



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

I	Introducción	1
1.	Introducción	3
1.1.	Motivación	6
1.2.	Objetivos del proyecto	7
1.2.1.	Objetivos orientados a la metodología	7
1.2.2.	Objetivos orientados a la técnica	7
1.2.3.	Objetivos personales	8
1.3.	Estructura del documento	9
II	Conceptos básicos	11
2.	Metodologías usadas	13
2.1.	Metodologías ágiles	13
2.1.1.	Manifiesto ágil	14
2.1.2.	Desarrollo iterativo incremental con Scrum	15
2.1.3.	Test Driven Development	18
2.1.4.	Pair programming	19
2.2.	Tecnologías	20
2.2.1.	Programación orientada a objetos	20
2.2.2.	C++	24
2.2.3.	Java	25
2.2.4.	Patrones de diseño	26
2.2.5.	Bibliotecas	27
2.2.6.	Ubuntu 12.04	28
2.2.7.	Qt Creator	29
2.2.8.	Eclipse	30
2.2.9.	Control de versiones	30
2.2.10.	Pruebas unitarias	34
2.2.11.	Mocks	35
2.2.12.	CMake	36
2.2.13.	Maven	38
2.2.14.	ANIMO DDS	40
2.2.15.	Spring	41

2.2.16. Struts 2	43
2.2.17. Hibernate	45
2.2.18. Valgrind	46
2.2.19. Mavlink	48
2.2.20. GCS	50
2.2.21. Locomove	52
2.2.22. Autopiloto	53
III Sistema a desarrollar	55
3. Planificación inicial	57
3.1. Lanzamiento del proyecto	57
3.1.1. Definición de roles	57
3.1.2. Historias de usuario	58
4. Iteraciones: Sprints de desarrollo	63
4.1. Sprint 1: Set Up del proyecto	63
4.1.1. Módulos del sistema	63
4.1.2. Estructura del sistema	64
4.1.3. Diagramas de secuencia	65
4.1.4. Diagrama de comunicaciones	70
4.1.5. Entradas y salidas de los módulos	72
4.1.6. Definición de interfaces	74
4.1.7. Diagrama de burndown	76
4.1.8. Diagrama de evolución	77
4.2. Sprint 2: Creación de la capa de comunicación	78
4.2.1. Set up del entorno de desarrollo	78
4.2.2. Implementación de la capa de comunicación	80
4.2.3. Implementación del modelo simple	81
4.2.4. Diagrama de burndown	84
4.2.5. Diagrama de evolución	85
4.3. Sprint 3: Creación del SIMCORE	86
4.3.1. Diagrama de burndown	86
4.4. Sprint 4: Implementación del modelo de Locomove	87
4.4.1. Diagrama de burndown	87
4.5. Sprint 5: Gestión de carga de la misión	88
4.5.1. Diagrama de burndown	88
4.6. Sprint 6: Puesto de instructor	89
4.6.1. Diagrama de burndown	89
4.7. Sprint 7: Primera versión y cambios en el objetivo	90
4.7.1. Diagrama de burndown	90
4.8. Sprint 8: Cambios en el diseño	91
4.8.1. Diagrama de burndown	91

IV Conclusiones 93

V Apéndices 95

Índice de figuras

1.1.	Soporte aéreo en el control de incendios.	3
1.2.	UAV tipo Predator en vuelo.	4
1.3.	GCS usada para el UAV Predator.	5
2.1.	Manifiesto ágil.	14
2.2.	Ciclo metodología Scrum.	17
2.3.	Ciclo TDD.	18
2.4.	Pair programming.	19
2.5.	Pilares de la POO.	22
2.6.	Ilustración c++11.	24
2.7.	Logo Java.	25
2.8.	Diagrama MVC.	26
2.9.	Logo Ubuntu.	28
2.10.	Logo Qt Creator.	29
2.11.	Logo Eclipse.	30
2.12.	Logo Mercurial.	32
2.13.	Logo Bitbucket.	33
2.14.	Ejemplo CMakeLists.txt.	37
2.15.	Ejemplo POM.xml.	39
2.16.	Logo de RTI.	40
2.17.	Logo Spring.	42
2.18.	Logo Struts 2.	44
2.19.	Logo Hibernate.	45
2.20.	Logo Valgrind.	46
2.21.	Resltado Valgrind.	47
2.22.	Logo Mavlink.	48
2.23.	Protocolo de escritura de misión en Mavlink.	49
2.24.	QGroundControl, gcs de Mavlink.	50
2.25.	GCA de Catec.	51
2.26.	UAV de Catec, Locomove.	52
2.27.	Autopiloto integrado en el Locomove.	54
3.1.	Ejemplo del documento de backlog.	62
4.1.	Arquitectura del sistema, con las relaciones físicas entre subsistemas. . . .	64
4.2.	Diagrama de secuencia de la ejecución de una misión.	66

4.3. Diagrama de secuencia de la carga de una misión en el simulador y el control.	67
4.4. Diagrama de secuencia con los comandos desde el puesto de instructor para iniciar, parar o pausar una ejecución.	68
4.5. Diagrama de secuencia para la introducción de fallos.	69
4.6. Diagrama de secuencia para hacer cambios en la misión.	70
4.7. Diagrama de las relaciones a la hora de comunicarse de los distintos módulos.	71
4.8. Definición de interfaces.	75
4.9. Diagrama de velocidad para el primer sprint.	76
4.10. Diagrama de evolución para el primer sprint.	77
4.11. Diagrama de velocidad para el segundo sprint.	84
4.12. Diagrama de evolución para el segundo sprint.	85

Parte I

Introducción

Capítulo 1. Introducción

AUNQUE el concepto de aviones no tripulados o UAV's (Unmanned Aerial Vehicles) es bastante antiguo, puesto que se empezaron a usar durante 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 1.1: 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 de 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.

Para ilustrar la necesidad de la implementación de un sistema de evaluación de pilotos tendremos en cuenta, entre otros factores los múltiples accidentes que ha habido en los últimos años, la mayoría de ellos debido a descuidos o a la falta de experiencia de los operadores de GCS. Uno de los casos que pueden darse es el de la falta de pruebas del sistema por parte de los desarrolladores la cual puede llevar a que los operadores puedan mandar comandos erróneos al avión provocando así un accidente, como pasó en el accidente del UAV tipo Predator en Septiembre del año 2000 cuando un piloto por equivocación liberó la memoria del UAV provocando un corte en las comunicaciones el cual no pudieron recuperar antes de que el avión se estrellara (Datos de la FAA, Federal Aviation Administration, de Estados Unidos durante una conferencia sobre seguridad durante vuelos de UAV dada por Kevin W. Williams), si se hubiera podido probar el software de forma segura sin poner en peligro el avión éste error podría haberse evitado.



Figura 1.2: UAV tipo Predator en vuelo.

Otro ejemplo de error común que podemos encontrar en la información ofrecida por la FAA es el de la falta de experiencia de los operadores controlando los mandos, uno de los ejemplos de este tipo de error fue el accidente ocurrido en abril de 2006,

el dispositivo de control de tierra constaba de dos estaciones iguales, una destinada al manejo de la aeronave y otra al de la cámara. En un momento dado se detectó un error en la comunicación entre el avión y la GCS y decidieron cambiar la función de las estaciones entre sí, pero la posición en la que se encontraba la estación que controlaba la cámara, concretamente una palanca que cerraba el diafragma de la misma, ordenaba al controlar el UAV que se parara el motor, de forma que en el momento del cambio perdieron el control de la aeronave estrellándola.



Figura 1.3: GCS usada para el UAV Predator.

En este accidente los pilotos tenían poca experiencia de vuelo y ningún certificado acreditativo para poder manejar este tipo de instrumental, la alarma del indicador de que algo iba mal no era específica para ese error por lo que no se tuvo en cuenta. Un error tan importante como la no implantación de un protocolo de comprobaciones al iniciar una nueva estación de control para volar un UAV podría haber sido evitado si se incluyera en un curso obligatorio para operadores de estos dispositivos.

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.

1.1. 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 expectativas 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.

1.2. 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.

1.2.1. 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.

1.2.2. 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.

1.2.3. 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.

1.3. ESTRUCTURA DEL DOCUMENTO

Este 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.

APÉNDICES: 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 2. 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.

2.1. 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.

2.1.1. MANIFIESTO ÁGIL



Figura 2.1: 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.

2.1.2. 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.

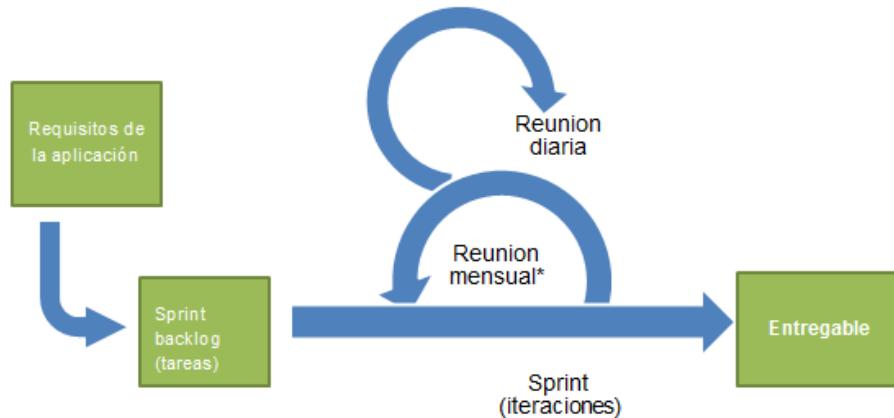


Figura 2.2: Ciclo metodología Scrum.

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.

2.1.3. 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 tendremos 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.

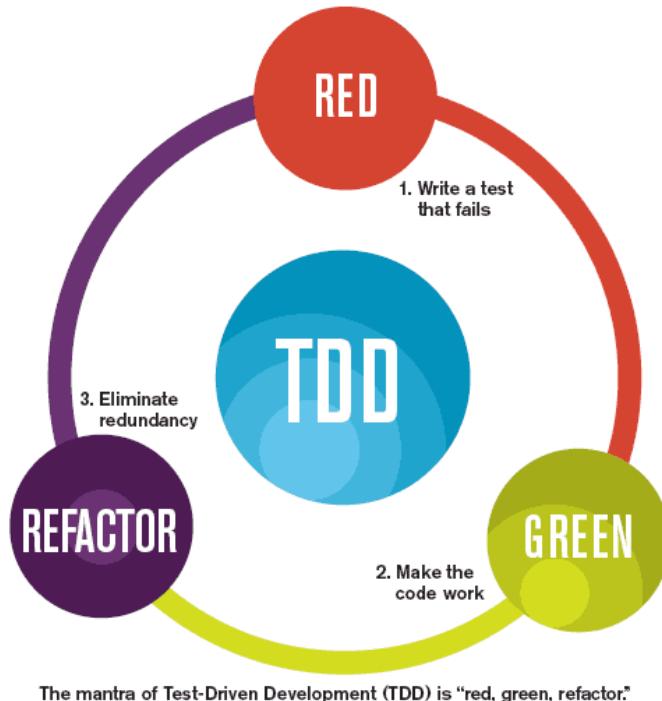


Figura 2.3: Ciclo TDD.

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.

2.1.4. 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 2.4: Pair programming.

2.2. 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.

2.2.1. 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.^bstracto que puede realizar trabajo, informar y cambiar su estado, y 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, 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.

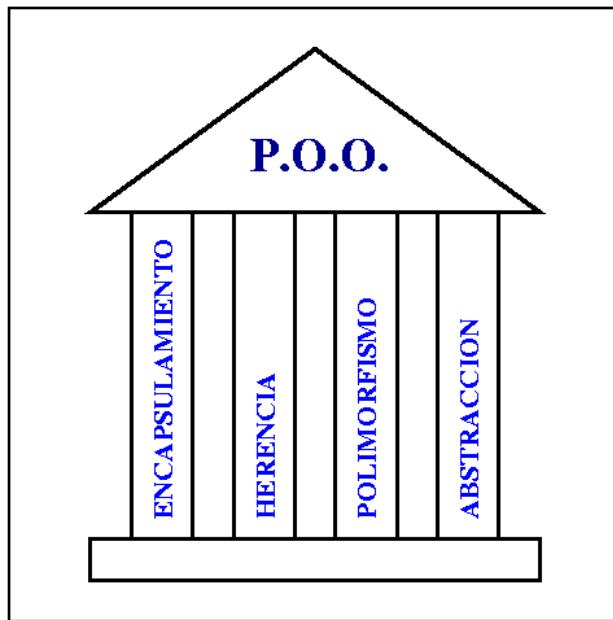


Figura 2.5: Pilares de la POO.

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.

R 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:

- **Single 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.
- **Open/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.
- **Liskov 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.
- **Interface 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.
- **Dependency inversion 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.

2.2.2.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 2.6: 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 multiplataforma 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.

2.2.3. 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 2.7: 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.

2.2.4. 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 nueva 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.

