece una herramienta como son los objetos *Mock*, estos objetos tienen la función de comportarse como una imitación de la clase real. Si por ejemplo necesitamos hacer uso de un método que realiza la multiplicación entre dos números y cuyo resultado usaremos para realizar alguna acción en la clase que estamos probando, la implementación de estas

clase multiplicación nos ofrecerá un método que nos devuelva un número, no nos interesa que realice ninguna operación sino que nos devuelva lo que necesitamos para ejecutar nuestro código. Si usáramos la clase real estaríamos condicionados a suponer el buen funcionamiento de esta clase, de esta forma, no dependemos de ninguna forma de la clase, sino que nos ceñiremos a probar que la clase que estamos probando funciona correctamente. El si las clases de las que dependemos funcionan bien o no será responsabilidad de los test de esas clases respectivamente.

GMock

Al igual que con los entornos de pruebas unitarias, tenemos bibliotecas que nos ayudan con la generación de objetos mock para que no tengamos que preocuparnos de generar todas estas clases adicionales nosotros mismos. Para pruebas unitarias con Google Test usaremos *GMock*, una biblioteca de Google que nos permite generar este tipo de objetos rápidamente, de manera sencilla y con una amplia gana de configuraciones entre las que podemos probar el número de veces que se espera llamar a ése método y qué objeto queremos que devuelva en cada una de las llamadas así como definir con qué parámetros de entrada se debe invocar, todo esto integrado con el entorno de pruebas de Google Test. Existe una amplia documentación sobre ésta tecnología incluido el *CookBook para GMock* de Google donde se explican detalladamente y con ejemplos cada una de las características de esta biblioteca.

JMock

Para JUnit también tenemos una biblioteca similar que es *jMock*, jMock nos ofrece las mismas características que ya explicamos anteriormente con GMock, tiene integración con JUnit y al estar diseñado con anotaciones también permite que nuestro código quede más limpio, rápido y sea más fácil de usar. Nos permite especificar que tipo de interacción existe entre nuestros objetos reduciendo la fragilidad de nuestro código.

1.2.12. CMake

CMake es una familia de herramientas diseñada para construir, probar y empaquetar software. *CMake* se utiliza para controlar el proceso de compilación del software usando ficheros de configuración sencillos e independientes de la plataforma. El proceso de construcción se controla creando uno o más ficheros CMakeLists.txt en cada directorio (incluyendo subdirectorios). Cada CMakeLists.txt consiste en uno o más comandos. Cada comando tiene la forma COMANDO (argumentos...) donde COMANDO es el nombre del comando, y argumentos es una lista de argumentos separados por espacios. CMake provee comandos predefinidos y definidos por el usuario. Entre las principales funcionalidades CMake nos ofrece un análisis automático de dependencias.

Debido a que en nuestro proyecto hacemos uso de subproyectos de configuración similar como pueden ser el núcleo del simulador, la primera versión del proyecto que

gestiona el modelo, etc... CMake nos aporta una gran ayuda a la hora de iniciar un nuevo proyecto con características similares a alguno que ya hayamos creado ya que la configuración la gestiona CMake y no tenemos que preocuparnos de configurar las dependencias ni otros tipos de configuraciones.

Ejemplo de CMakeLists.txt

```
if (${UNIX})
  set (DESKTOP $ENV{HOME})
else()
  set (DESKTOP $ENV{USERPROFILE}/Desktop)
endif()

set  (PRJ      ${DESKTOP}/common/svn )
set  (FILELIST ${PRJ}/src/source.txt )

message(STATUS "CMAKE_GENERATOR : ${CMAKE_GENERATOR}")
message(STATUS "DESKTOP         : ${DESKTOP}")
message(STATUS "PRJ             : ${PRJ}")
message(STATUS "FILELIST        : ${FILELIST}")
message(STATUS "SYSTEM_NAME     : ${CMAKE_SYSTEM_NAME}")

project(project_name)

include_directories(
  ${PRJ}/src
  ${PRJ}/includes
)

# Load SRC Variable from file
file(READ ${FILELIST} SRC)
string(REGEX REPLACE  "#.*$"   ""   SRC  ${SRC})
string(REPLACE        "\n"     ";" SRC  ${SRC})

add_executable(${PROJECT_NAME} ${SRC} )

foreach (f ${SRC})
  set_source_files_properties(${f} PROPERTIES LANGUAGE       CXX)
endforeach(f)

if (${WIN32})
  link_directories(
  )

  add_definitions(
    -DDEFINE1
  )

  target_link_libraries(
    ${PROJECT_NAME}
    wsock32.lib
  )
endif()
```

Figura 1.14: Ejemplo CMakeLists.txt.

1.2.13. MAVEN

MAVEN, al igual que CMake, es una herramienta software para la creación de proyectos, en este caso para proyectos Java con un modelo de configuración muy simple basado en *XML*. Maven utiliza un Project Object Model(POM) 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. Una característica clave de Maven es que está listo para usar en red. 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 en Java, de Apache y otras organizaciones y desarrolladores. Maven provee soporte no sólo para obtener archivos de su repositorio, sino también para subir artefactos al repositorio al final de la construcción de la aplicación, dejándola al acceso de todos los usuarios. Una caché local de artefactos actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

A la hora de programar el puesto de instructor necesitaremos que sea una aplicación web y aprovechar las múltiples bibliotecas de utilidad que nos ofrece Java como Spring, Struts, Hibernate, etc... Para gestionar todas estas configuraciones necesarias haremos uso de Maven.

Ejemplo de POM.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <!-- POM Relationships -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <dependencies>...</dependencies>
  <modules>...</modules>

  <!-- Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <developers>...</developers>
  <contributors>...</contributors>
  <organization>...</organization>

  <!-- Build Settings -->
  <packaging>...</packaging>
  <properties>...</properties>
  <build>...</build>
  <reporting>...</reporting>

<!-- Build Environment -->

<!-- Environment Information -->


  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>


<!-- Maven Environment -->


  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>
</project>
```

Figura 1.15: Ejemplo POM.xml.