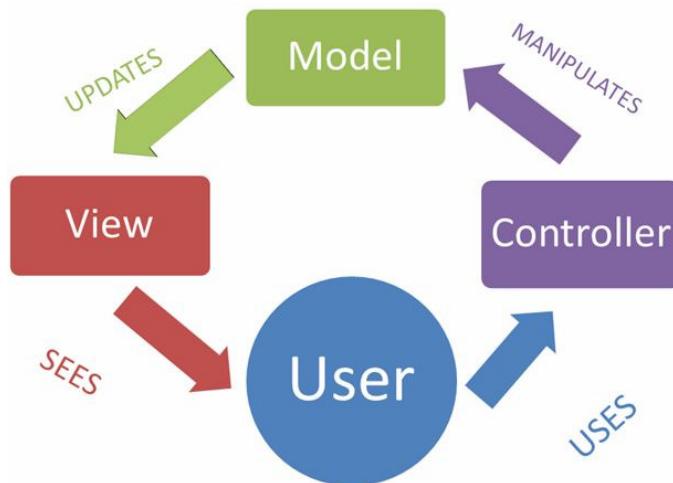


Figura 2.8: Diagrama MVC.

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.

2.2.5. 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...

2.2.6. 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 2.9: Logo Ubuntu.

2.2.7. 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 2.10: Logo Qt Creator.

2.2.8.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 propia 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 2.11: Logo Eclipse.

2.2.9.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 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 2.12: 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 colocar 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 2.13: 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.

2.2.10. PRUEBAS UNITARIAS

El código escrito por un programador está siempre sujeto a errores inherentes a la condición humana, por este 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 sea 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 desarrollador 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 para 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 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 manera que nos libera de tener que usar una biblioteca y añadir más líneas a nuestro código.

2.2.11. MOCKS

AS 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 gama de configuraciones entre las que podemos probar el número de veces que se espera llamar a ese 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.

2.2.12. 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}
    ws2_32.lib
  )
endif()

```

Figura 2.14: Ejemplo CMakeLists.txt.

2.2.13. 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.

Otra aplicación interesante de Maven es la posibilidad de crear *arquetipos*. Los arquetipos se pueden considerar como plantillas de configuración para un nuevo proyecto, de forma que una vez configurado nuestro proyecto con Maven podemos guardar esas características, librerías usadas, estructura del proyecto, etc... y guardarla de manera que podamos usarla para la generación de un proyecto de características similares.

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 2.15: Ejemplo POM.xml.

2.2.14. ANIMO DDS

PARA solucionar el problema de intercambio de datos entre los distintos módulos que compondrán nuestro sistema usaremos un middleware de comunicaciones basado en DDS RTI (Data DIstribution Service de la compañía Real Time Innovations), una tecnología que nos permite conectar varios sistemas entre sí de forma distribuida, de esta forma disponemos de un medio de intercambio de información sencillo, escalable y sin necesidad de preocuparnos de conectividad a bajo nivel. El uso de DDS RTI también nos garantiza que las comunicaciones se realizarán en tiempo real y sin perder ningún dato, una funcionalidad indispensable cuando estamos trabajando con dispositivos críticos como los UAV en los que cualquier pérdida de algún dato puede introducir fallos en el sistema. Al estar basado en el patrón publicador-subscriptor y nuestro entorno estar estructurado en módulos podemos ampliar la funcionalidad como queramos sin necesidad de afectar al resto de módulos. Por ejemplo, si la aeronave está publicando su posición podemos crear una aplicación de monitorización que se subscriba a este dato sin afectar al resto de componentes.

ANIMO es un framework envoltura de DDS RTI que nos ofrece una serie de herramientas que facilitan el uso de estas librerías ya que simplifica muchas de las configuraciones previas necesarias para realizar una comunicación DDS, también permite mediante un sencillo archivo de configuración crear los archivos necesarios para un nuevo tipo de dato automáticamente mediante un generador de códigos. Gracias a las calidades de servicio de DDS RTI podemos definir muchas variables en la comunicación de los datos, desde enviar un primer dato cuando se inicie la comunicación para un cierto tipo con el fin de comprobar que todo funciona bien hasta decidir el tiempo que tiene que pasar para que un dato sea perdido o incluso establecer un identificador para el destinatario, de esta forma nos aseguramos que la comunicación es fiable y segura.



Figura 2.16: Logo de RTI.

2.2.15. SPRING

En la realización de nuestro proyecto usaremos un patrón de diseño muy extendido sobre la programación orientada a objetos que es la *Inyección de dependencias*, este patrón consiste en suministrar objetos a una clase en lugar de que sea esa clase la que los crea. De esta forma se controla mejor el acceso a los componentes y no tenemos que encapsular objetos dentro de otros objetos para mantener el acceso de forma que se nos ensucie el código y se ponga en riesgo su legibilidad y escalabilidad. En lugar de eso una clase es la encargada de hacer de contenedor de todos los objetos necesarios durante la ejecución del programa y desde esa clase se tomaran las instancias que se pasarán como parámetros de entrada de los constructores de las clases que hagan uso de ellos.

Para implementar este patrón usaremos el framework open source Spring que nos provee una serie de librerías para Java para controlar de forma sencilla la inyección de dependencias en nuestra aplicación. Debido al éxito y las múltiples ampliaciones que ha tenido Spring, éste también nos provee de otras herramientas útiles a la hora de programar como un módulo de acceso a datos para trabajar con bases de datos relacionales y no relacionales, un módulo de aplicación del modelo vista controlador, un módulo de testing, etcétera...

Spring nos ofrece varias alternativas a la hora de configurar la creación de objetos o *beans*, una de ellas es mediante un archivo XML donde se define la ruta a las clases que van a ser beans y donde se indica cuantos beans se van a crear. Otra forma es mediante el uso de *anotaciones*, definiendo directamente una anotación en el código de la clase podemos controlar más fácilmente el uso de estos beans.

VENTAJAS DE LA INYECCIÓN DE DEPENDENCIAS

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 buenas 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 representen datos de prueba, de servicios o cualquier dependencia que necesite el componente que estamos probando.

MÓDULOS DE SPRING

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 contenedor para la Inversión de Control les proporciona automáticamente estas dependencias.

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 bean como servicio o para acceder desde un bean 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.



Figura 2.17: Logo Spring.

2.2.16.STRUTS 2

YA introdujimos anteriormente el modelo vista controlador o MVC y los beneficios de usarlo en nuestra aplicación. Para su implementación se hará uso de Struts 2, una herramienta de soporte para el desarrollo de aplicaciones web en Java de la compañía Apache. Es de código abierto y nos permite simplificar la conexión entre nuestro código java y la parte de la vista web de la aplicación.

El núcleo de Struts es un filtro conocido como *FilterDispatcher* el cual nos permite ejecutar los *actions* que son los métodos lanzados por peticiones web, comenzar la ejecución de interceptores y gestionar la memoria para evitar fugas.

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

INTERCEPTORS

Los Interceptores son clases que siguen el Patrón de Diseño Interceptor. 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 Interceptores 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.

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.

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 success, si algo sale mal lo enviaremos al objeto tipo Result error, o si no tiene permisos lo enviaremos al objeto tipo Result denied.



Figura 2.18: Logo Struts 2.

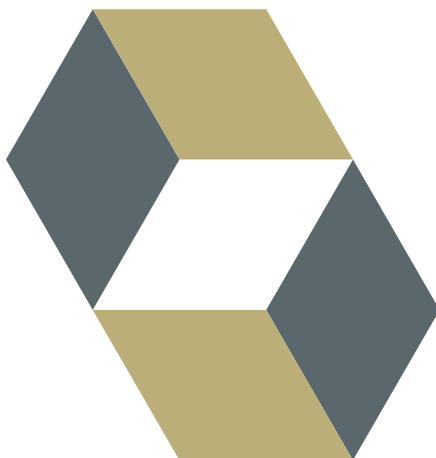
Para evitar tener que realizar la configuración de Struts mediante el archivo XML, usaremos un plugin de Apache para Struts llamado *Convention Plugin* el cual nos ofrece una forma sencilla de determinar en el mismo código como realizar las conexiones entre vistas y modelos. Usando la nomenclatura de las clases que creamos podemos determinar que parte del modelo se lanzará cuando se haga una llamada desde la vista. Así si por ejemplo tenemos una acción llamada NuestraAction.java, cuando cargamos la url con nombre nuestra-action.jsp se hará una llamada a la acción del mismo nombre, así podemos controlar las conexiones sin necesidad de usar ni archivos de configuración ni anotaciones.

2.2.17. HIBERNATE

EN vistas de la futura necesidad de almacenar datos de forma persistente, ya sean misiones a realizar por un alumno, log de misiones realizadas, calificaciones, datos de alumnos, modelos de distintos UAVS, etcétera, implementaremos un sistema que nos permitan trabajar con bases de datos de forma sencilla y escalable. Para ello requeriremos de un sistema que nos permita hacer una implementación del patrón de diseño *DAO* o Data Access Object. Al diseñar una base de datos de forma tradicional necesitamos adaptar nuestro código a los datos que vayan a guardarse en esa base de datos, y dependeremos de la forma en que estén guardados, el tipo de base de datos y cómo se nos presenta esos datos de manera que si en algún momento se cambia algo en la base de datos será muy complicado reutilizar el código que teníamos y necesitaremos realizar muchos cambios para adaptar la nueva base de datos.

Para evitar esto disponemos del patrón DAO el cual consiste en disponer de unos objetos que nos hagan de puente entre nuestro programa y la base de datos, estos objetos se encargarán de realizar las conexiones con la base de datos y crear a partir de la petición que hagamos a esa base de datos de alguna entidad el objeto java que usaremos en la ejecución. Si en algún momento necesitamos modificar la base de datos solo tendríamos que cambiar estos objetos por unos que se adecuen a los nuevos requerimientos de forma que no afecta al resto de la aplicación.

Para implementar esta solución usaremos Hibernate en su versión 4.3.4, Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java, mediante anotaciones sobre las clases que queremos guardar en la base de datos podemos autogenerar los objetos DAO e incluso crear las tablas en la base de datos, todo de forma sencilla y automática.



HIBERNATE

Figura 2.19: Logo Hibernate.

2.2.18. VALGRIND

A la hora de hacer las pruebas es conveniente también asegurarnos de que nuestro código no tiene ninguna fuga de memoria que pudiera hacer que surgieran fallos o incluso que el sistema se colapsara, algunas pistas de que hay errores de memorias son la disminución del rendimiento, que el programa vaya lento o que haya objetos devolviéndonos datos erróneos. Para hacer una comprobación del estado de la memoria usada por nuestro programa usaremos *Valgrind*, un conjunto de herramientas de software libre que nos ayudará en este proceso.

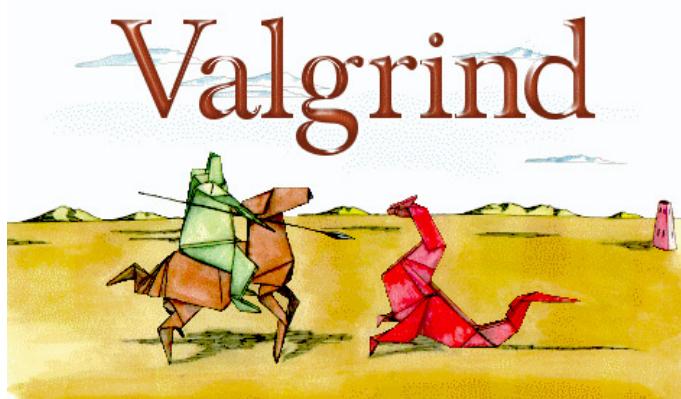


Figura 2.20: Logo Valgrind.

La herramienta más usada es *Memcheck*, que permite realizar un seguimiento del uso de la memoria y detectar múltiples errores como uso de memoria no inicializada, lectura o escritura de memoria que ha sido previamente liberada, lectura o escritura fuera de los límites del bloque de memoria dinámica, fugas de memoria, etcétera.

El uso de Valgrind sobre nuestro programa relentiza considerablemente el rendimiento de éste, se ejecuta mucho más lento, por eso es conveniente usarlo solo en momentos críticos, durante una búsqueda concreta de estos errores por ejemplo.

Otras herramientas incluidas por Valgrind son Massif que mide el rendimiento de la porción de memoria total, Helgrind que detecta condiciones de carrera cuando varios procesos intentan acceder a la vez al mismo recurso o Cachegrind que mide el rendimiento de la caché durante la ejecución.

```
==47243== HEAP SUMMARY:
==47243==     in use at exit: 1,344 bytes in 8 blocks
==47243==   total heap usage: 8 allocs, 0 frees, 1,344 bytes allocated
==47243==
==47243== 1,024 bytes in 1 blocks are definitely lost in loss record 4 of 4
==47243==    at 0xF656: malloc (vg_replace_malloc.c:195)
==47243==    by 0x1F66: main (in ./a.out)
==47243==
==47243== LEAK SUMMARY:
==47243==  definitely lost: 1,024 bytes in 1 blocks
==47243==  indirectly lost: 0 bytes in 0 blocks
==47243==  possibly lost: 0 bytes in 0 blocks
==47243==  still reachable: 320 bytes in 7 blocks
==47243==    suppressed: 0 bytes in 0 blocks
==47243== Reachable blocks (those to which a pointer was found) are not shown.
==47243== To see them, rerun with: --leak-check=full --show-reachable=yes
==47243==
```

Figura 2.21: Resltado Valgrind.

2.2.19.MAVLINK

PARA realizar las comunicaciones entre los distintos módulos que compondrán el simulador usaremos el protocolo de comunicaciones Mavlink que es el que se usará realmente en un vuelo con el UAV para el que realizaremos el entorno de simulación, es un protocolo de comunicaciones simple para aeronaves no tripuladas de menos de 25 kilogramos, se sopesó utilizar el protocolo definido por STANAG, el acuerdo de normalización de la OTAN, pero éste protocolo es más amplio y detallado y hubiéramos gastado demasiado tiempo en la implementación del protocolo que diseñando la funcionalidad del simulador que era lo que nos interesaba, además los componentes que vamos a usar, tanto el UAV como la GCS ya tienen implementados el protocolo Mavlink, por tanto nos aprovecharemos de esto para realizar la primera aproximación.



Figura 2.22: Logo Mavlink.

El protocolo Mavlink define una serie de interfaces a usar en las comunicaciones entre la estación de control de tierra y la aeronave y unos protocolos de utilidad durante el revuelo y la misión del avión como pueden ser carga de misión, órdenes, mensajes de error, monitorización de sensores, etcétera.

Cada mensaje del protocolo va a su vez encapsulado dentro de un paquete que guarda datos del envío como el número de secuencia de ese paquete, el sistema y el componente al que va destinado, un flag de control de envío, un flag de inicio de transmisión...

Es un protocolo muy extendido y tiene soporte para muchas plataformas, tanto software como hardware, se puede usar con los autopilotos Parrot AR.Drone, ArduPilot, PX4FMU, pxIMU, SmartAP, MatrixPilot, Armazilla 10dM3UOP88 y software como AndroidPilot, APM Planner 2.0, DroidPlanner, MAVProxy, MissionPlanner, QGroundControl(implementado por la compañía creadora de Mavlink). Éstos son los oficiales, mucha gente lo ha usado como nosotros para sus propios autopilotos a parte de los aquí redactados y tiene una gran comunidad detrás.

En nuestro sistema usaremos a parte de los mensajes de comando, monitorización y errores la secuencia definida por Mavlink para la escritura de la misión en el avión cuyo funcionamiento describiré a continuación para ilustrar el uso de los mensajes de este protocolo.

ESCRITURA DE MISIÓN CON EL PROTOCOLO MAVLINK

Desde la GCS se envía un primer mensaje hacia el UAV MISSION-COUNT, mensaje que pide iniciar una escritura de misión y que lleva un campo con el número de waypoints que va a tener esa misión, luego el UAV, cuando recibe éste mensaje, responde con un MISSION-REQUEST con el número de waypoint que quiere pedir, en el primer caso será el waypoint con número de secuencia 0, al llegar ese mensaje a la GCS ésta sabrá que puede enviar el primer waypoint e inicia un contador de tiempo, si en un tiempo determinado no ha recibido el siguiente MISSION-REQUEST desde el UAV significa que ha habido algún problema en el envío por lo que tendremos que enviarlo otra vez, cuando ha llegado el mensaje al UAV éste vuelve a enviar un nuevo MISSION-REQUEST pidiendo el siguiente waypoint, cuando todos los waypoints han sido enviados el UAV envía un mensaje MISSION-ACK que informa a la GCS que ha llegado el último waypoint correctamente y la misión ha sido escrita entera en la aeronave. Éste protocolo se ilustra en la siguiente figura.

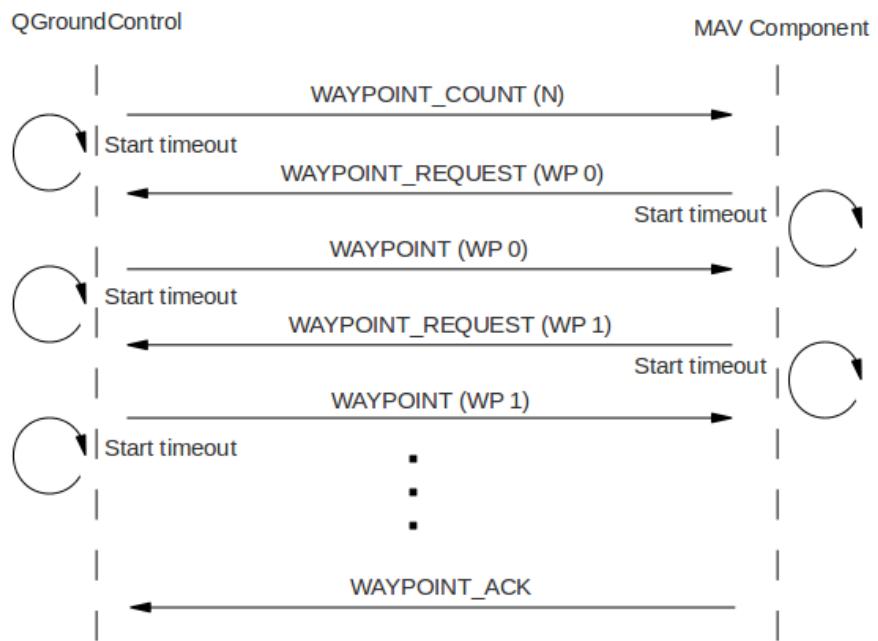


Figura 2.23: Protocolo de escritura de misión en Mavlink.

2.2.20.GCS

UNA GCS o estación de control de tierra por sus siglas en inglés (Ground Control Station) es un centro de control que facilita a un ser humano el control de vehículos aéreos no tripulados en el aire o en el espacio. La estación de control de tierra te permite una comunicación directa con el UAV a la hora de mandarle comandos o programar una misión, cuando se realiza una misión hay que hacer una sesión de pre-vuelo donde se comprueba que todos los componentes del avión funcionan correctamente, luego se envía una orden de despegue y una vez volando se comanda una misión al UAV, una serie de waypoints y acciones que tendrá que realizar, en cualquier momento se puede enviar una orden de vuelta a casa, una orden de que permanezca en el sitio volando en círculos y otras muchas. Todo esto se controla desde la estación de control, que en todo momento monitoriza la posición del UAV, la información de telemetría que nos envía, el estado de la batería o la gasolina, los motores, la attitude, etcétera.

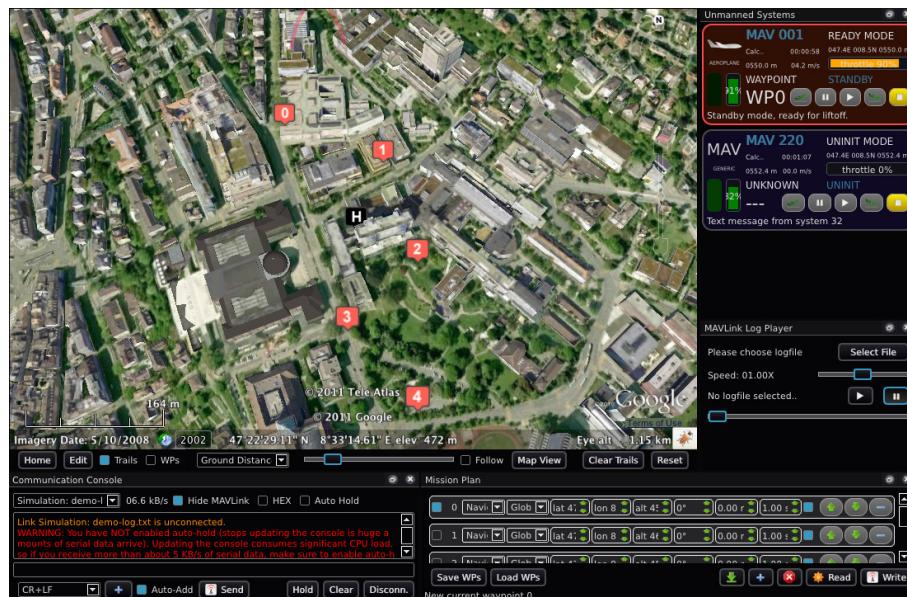


Figura 2.24: QGroundControl, gcs de Mavlink.

En nuestro caso usaremos la GCS creada por los departamentos de Aviación y Sistemas No Tripulados y Simulación y Software de FADA-Catec. La GCS de Catec nos permite realizar las operaciones necesarias para probar nuestro sistema, usa el protocolo de comunicaciones Mavlink mediante UDP, nosotros le realizaremos una modificación para que pueda comunicarse con el resto de módulos mediante ANIMO DDS. Con ésta GCS podemos monitorizar los datos del UAV, comandarle una misión, comandarle que haga *loiter* (Que gire en torno a un waypoint) o que entre en *failsafe* y vaya a una zona segura. También incorpora una pantalla de monitorización de alarmas que nos indica en todo momento el estado de la aeronave.

La GCS de Catec está basada en plugins, por lo que su funcionalidad es ampliable, disponemos de plugins de monitorización 3D de la aeronave, plugins de control de waypoints, plugins de monitorizaciones de telemetría, de control de carga de pago, etcétera.

Nosotros hemos creado un plugin de comunicación para ANIMO DDS el cual transforma los mensajes que maneja la GCS a mensajes que puedan viajar mediante ANIMO DDS para que puedan comunicarse con nuestro simulador.

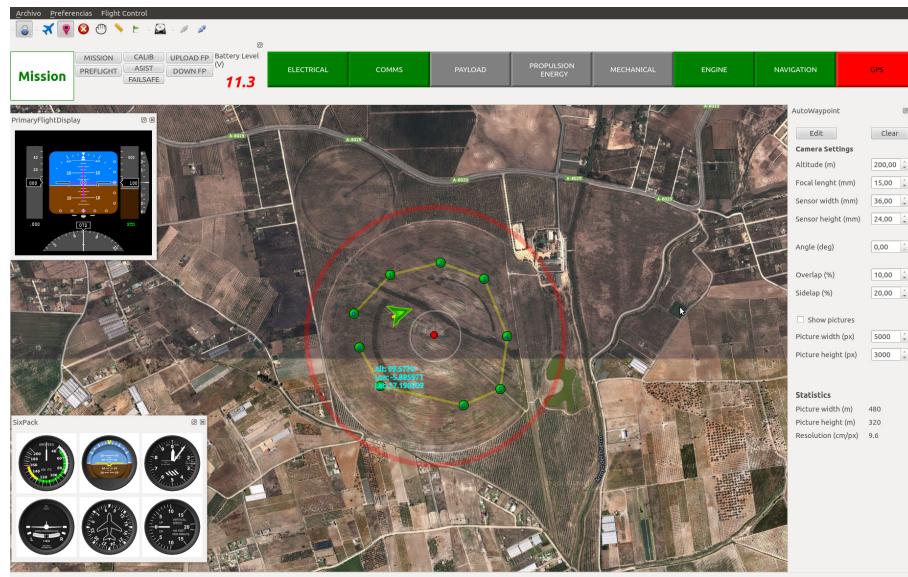


Figura 2.25: GCA de Catec.

2.2.21.LOCOMOVE

FADA-Catec dispone de varias aeronaves de desarrollo propio, para nuestro simulador usaremos el avión eléctrico no tripulado Locomove. Es un RPAS ligero con un motor eléctrico que destaca por sus prestaciones de carga de pago, autonomía y rango de vuelo para el desarrollo de aplicaciones en diferentes ámbitos. Es un avión de ala fija con una envergadura de algo más de dos metros, es de despegue manual, por lo que no necesita pista de despegue y una autonomía de unos 45 a 60 minutos, puede llevar una carga de pago de un kilo y alcanza velocidades de entre 60 y 100 kilómetros por hora.

Al estar el autopiloto de este dispositivo preparado para trabajar con el resto de nuestros componentes y estar realizado íntegramente en el centro supone un sujeto de pruebas perfecto ya que tenemos total control sobre sus componentes y a su vez el proyecto servirá para depurar fallos en el mismo avión y poder probar las nuevas funcionalidades que se le añadan antes de realizar un vuelo real.



Figura 2.26: UAV de Catec, Locomove.

2.2.22. AUTOPILOTO

El autopiloto embarcado dentro del UAV Locomove ha sido diseñado y fabricado por el departamento de Aviónica y Sistemas No Tripulados de FADA-Catec, es un dispositivo montado sobre una placa *BeagleBone*, una placa de bajo consumo y hardware libre que contiene en un único panel un ordenador completo, está pensada para utilizar software libre y se usa debido a su pequeño tamaño y peso y gran flexibilidad a la hora de desarrollar sobre ella. Las barrica la empresa americana Texas Instruments junto con Digi-key y Newmark element14.

La BeagleBone dispone de un procesador Cortex-A8, una memoria DDR2 de 256 MB de capacidad, una ranura para memorias microSD, una entrada Fast Ethernet, y una entrada de alimentación de 300-500 mA y 5 V mini USB.

El sistema operativo utilizado en el autopiloto es QNX, un sistema operativo en tiempo real de tipo Unix desarrollado para su uso en sistemas embebidos por QNX Software Systems, empresa adquirida por BlackBerry. Está basado en una estructura micronúcleo que proporciona características de estabilidad avanzadas frente a fallos de dispositivos, aplicaciones, etcétera. Los sistemas operativos de tiempo real son interesantes para situaciones donde sea absolutamente necesaria una toma continua de, por ejemplo, muestra de datos. Basándose en este interés, existen diversos proyectos para crear nuevas versiones en tiempo real de otros sistemas. Está orientado a su utilización en microcontroladores y sistemas críticos como podría ser nuestro ordenador embarcado.

El autopiloto se compone de varios módulos funcionales, cada uno con una misión bien definida. Por una parte tenemos un módulo de tratamiento de mensajes que es el encargado de hacer llegar a cada componente su respectivo mensaje, otro módulo se encarga de manejar la máquina de estados de la aeronave, también hay un módulo de control y guiado de la misión y un estimador que ayuda al pilotaje del avión.

A parte de los módulos incluidos y necesarios para el vuelo real del UAV, en nuestro autopiloto se incluirá un módulo dedicado a emular los sensores del avión, ya que nuestro autopiloto no estará volando realmente no podrá leer datos de posición GPS ni de velocidad de viento etc, por tanto estos datos se los proporcionaremos nosotros simulando el entorno real de vuelo. Este modulo llamado *modelo* del avión nos devolverá respuestas frente a órdenes enviadas por el autopiloto de la misma forma que lo harían los servos y sensores del UAV Locomove. Más adelante se implantará la posibilidad de cambiar éste modelo para poder emular diferentes tipos de aviones en nuestro entorno.

Quitando el modelo incorporado al autopiloto el resto del sistema es exactamente el mismo sistema que va embarcado dentro del UAV, por tanto tenemos la seguridad que el tratamiento y la respuesta será exactamente igual que en unas condiciones de vuelo real.



Figura 2.27: Autopiloto integrado en el Locomove.

Parte III

Sistema a desarrollar

Capítulo 3. Planificación inicial

3.1. LANZAMIENTO DEL PROYECTO

A principios del mes de Octubre de 2013 se reúne el equipo de desarrollo del proyecto SIMFORPAS formado por Irene Alejo, Jaime Rey, Manuel Mateos y Yamnia Rodríguez junto con Pablo Morillas desarrollado de la GCS en carácter de experto sobre comunicaciones entre GCS y UAV y manejo de la GCS. Ésta reunión constituye la primera etapa del desarrollo incremental guiado por Scrum, en ella se definirán los roles existentes y se pondrán en común las historias de usuario necesarias para la consecución de los objetivos del proyecto.

3.1.1. DEFINICIÓN DE ROLES

S E ponen en común los posibles roles que participarán en el desarrollo entre los que salen el cliente, que comprará la aplicación, los usuarios que serán el instructor y el alumno, el equipo de desarrollo y el instalador del sistema. Estos actores tendrán sus requerimientos sobre el sistema y sus exigencias sobre la funcionalidad que tiene que darse en él para su beneficio. Por tanto haremos una descripción de la función de cada uno y después los usaremos para definir las historias de usuarios.

Al ser un proyecto de investigación sobre ésta tecnología no disponemos de un cliente, si en un futuro se pretende vender éste producto nos pondremos nosotros mismos en el papel del cliente imaginando cómo debe ser el producto final.

- **Instructor:** Será el encargado de manejar el puesto de instructor mediante el cual podrá recibir datos del puesto de un alumno y en base a la información intercambiada decidir si éste es apto o no para recibir un título de operador de GCS.

El instructor necesitará herramientas que le permitan evaluar los conocimientos del alumno. Deberá visualizar cuando sea preciso el puesto del alumno para monitorear sus acciones, ser capaz de introducir errores en el modelo para comprobar la reacción del alumno y guardar un log con la evolución de la misión.

- **Alumno:** Este actor será evaluado por un instructor haciendo uso del sistema. Requerirá disponer de una GCS que simule un entorno real de pilotaje de UAV, un instructor que le cargue una misión y supervise durante el desarrollo de la misma. El sistema deberá tener unas condiciones lo más parecidas a un caso real para

comprobar que los conocimientos adquiridos por el alumno son correctos y pueden usarse en un entorno real.

- **Desarrollador:** Es el encargado del diseño, escritura y posterior montaje del sistema. Necesitará para ello información verídica sobre los requisitos que se necesitan, un entorno de desarrollo adecuado y las herramientas necesarias.
- **Instalador del sistema:** Es el actor encargado de preparar los equipos para que trabajen con el software desarrollado durante el proyecto. Necesitará un equipo adecuado para instalar el puesto del alumno y otro para el del instructor, que el software disponga de un instalador con todas las herramientas necesarias para el correcto funcionamiento del sistema.
- **Cliente:** Será el interesado en adquirir el software para instruir a futuros operadores de GCS. Necesitará un entorno de aprendizaje que simule fielmente las condiciones de pilotaje de UAV y permita a un instructor ser capaz de evaluar al alumno.

3.1.2. HISTORIAS DE USUARIO

A partir de los actores del proyecto se definirán las historias de usuario. Una historia de usuario es una representación de un requisito de software escrito en una o dos frases utilizando el lenguaje común del usuario. Éstas historias de usuario son utilizadas en la metodología Scrum para la especificación de requisitos. Cada historia de usuario debe ser limitada, esta debería poderse escribir sobre una nota adhesiva pequeña. Son una forma rápida de administrar los requisitos de los usuarios sin tener que elaborar gran cantidad de documentos formales y sin requerir de mucho tiempo para administrarlos. Las historias de usuario permiten responder rápidamente a los requisitos cambiantes. Vamos a enumerar características que deben cumplir las historias de usuario.

Deben ser:

- **Independientes unas de otras:** De ser necesario, combinar las historias dependientes o buscar otra forma de dividir las historias de manera que resulten independientes.
- **Negociables:** La historia en si misma no es lo suficientemente explícita como para considerarse un contrato, la discusión con los usuarios debe permitir esclarecer su alcance y éste debe dejarse explícito bajo la forma de pruebas de validación.
- **Valoradas por los clientes o usuarios:** Los intereses de los clientes y de los usuarios no siempre coinciden, pero en todo caso, cada historia debe ser importante para alguno de ellos más que para el desarrollador.
- **Estimables:** Un resultado de la discusión de una historia de usuario es la estimación del tiempo que tomará completarla. Esto permite estimar el tiempo total del proyecto.
- **Pequeñas:** Las historias muy largas son difíciles de estimar e imponen restricciones sobre la planificación de un desarrollo iterativo. Generalmente se recomienda la consolidación de historias muy cortas en una sola historia.

- **Verificables:** Las historias de usuario cubren requerimientos funcionales, por lo que generalmente son verificables. Cuando sea posible, la verificación debe automatizarse, de manera que pueda ser verificada en cada entrega del proyecto.

Después de definir todas las historias de usuario hay que valorar la dificultad de cada una de las historias de usuario para hacer una estimación del tiempo necesario para llevarlas a cabo. El siguiente paso consiste en darles a cada una una prioridad basada en la necesidad que tenemos de que se realicen primero o de la importancia que tengan en el sistema. Ahora enumeraremos las historias de usuario propuestas durante la reunión de lanzamiento del proyecto y la importancia que se le dio a cada uno, a mayor número junto a la historia de usuario mayor importancia tendrá, también añadiremos el grado de dificultad de cada historia, para dar una imagen global de como usar los puntos, un grupo de trabajo de cuatro personas tiene una velocidad aproximada de 20 puntos por semana.

- Como cliente quiero que el desarrollo del simulador se lleve usando las herramientas y metodologías necesarias para augurar el éxito en el desarrollo del proyecto.

Dificultad: 2

Prioridad: 9990

- Como desarrollador quiero tener bien definida la arquitectura básica a alto nivel y las herramientas principales que se van a utilizar para no perder la vista del objetivo principal del proyecto.

Dificultad: 2

Prioridad: 9980

- Como instructor quiero que las comunicaciones entre los sistemas se realice en tiempo real como uno de los puntos para garantizar el entrenamiento veraz.

Dificultad: 3

Prioridad: 9890

- Como desarrollador quiero poder insertar un modelo simple de UAV en el simulador para ayudar a definir el método de carga de modelos físicos.

Dificultad: 3

Prioridad: 9880

- Como desarrollador quiero un módulo central capaz de cambiar y monitorizar el estado del modelo.

Dificultad: 5

Prioridad: 9790

- Como desarrollador quiero un módulo que emule las GCS en el envío y recepción de datos para no depender de las GCS a la hora de desarrollar el resto de módulos y tener así el bucle completo cerrado GCS-SIMCore-SIMModel.

Dificultad: 3

Prioridad: 9780

- Como cliente quiero que el modelo contemple la carga de los datos de misión utilizados en el protocolo Mavlink.

Dificultad: 5

Prioridad: 9785

- Como desarrollador quiero que el módulo central esté sincronizado con el modelo para asegurar que los datos son coherentes.
Dificultad: 8
Prioridad: 9775
- Como instructor quiero que las GCS visualicen la telemetría del UAV simulado para tener control de la misión.
Dificultad: 5
Prioridad: 9770
- Como instructor quiero que el modelo del UAV sea lo más real posible para garantizar los conocimiento del alumno.
Dificultad: 8
Prioridad: 9760
- Como cliente quiero que el desarrollo del puesto de instructor se lleve usando las herramientas y metodologías necesarias para augurar el éxito en el desarrollo del proyecto.
Dificultad: 2
Prioridad: 9755
- Como instructor quiero usar el autopiloto real para realizar el entrenamiento y examen.
Dificultad: 8
Prioridad: 9754
- Como instructor quiero que las GCS establezcan un plan de vuelo para la misión simulada.
Dificultad: 8
Prioridad: 9750
- Como instructor quiero poder iniciar la simulación para tener control de la misión.
Dificultad: 8
Prioridad: 9680
- Como instructor quiero poder parar la simulación para tener control de la misión.
Dificultad: 5
Prioridad: 9670
- Como instructor quiero poder pausar la simulación para tener control de la misión.
Dificultad: 5
Prioridad: 9660
- Como instructor quiero introducir fallos de pérdida total de comunicación para poner a prueba al alumno.
Dificultad: 8
Prioridad: 9650
- Como instructor quiero visualizar las GCS para tener controlado al alumno.
Dificultad: 5
Prioridad: 9590

- Como instructor quiero crear misiones para tener una batería de exámenes.
Dificultad: 2
Prioridad: 9580
- Como instructor quiero grabar misiones que he generado en las GCS para poner a prueba al alumno en diferentes contextos.
Dificultad: 5
Prioridad: 9570
- Como instructor quiero cargar misiones que he generado en las GCS para poner a prueba al alumno en diferentes contextos.
Dificultad: 2
Prioridad: 9560
- Como instalador quiero que el sistema completo sea sencillo de administrar en instalar.
Dificultad: 1
Prioridad: 9490
- Como Instructor, quiero tener acceso a información que me indique, paso a paso, el funcionamiento del puesto de instructor.
Dificultad: 1
Prioridad: 9480
- Como instructor quiero que el modelo del UAV sea lo más real posible para garantizar los conocimiento del alumno.
Dificultad: 20
Prioridad: 8990
- Como instructor quiero cambiar las condiciones del entorno (lluvia, viento?) para poner a prueba al alumno.
Dificultad: 20
Prioridad: 8980
- Como instructor quiero elegir un UAV para la misión con el objetivo de evaluar al alumno en un UAV concreto.
Dificultad: 8
Prioridad: 8970
- Como Instructor quiero almacenar/ consultar resultados de las pruebas para posterior evaluación o consulta a histórico del alumno.
Dificultad: 5
Prioridad: 8960
- Como instructor quiero tomar y guardar anotaciones sobre el examen de un alumno para posterior evaluación.
Dificultad: 3
Prioridad: 8950

Una vez tenemos todos estos datos podemos realizar el documento de backlog, un documento de alto nivel para todo el proyecto. Éste documento va a ir incluyendo gráficas

que representan la evolución del proyecto en cada sprint.

Prioridad	Título	Descripción y notas	Test de aceptación	Estimación de puntos	Sprint comprometido	Sprint terminado	Sprint en que se ha iniciado
9990 Entorno de desarrollo I		Como cliente quiero que el desarrollo del simulador se lleve usando las herramientas y metodologías necesarias para augurar el éxito en el desarrollo del proyecto	* Desarrollo usando metodologías ágiles como: • TDD • Refactor Mercurial para proyecto • Entorno de IC configurado para el simulador • Analizador estático de código configurado, Klocwork, para el simulador. • Entorno de desarrollo, Eclipse, para el simulador. • Configuración docker. • Reglas de la empresa que link con documentación autogenerada de código.	2	1	1	
9980 Documento de Análisis		Como desarrollador quiero tener bien definida la arquitectura básica a alto nivel y las herramientas principales que se van a utilizar para no perder la vista del objetivo principal del proyecto	* Generación de documento no formal de arquitectura a alto nivel.	2	1	1	
9980 Interfaces		Como instructor quiero que las comunicaciones entre los sistemas se realicen en tiempo real como uno de los puntos para garantizar el entrenamiento veraz.	* Generación de documento de interfaces entre módulos que se anexará al de arquitectura	3	1	2	
9980 Modelos Matlab-Simulink		Como desarrollador quiero poder insertar un modelo simple de UAV en el simulador para ayudar a refinar el método de carga de modelos físicos.	* Generación de entradas y salidas de los modelos en matlab simulink • Generación de integración de los modelos en matlab simulink • Spike de integración de módulo de Matlab-Simulink en un programa en código C++ que permita cargar datos de entrada y mostrar los datos de salida.	3	2	2	
9790 Makeup de Core de simulador I		Como desarrollador quiero un módulo central capaz de cambiar y monitorizar el estado del modelo.	* El SIMCore recibe los datos de SIMModel y GCS y los renvía, haciendo de puente usando ANNN • Transmisor de datos. Dicho envío desde SIMModel a UAV_Sim, para que la GCS tenga los datos necesarios siguiendo el protocolo MavLink.	5	2	2	
9780 Módulo Dummy de GCS		Como desarrollador quiero un módulo que envíe las OCS en el envío y recepción de datos para no depender de las OCS a la hora de desarrollar el resto de módulos y tener así el bucle completo cerrado GCS-SIMCore-SIMModel	* GCSDummy envía la carga de datos de un conjunto de puntos fija, enviando los waypoints cuando correspondan al SIMCore • El SIMCore recibe los datos de GCSDummy y los guarda en memoria • Spike de integración de módulo de GCS-Dummy en un programa en código C++ que represente los datos.	3	3	3	
9785 Gestión de carga de waypoints en SimModel		Como cliente quiero que el modelo contemple la carga de los datos utilizados en el protocolo Mavlink.	* El SIMModel recibe los datos de mision para lo misioner al modelo, uno por uno, los waypoints generandole una pequeña capa de navegación.	5	3	3	
9775 Framework de simulación		Como desarrollador quiero que el módulo central esté sincronizado con el modelo para asegurar que los datos son coherentes	* Se envía el estado del modelo cada n Hz, siendo n una medida configurada de antemano. Los n Hz del modelo pueden ser menores al de la simulación, en ese caso el modelo se actualiza y se reinicia una vez cada n Hz. • El SIMCore manda cada 1 ms al módulo para su actualización, si todos los demás módulos han enviado sus datos para el paso actual... • Es decir, el SIMCore manda los datos de los demás módulos para que el módulo el SIMDep. Si aún así, en varios intentos no se han recibido los datos de los demás módulos, el SIMCore manda una señal de error al módulo, ignorando los datos que tuviera anteriormente.—dato lo hace manual. • El SIMCore manda los datos de los demás módulos cuando se actualiza el paso de simulación actual. Si no, se desactualizan los datos de los demás módulos. • El sistema, en general, está sincronizado a los n Hz o los que establece el módulo GCS, ignorándose cualquier irregularidad que ocurra (retrasos, pausas o conexión...).	8	3	3	
9770 Integración GCS I		Como instructor quiero que las OCS visualicen la telemetría del UAV simulado para tener control de la misión.	* Se ve la telemetría en las OCS. • No hay retraso perceptible en la recepción de los datos.	5	4	6	
9780 Integración de modelo complejo		Como instructor quiero que el modelo del UAV sea lo más real posible para garantizar los conocimientos del alumno.	* Nuevo SimLectorModel que integra el modelo y submodelos de ASNT. • Nuevo SimLectorModel que integra el modelo y submodelos de ASNT. • Nuevo SimLectorModel que integra el modelo y submodelos de ASNT. • Si es necesario, añadir información al plugin GCS. • Actualizar el plugin GCS para que el simulador de GCS o las mismas OCS, este haga un recuento resultante.	8	4	4	
9765 Entorno de desarrollo II		Como cliente quiero que el desarrollo del puesto de instructor se lleve usando las herramientas y metodologías necesarias para augurar el éxito en el desarrollo del proyecto	* Desarrollo usando metodologías ágiles como: • TDD • Refactor Mercurial para proyecto (puede ser el mismo que el simulador) • Entorno de IC configurado. • Actualización constante de sistemas configuración. (Síguer, para muestra de)	2	4	6	

Figura 3.1: Ejemplo del documento de backlog.

Se concretan las acciones para el primer sprint, que consistirán en realizar documentación sobre la estructura que tendrá el proyecto, estudio de interfaces y set up de los entornos de trabajo.

Capítulo 4. Iteraciones: Sprints de desarrollo

4.1. SPRINT 1: SET UP DEL PROYECTO

Una vez estudiados los requisitos del sistema a partir de las historias de usuario pasaremos a desarrollar la estructura global del sistema y a documentarlo todo.

4.1.1. MÓDULOS DEL SISTEMA

Se necesitará una serie de módulos que tengan una función definida cada uno de ellos con el fin de hacer el sistema lo más escalable posible de manera que si en algún momento hay que realizar un cambio grande no afecte al resto de componentes, cada módulo será un proyecto independiente para descentralizar el desarrollo, así se determina la necesidad de disponer de los siguientes elementos:

- **Control:** Recibe un comando de control (Waypoint, velocidad, acción de algún sensor, etc) y calcula qué cambios hay que realizar en el modelo para alcanzarlo. El autopiloto puede ser parte del módulo de control.
- **Modelo:** Simula las condiciones del entorno y comportamiento del UAV. Utilizará modelos lo más reales posibles. Mantendrá en todo momento un estado coherente del sistema, es decir, un cambio en alguno de sus parámetros afectará al resto según las reglas de la física y de comportamiento que se hayan implementado.
- **Simulador:** Coordina el funcionamiento del resto de módulos y monitoriza el estado del sistema en cada momento. Gestiona el flujo de información entre los distintos módulos durante la ejecución del programa.
- **GCS:** Estación de comando y de control del UAV. Es la interfaz entre el sistema y el operador, muestra en todo momento el estado del modelo y la misión. Esta será la plataforma sobre la que el operador será entrenado y evaluado. El Instructor debe tener visibilidad sobre lo que el operador está haciendo en cada momento durante el desarrollo de la misión.
- **Puesto de instructor:** Sirve de interfaz entre el instructor y el sistema, desde ella se podrá cargar un modelo, una misión y controlar las acciones del alumno durante la misma. También permitirá introducir errores o cambios en el sistema durante una misión con el fin de evaluar la reacción del alumno.

Es necesario un elemento central que controle el flujo en información entre el resto de módulos y los sincronice para que no se produzcan situaciones de pérdidas de datos o de desfases de tiempo entre los distintos módulos que puedan provocar errores. Otro de los elementos necesarios será el llamado modelo del avión, éste modelo en una primera instancia deberá de disponer de un sistema de guiado y de tratamiento de acciones como la de la carga y gestión de la misión de forma que nos pueda devolver datos de telemetría actualizados según las condiciones globales del sistema. También se dispondrá de un puesto de instructor que permitirá a un examinador monitorizar la misión del alumno y de modificar las condiciones para comprobar su reacción. Por último haremos uso de la GCS ya implementada por Catec para que el alumno maneje el simulador.

4.1.2. ESTRUCTURA DEL SISTEMA

Al final la arquitectura queda dispuesta de la siguiente manera, comunicándose todos los módulos mediante ANIMO DDS:

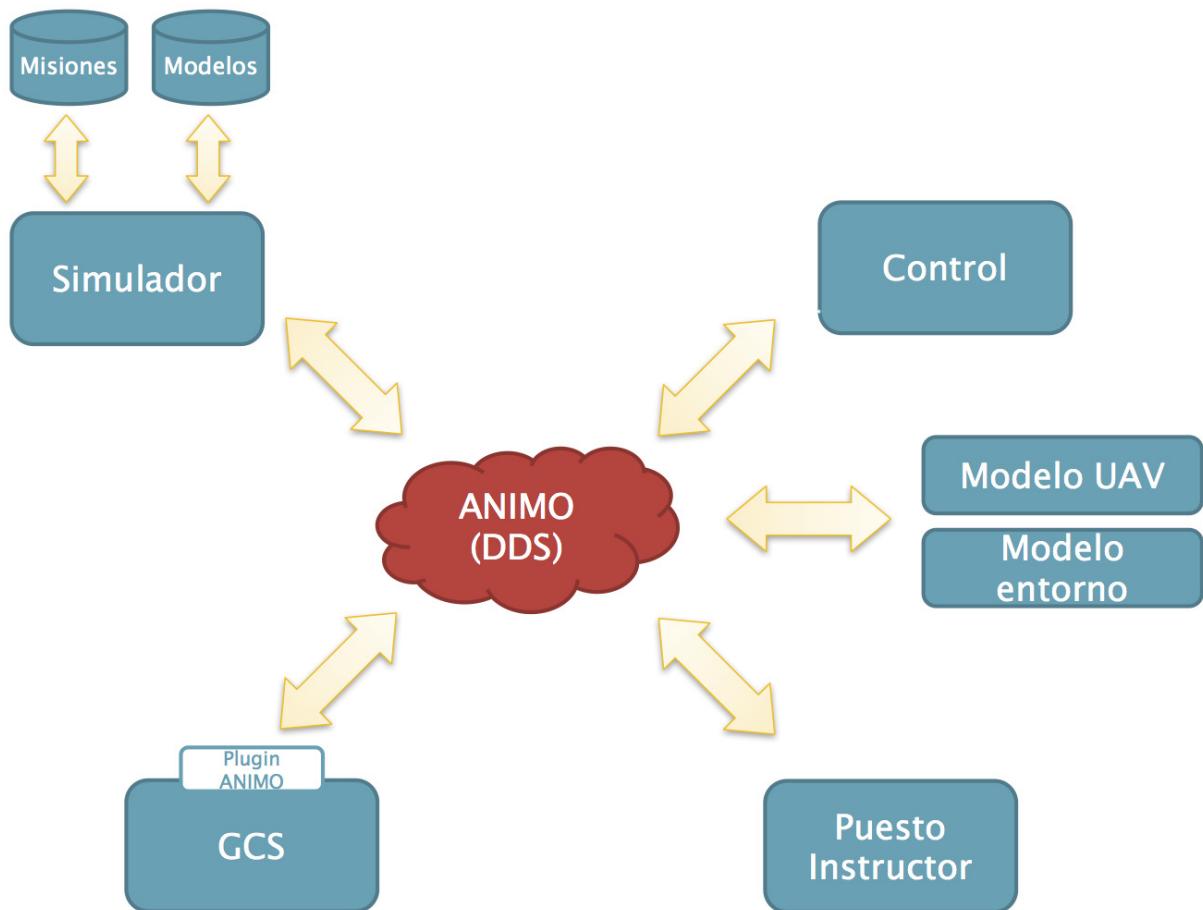


Figura 4.1: Arquitectura del sistema, con las relaciones físicas entre subsistemas.

Como podemos observar en el diagrama de arquitectura, el proyecto se dividirá en módulos que se encargarán de una función específica. Para realizar la interconexión entre

los diferentes módulos usaremos el Framework de comunicaciones ANIMO para garantizar que las comunicaciones se establecen en tiempo real.

El módulo del modelo y el control serán desarrollados con Matlab-Simulink. Para el simulador se usarán tecnologías web y servicios web para comunicarse con él (haciendo uso de ANIMO).

En el caso de las bases de datos necesarias para guardar información sobre modelos de UAV, de entorno y misiones se hará uso de bases de datos no relacionales debido a que son más fáciles de diseñar, administrar y proveen de una mejora sustancial en la velocidad del sistema.

La GCS permitirá al usuario crear, modificar e incluso cargar misiones y modelos desde un archivo externo al sistema, el código que gestiona el funcionamiento de la GCS estará desarrollado en C++ y la parte encargada de la interfaz de usuario se realizará usando las bibliotecas QT.

4.1.3. DIAGRAMAS DE SECUENCIA

Una vez estudiados los elementos necesarios haremos una lista de las interfaces u objetos que necesitaremos manejar durante la ejecución de nuestro programa. Para ello tuvimos una reunión con el equipo que trabaja en el desarrollo de la estación de control para saber qué datos manejan ellos, conocer las entradas y salidas de su aplicación, los pasos de ejecución etcétera. Como nosotros debemos simular el comportamiento de la aeronave vamos a definir una serie de diagramas de secuencias donde se definen las distintas acciones que implementaremos en nuestra aplicación.

- **Funcionamiento básico durante la ejecución de una misión**

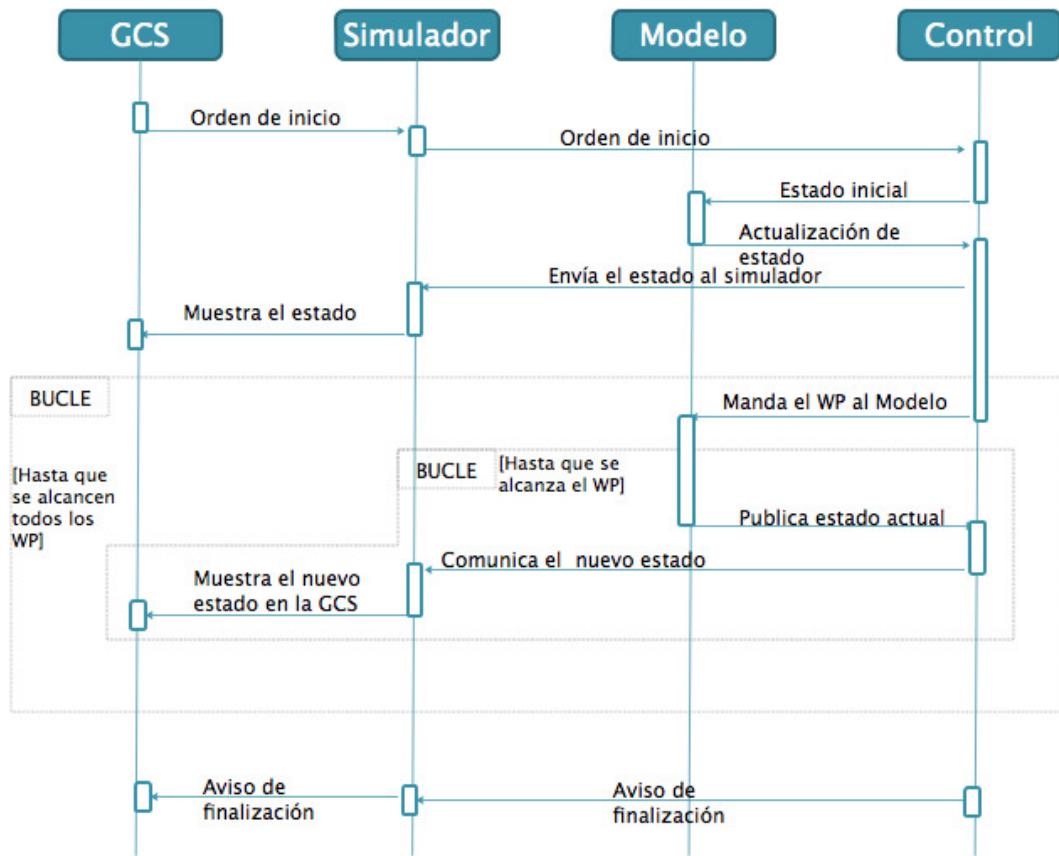


Figura 4.2: Diagrama de secuencia de la ejecución de una misión.

En el primer paso la GCS envía una señal de inicio de ejecución de una misión al simulador que reenvía al control. El control envía el estado inicial al modelo que responde con el estado actualizado que más tarde se envía al simulador para mostrarlo en la GCS.

A continuación se inicia el bucle principal de la misión, el cual acabará cuando se hayan alcanzado todos los Waypoints. El control envía un Waypoint al modelo, que responde con su estado el cual se envía al simulador para mostrarlo en la GCS. Una vez alcanzado este Waypoint el control procederá a enviar el siguiente. Cuando se hayan alcanzado todos los Waypoints ha acabado la misión y el simulador envía un mensaje a la GCS.

- Carga de la misión en el simulador y en control

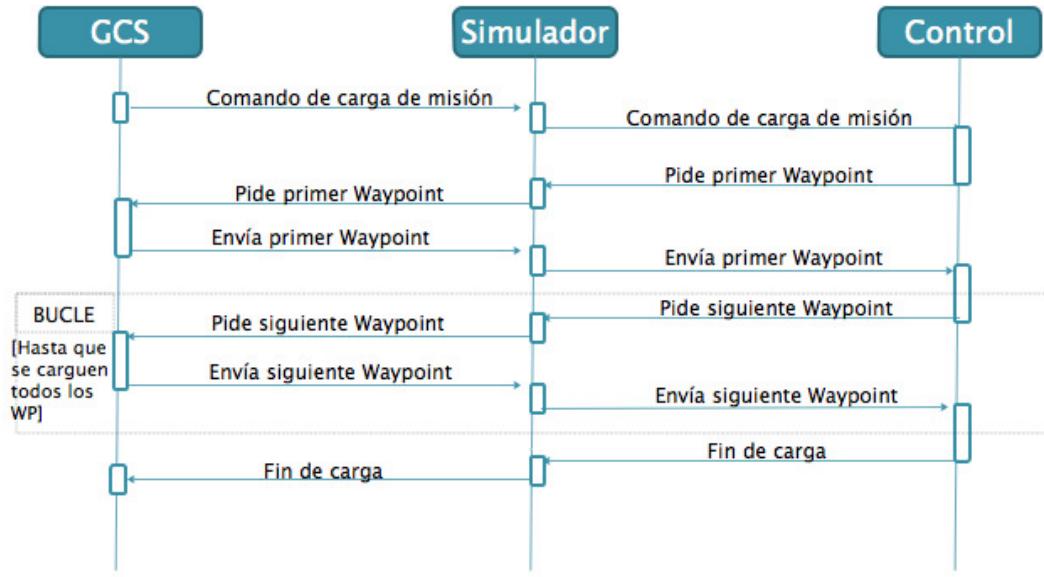


Figura 4.3: Diagrama de secuencia de la carga de una misión en el simulador y el control.

El método de carga de la misión está basado en el protocolo MAVLINK. La GCS envía el número de Waypoints de la misión al simulador, que hace de puente de comunicación entre el control y la GCS, el control pide uno a uno los waypoint y la GCS se los va mandando. Cuando recibe el último waypoint el control envía un mensaje de fin de carga de la misión. En el protocolo de escritura de misiones en el UAV de MAVLINK cuando se envía un mensaje que espera respuesta se inicia un timer si se consume ese tiempo sin respuesta se realiza otra vez la petición, esta acción la realizará el simulador.

- Inicio, parada y pausa de la simulación

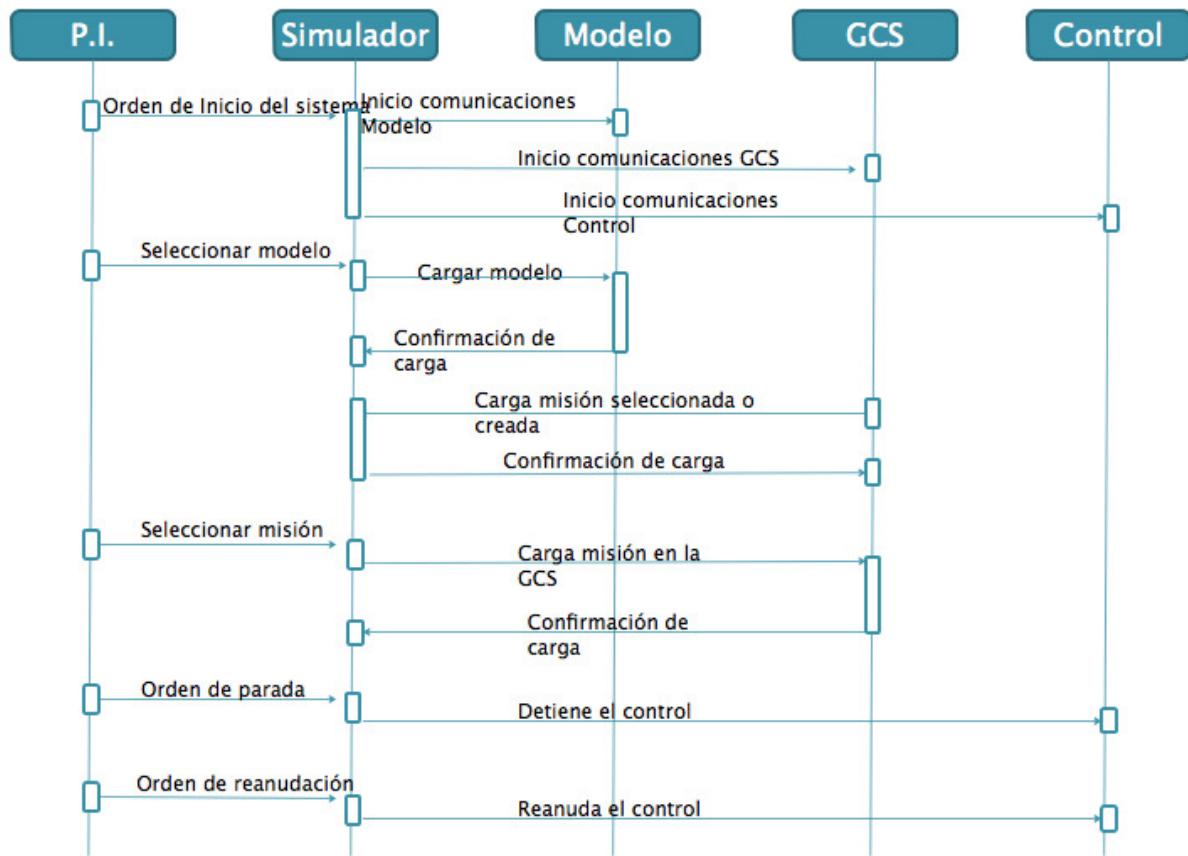


Figura 4.4: Diagrama de secuencia con los comandos desde el puesto de instructor para iniciar, parar o pausar una ejecución.

El primer comando es enviado por el puesto de instructor al simulador para que éste establezca las comunicaciones con cada uno de los módulos para iniciar la ejecución. El segundo comando muestra el flujo de información al cargar el modelo desde el puesto de instructor. El instructor selecciona el modelo a cargar (tanto del entorno como del UAV), el simulador recibe ese modelo y lo carga en el módulo ?Modelo? y se envía un mensaje al simulador mostrando si ha ocurrido algún error durante la carga. El tercer comando muestra al alumno cargando (o creando) una misión, la cual enviará al simulador y éste le devolverá un mensaje con el resultado de la carga. El cuarto es la operación anterior pero realizada desde el puesto de instructor, carga la misión en el simulador, la inicializa en la GCS y se envía el correspondiente mensaje de errores. Los dos últimos comandos muestran el funcionamiento de las órdenes de parada y reanudación, las cuales se realizarán sobre el módulo de control.

- **Introducción de fallos en el modelo**

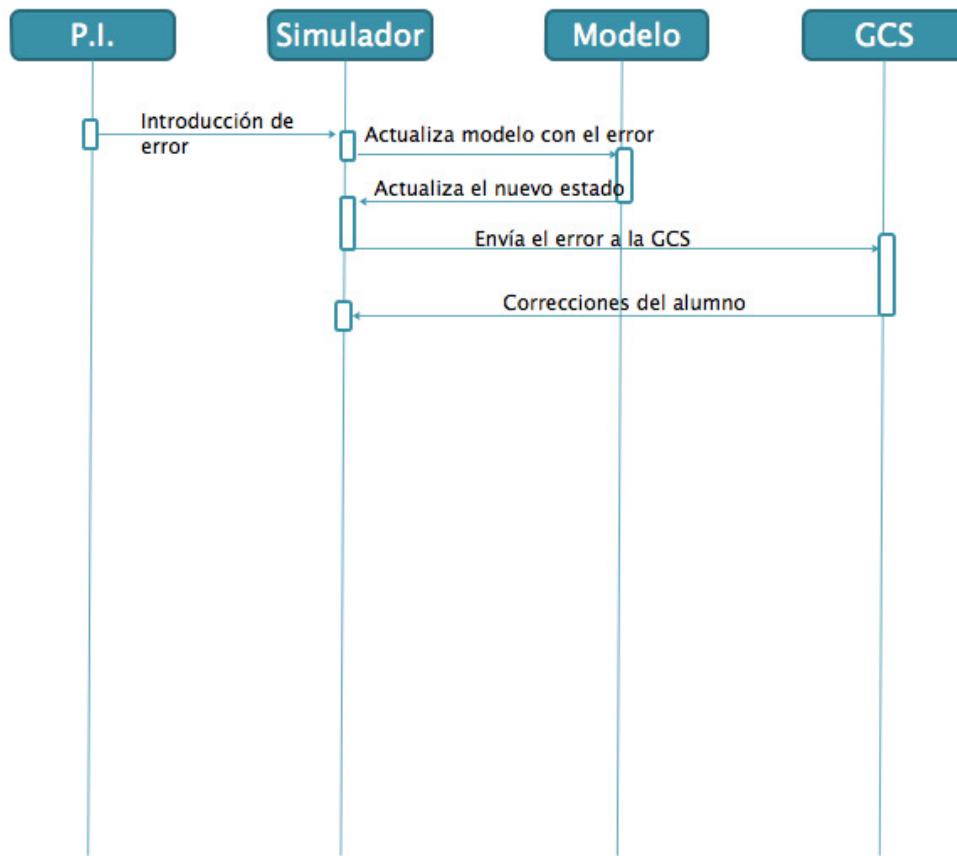


Figura 4.5: Diagrama de secuencia para la introducción de fallos.

El diagrama muestra el intercambio de datos cuando el instructor desea introducir errores en el sistema con el fin de comprobar la reacción del alumno. Se selecciona el error a introducir en el puesto de instructor y se notifica al simulador, se procesa el error y se actualiza el modelo, el modelo realiza los cambios necesarios y para finalizar el simulador envía el nuevo estado del modelo a la GCS.

En otra posible solución el control es el que detecta el fallo una vez se ha introducido en el modelo y genera el mensaje de error que envía al simulador para que se lo comunique a la GCS.

- Cambios en la misión

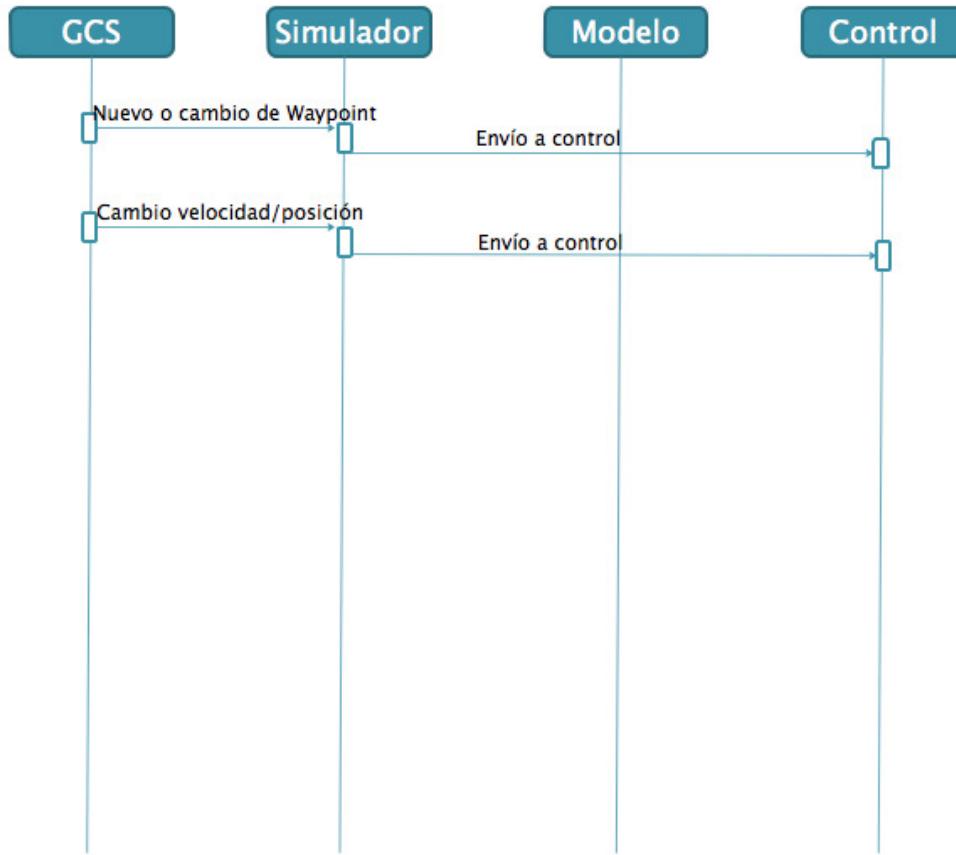


Figura 4.6: Diagrama de secuencia para hacer cambios en la misión.

Estas transacciones explican el intercambio de información en el caso en que se introdujeran cambios en la misión durante el vuelo (introducción de un nuevo Waypoint o cambios en el modelo de UAV como velocidad etc). La GCS introduce o modifica un Waypoint o modifica algún parámetro en el UAV y se lo comunica al simulador que es donde está almacenada la misión, el simulador le envía el nuevo Waypoint o parámetro al control cuando se lo pida.

4.1.4. DIAGRAMA DE COMUNICACIONES

Una vez tenemos las acciones que va seguir nuestro programa podemos hacer un diagrama de cómo se van a comunicar los distintos módulos y de esta manera simplificar la información que obtuvimos anteriormente.

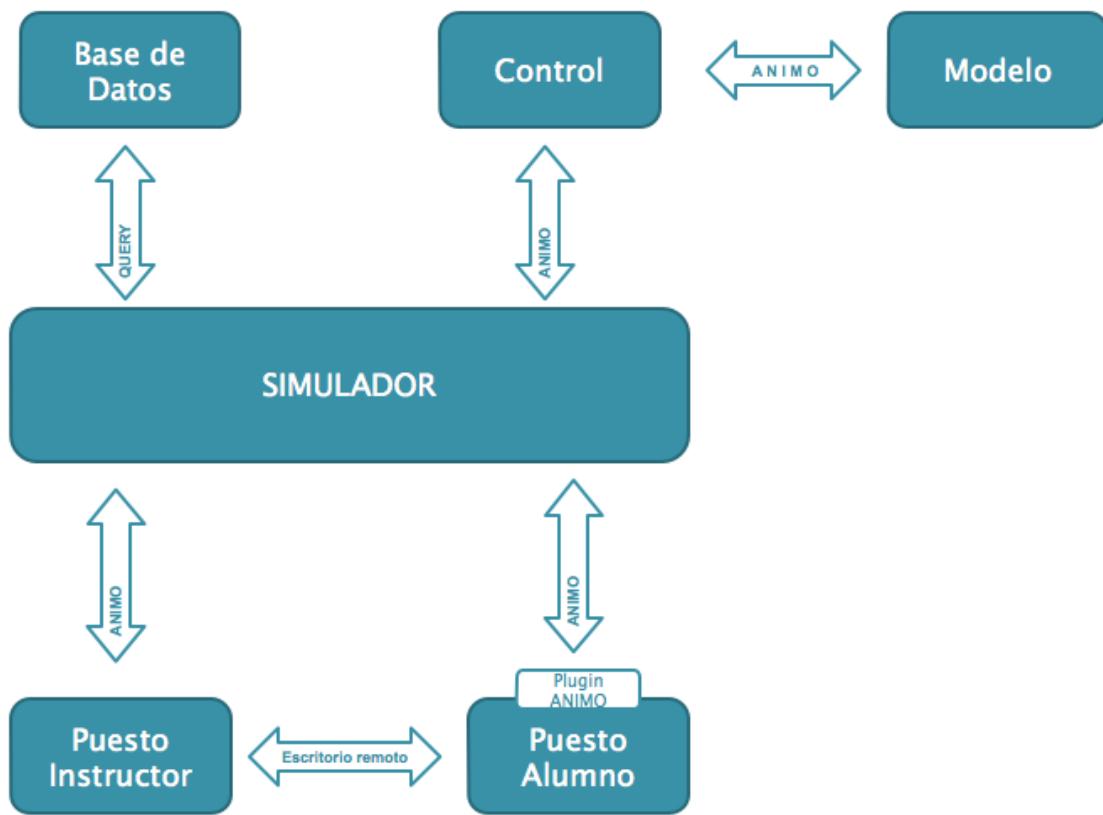


Figura 4.7: Diagrama de las relaciones a la hora de comunicarse de los distintos módulos.

De esta forma se puede observar de un vistazo que el simulador será el centro de las comunicaciones del sistema, el control se comunicará con el simulador para recibir los comandos cuya interpretación pasará al modelo y devolverá al simulador el nuevo estado de éste. El puesto de instructor enviará las acciones al simulador que será el encargado de redirigirlas hacia el puesto del alumno, el control o de hacer la comunicación con la base de datos. El puesto del alumno se conectará al simulador y desde ahí manejará el control y el modelo. El puesto de instructor podrá comunicarse con el puesto del alumno para monitorizar las acciones de éste y poder evaluarle luego.

4.1.5. ENTRADAS Y SALIDAS DE LOS MÓDULOS

Una vez sabemos qué datos se van a enviar entre los diferentes módulos podemos agrupar cada transacción según sean entradas o salidas de cada uno de los módulos de la siguiente manera:

1.- Puesto de instructor

Salidas

- Orden de inicio del sistema (tipo Command) → Simulador
- Identificador del modelo (tipo Command) → Simulador
- Misión (tipo Mission) → Simulador
- Orden de parada (tipo Command) → Simulador
- Orden de reanudación (tipo Command) → Simulador
- Fallo en el sistema (tipo Error) → Simulador

2.- Simulador

Entradas

- Orden de inicio (tipo Command) ← Puesto del instructor
- Identificador del modelo (tipo Command) ← Simulador
- Confirmación de carga (tipo Error) ← Modelo
- Misión (creadas en la GCS) (tipo Mission) ← GCS
- Identificador de la misión (tipo Mission) ← Puesto del instructor
- Confirmación de recepción de misión (cuando el instructor carga una misión en el simulador la GCS debe darse cuenta) (tipo Error) ← GCS
- Orden de parada (tipo Command) ← Puesto del instructor
- Orden de reanudación (tipo Command) ← Puesto del instructor
- Orden de inicio de misión (tipo Command) ← GCS
- Nuevo estado del modelo (Al inicio, tipos PosWgs , Rotation y VelEarth) ← Modelo
- Nuevo estado del modelo (En el bucle, tipos PosWgs , Rotation y VelEarth) ← Modelo
- Fallo en el sistema (tipo Error) ← Puesto de instructor
- Nuevo estado del modelo (después del error, tipos PosWgs , Rotation y VelEarth) ← Modelo
- Correcciones del alumno (después del error, tipo Command) ← GCS

Salidas

- Carga modelo (tipos PosWgs , Rotation y VelEarth) → Modelo
- Dato de confirmación (tipo Error) → GCS
- Misión (tipo Mission) → GCS
- Orden de parada (tipo Command) → Control
- Orden de reanudación (tipo Command) → Control
- Estado inicial del UAV (tipos PosWgs, Rotation y VelEarth) → Modelo
- Waypoint (tipo FlightPlan) → Control
- Introducción de error (tipo Command) → Modelo
- Aviso de error (tipo ?Error) → GCS

3.- GCS

Entradas

- Orden de inicio (tipo Command) ← Simulador
- Confirmación de carga de la misión (tipo Error) ← Simulador
- Misión cargada por el instructor en el simulador (tipo Mission) ← Simulador
- Estado del modelo (tipos PosWgs , Rotation y VelEarth) ← Simulador
- Aviso finalización (tipo Error) ← Simulador
- Aviso de error (tipo Error) ← Simulador

Salidas

- Misión (tipo Mission) → Simulador
- Confirmación de recepción de misión (tipo Error) → Simulador
- Orden de inicio de misión (tipo Command) → Simulador
- Reacción ante un problema (tipo Command) → Simulador

4.- Modelo

Entradas

- Orden de inicio (tipo Command) ← Simulador
- Tipo de dato modelo (tipos PosWgs , Rotation y VelEarth) ← Simulador
- Estado inicial del UAV (tipos PosWgs , Rotation y VelEarth) ← Simulador
- Consulta estado del modelo (tipos PosWgs , Rotation y VelEarth) ← Control
- Actualiza el modelo (tipos PosWgs , Rotation y VelEarth) ← Control
- Actualiza modelo con el error (tipos PosWgs , Rotation y VelEarth) ← Modelo

Salidas

- Confirmación de carga (tipo Error) → Simulador

- Comunica el estado inicial (tipos PosWgs , Rotation y VelEarth) → Simulador
- Publica estado actual (tipos PosWgs , Rotation y VelEarth) → Control
- Comunica nuevo estado del modelo (tipos PosWgs , Rotation y VelEarth) → Simulador
- Actualiza el nuevo estado (después de error, tipos PosWgs , Rotation y VelEarth) → Simulador

5.- Control

Entradas

- Orden de inicio (tipo Command) ← Simulador
- Orden de parada (tipo Command) ← Simulador
- Orden de reanudación (tipo Command) ← Simulador
- Waypoint (tipo FlightPlan) ← Simulador
- Estado actual del modelo (tipos PosWgs , Rotation y VelEarth) ← Modelo

Salidas

- Consulta estado del modelo (tipo Command) → Modelo
- Actualiza el modelo (tipos PosWgs , Rotation y VelEarth) → Modelo

4.1.6. DEFINICIÓN DE INTERFACES

A la hora de empezar a programar nuestra aplicación necesitamos conocer qué clases vamos a tener y qué información guardará cada una, ya que hemos realizado el ejercicio de anotar todas las comunicaciones de datos que se realizarán entre nuestros módulos y de las entradas y salidas de cada uno de ellos ya sabemos qué datos manejará cada uno, por tanto, definiremos cuáles son los datos que tenemos que modelar con clases.

GCS	PLUGIN
GLOBAL_POSITION_INT	I_State
time_boot_ms (unsigned int) lat (int) lon (int) alt (int) relative_alt (int) vx (int) vy (int) vz (int) hdg (unsigned int)	time_boot_ms (unsigned int) lat (int) lon (int) alt (int) relative_alt (int) vx (int) vy (int) vz (int) hdg (unsigned int)
ATTITUDE	I_Rotation
time_boot_ms (unsigned int) roll (float) pitch (float) yaw (float) rollspeed (float) pitchspeed (float) yawspeed (float)	data status (byte) data quality (byte) timestamp (timestampType) roll (double) pitch (double) yaw (double) attitudeType (EAttitudeType) deviceID (string)
MISSION_ITEM	I_Waypoint
target_system (unsigned int) target_component (unsigned int) seq (unsigned int) frame (unsigned int) command (unsigned int) current (unsigned int) autocontinue (unsigned int) param1 (float) param2 (float) param3 (float) param4 (float) x (float) y (float) z (float)	target_system (unsigned int) target_component (unsigned int) seq (unsigned int) frame (unsigned int) command (unsigned int) current (unsigned int) autocontinue (unsigned int) param1 (float) param2 (float) param3 (float) param4 (float) x (float) y (float) z (float)
COMMAND_LONG	I_Command
target_system (unsigned int) target_component (unsigned int) command (unsigned int) confirmation (unsigned int) param1 (float) param2 (float) param3 (float) param4 (float) param5 (float) param6 (float) param7 (float)	target_system (unsigned int) target_component (unsigned int) command (unsigned int) confirmation (unsigned int) param1 (float) param2 (float) param3 (float) param4 (float) param5 (float) param6 (float) param7 (float)
STATUSTEXT	I_Error
Severity (unsigned int) Text (char[50])	errorMessage (string) id (string) level (ECriticalityLevel) timestamp (timeStampType)

Figura 4.8: Definición de interfaces.

4.1.7. DIAGRAMA DE BURNDOWN

Para el primer sprint se esperaba realizar el documento de análisis inicial con un valor de dificultad de dos puntos y la definición de interfaces con otros dos puntos. Al haber realizado las dos acciones finalizamos el sprint con un total de 4 puntos en la evolución de velocidad de la iteración que plasmaremos en el siguiente diagrama de velocidad.



Figura 4.9: Diagrama de velocidad para el primer sprint.

4.1.8. DIAGRAMA DE EVOLUCIÓN

El diagrama de evolución nos muestra las historias de usuario que hemos realizado frente a las totales del proyecto, lo que nos da una idea aproximada de lo que nos queda para finalizar.



Figura 4.10: Diagrama de evolución para el primer sprint.

4.2. SPRINT 2: CREACIÓN DE LA CAPA DE COMUNICACIÓN

ESTE sprint lo dividiremos en tres partes, en un principio explicaremos los pasos necesarios para poner a punto las herramientas que usaremos durante el desarrollo del proyecto, entornos de programación, librerías necesarias, etcétera. En la segunda parte explicaremos cómo se implementó la capa de comunicación de los distintos módulos para realizar el intercambio de datos entre ellos, parte muy importante en el proyecto, también explicaremos como se usa ANIMO DDS. Para terminar realizaremos una primera integración del simulador con un modelo simple que acepta una trayectoria tomando un camino rectilíneo de un punto a otro de la misión.

4.2.1. SET UP DEL ENTORNO DE DESARROLLO

EL primer paso a la hora de iniciar el desarrollo del proyecto debe ser el de disponer de un entorno de desarrollo con todas las herramientas necesarias a la hora de llevar a cabo la programación de nuestro sistema, para ello haremos un recuento de las tecnologías que vamos a usar e instalaremos las correspondientes herramientas que faciliten el uso de éstas tecnologías.

ENTORNO DE DESARROLLO C++

Ya que la mayor parte del proyecto va a ser desarrollado en el lenguaje de programación C++ debemos disponer de un entorno de programación adecuado. Decidimos hacer uso de la herramienta de generación de proyectos en C++ Cmake, la cual nos resultará muy útil a la hora de crear varios proyectos que tengan las mismas características y usen las mismas tecnologías. Al decidir hacer uso de CMake nos decantamos por trabajar con el entorno de desarrollo integrado o *IDE* (Integrated Development Environment) QtCreator ya que éste tenía una mejor integración con CMake sobre la otra opción que tuvimos que era usar el entorno Eclipse adaptado a C++ que, a parte de ser más lento al ejecutarse en la máquina virtual de Java, disponía de herramientas útiles a la hora de trabajar con CMake y con las librerías de C++ Qt de las que haremos uso más adelante.

Tanto la última versión de Qt como el entorno de desarrollo QtCreator lo podemos encontrar en la página del proyecto <http://qt-project.org>.

Una vez disponemos de este entorno de desarrollo ya podemos crear nuestro primer proyecto con C++.

CREACIÓN DEL REPOSITORIO Y LA ESTRUCTURA DE CARPETAS

Ahora que tenemos el proyecto creado nos creamos un repositorio, usamos bitbucket ya que es el más usado por el departamento de Simulación y Software de Catec.

Mediante la herramienta TortoiseHG podremos manejar nuestro repositorio desde el escritorio. La metodología de trabajo será la siguiente: nos crearemos un *fork* del repositorio principal, subiremos todas las modificaciones que realicemos a ese fork, y cada vez que tengamos una versión estable del sistema haremos una petición de unión con el repositorio principal o *pull request*. Para la estructura de carpetas se definió que la ruta principal tendría dos carpetas, una destinada al código y otra a la documentación, dentro de la carpeta código la estructura sería crear una carpeta por cada módulo o proyecto que se necesite en el sistema.

INSTALACIÓN ANIMO DDS

El primer paso que tomamos una vez pudimos empezar a generar nuestro código en C++ fue el de instalarnos las librerías de ANIMO proporcionadas por Catec e integrarlas en un proyecto de prueba que consistía en dos aplicaciones, una que permitía tomar datos de un joystick conectado por USB al ordenador, enviar éstos datos por ANIMO y recibirlas en otra aplicación que dibujaba un punto sobre una ventana representando el movimiento del joystick.

El primer paso es descargarse las librerías de RTI DDS, las podemos encontrar en la siguiente url <http://www.rti.com/downloads/connexx-files.html/>, una vez tenemos el archivo de instalación debemos darle permisos de ejecución y ejecutarlo, aceptar las condiciones de uso y seleccionar el directorio de instalación. Antes de ejecutarlo debemos revisar algunas dependencias que tienen estas librerías, debemos tener instaladas en nuestro sistema las librerías *libtiff3* y *libjpeg62*. Una vez termine la instalación de RTI DDS debemos de hacernos con un archivo de licencia, en nuestro caso fue provisto por Catec, y lo copiamos en la carpeta de instalación de RTI.

Lo siguiente es instalar el paquete debian creado por el equipo de desarrollo de ANIMO DDS de Catec, el cual se instalará con el correspondiente comando desde nuestra terminal de Ubuntu.

Una vez tenemos todas las librerías instaladas correctamente podemos hacer la implementación de nuestra primera prueba de comunicaciones mediante ANIMO DDS.

Uso de las librerías de ANIMO DDS: El uso es bastante sencillo, sólo basta con inicializar ANIMO obteniendo una instancia del framework a partir de un archivo de configuración en el que se indica el dominio o “canal” por el que se van a transmitir los datos, las calidades de servicios, la ruta del fichero donde se encuentran la definición de calidades, la ruta del log y el identificador del log. A partir de este objeto crearemos dos instancias de los llamados “Access Points” que son los objetos que usaremos para publicar y suscribirnos. La dinámica de funcionamiento del patrón publicador subscriptor es muy sencilla, una aplicación publica un dato bajo un tópico y éste dato es recibido por todas las aplicaciones que estén suscritas a ese dato con ese tópico, así mediante el tópico podemos decidir qué datos vamos a recibir y cuáles no, en nuestra aplicación asignaremos como tópico el identificador correspondiente al programa que ha enviado ese dato, por tanto el resto de módulos se suscribirán al identificador correspondiente a los módulos que envían información que él necesita. Entonces nos crearemos un objeto AccessPoint-

ForSending y otro objeto AccessPointForReceiving.

A la hora de enviar un dato lo único que necesitamos hacer es tomar la instancia de nuestro AccessPointForSending y llamar a su método sendInterfaceData(data) donde "data."es cualquier objeto que herede de la clase `IData`" de ANIMO DDS. La recepción del dato se hará mediante listeners, un listener es un objeto que tendrá un método process el cual será llamado cuando haya llegado un dato que esté destinado a ese listener proporcionándole el dato. Para que ese listener reciba ese dato debemos registrarlo mediante el framework de ANIMO, más concretamente haciendo uso del AccessPointForReceiving el cual tiene dos métodos, uno llamado registerInterface(Type, listener) el cual recibe un enumerado con el tipo de dato que se va a recibir en ese listener y un listener que debe heredar del tipo de ANIMO `IDataListener` el cual posee un método process que es el que recibirá el dato.

Los tipos que pueden ser enviados mediante ANIMO debemos definirlos antes y, haciendo uso de las herramientas de generación de código de ANIMO DDS, recoger sus propiedades dentro de un archivo de configuración a partir del cual ANIMO automáticamente generará las clases necesarias para el envío de ese dato por DDS y hará las modificaciones pertinentes en el framework, incluido el añadir al enumerado de tipos que se pueden enviar por ANIMO el nuevo dato, enumerado que necesitaremos en el AccessPointForReceiving. Después de esto debemos generar los paquetes debían de nuevo mediante un script incluido en el proyecto ANIMO DDS y volver a instalarlos.

El siguiente paso es indicarle AccessPointForReceiving que va a recibir datos de un tópico para empezar a escuchar, por tanto se llamará a su método startDevice(Type, Topic) donde "Type" volverá a ser el enumerado con valor correspondiente al tipo de dato que vamos a recibir y "Topic" será el tópico bajo el que se va a enviar ese dato.

Así pues en el process del listener se define qué es lo que se va a hacer con ese dato. En nuestro caso la acción era modificar la posición de un punto en nuestra ventana según el valor de movimiento que nos hubiera mandado el joystick.

4.2.2. IMPLEMENTACIÓN DE LA CAPA DE COMUNICACIÓN

DESPUÉS de la implementación del proyecto de prueba de comunicaciones con ANIMO se definió una estructura a seguir para el tratamiento de los datos. Según ésta estructura se definiría una clase contenedor para todos los elementos referentes a las comunicaciones y un tipo de listener especial para nuestro proyecto. A continuación explicaremos cómo están formados estos elementos.

CONTENEDOR

El contenedor es la clase principal de nuestras comunicaciones mediante ANIMO, ésta estará encargada de establecer los elementos necesarios para iniciar las comunicaciones (creación de los Access Points, registros, etcétera...), crear los listeners que nos harán falta, almacenarlos y proveer al resto del proyecto de los métodos necesarios tanto para

realizar un envío de un dato mediante DDS como para acceder a los datos que nos han llegado.

El constructor del container tiene como parámetros de entrada el archivo de configuración necesario para la inicialización de ANIMO y el dominio, ambos archivos los incluiremos en el archivo de configuración general de la aplicación que tendrá los parámetros característicos de cada módulo a parte de las configuraciones necesarias para las comunicaciones a través de ANIMO. También será encargado de inicializar el framework de ANIMO y de crear los Access Points, tanto el de subscriptor como el de publicador. Y por último inicializará los listeners necesarios para nuestro proyecto.

Uno de los métodos necesarios es el de subscribe(), el cual a partir de la lista de listeners que hemos creado y el tópico identificador para el módulo que estamos implementando subscribirá cada uno de los listeners para poder iniciar la recepción de datos.

También dispondrá de un método de envío que recibirá un dato de ANIMO a publicar y hará la llamada al AccessPointForSending que realizará el envío.

Por último, esta clase dispondrá de una serie de métodos observadores que devolverán cada uno de los datos a los que nos hemos suscrito.

LISTENERS

Debido a que queremos mantener un control sobre el flujo de datos entre los sistemas de que disponemos y sobre la ejecución de la simulación tenemos que marcar unas pautas que definan qué se va a hacer con los mensajes cuando nos llegan. Una opción consiste en hacer en el process del listener la llamada a la clase que vaya a usar ese dato o un controlador de datos que lo redirija al objeto en el que va a ser usado, esto podría darnos problemas y complicar mucho el flujo de ejecución elevando el riesgo de que se entre en condiciones de carrera, etcétera.

La segunda opción, por la que finalmente optamos fue la de almacenar dentro del listener los datos nuevos cuando llegan, por lo que los listeners tienen un atributo del tipo del dato que están esperando recibir, el cual se actualiza en el momento en que se comprueba en el process que el dato es el que queremos, almacenándose en ese atributo.

El listener nos permite la visualización de ese dato almacenado mediante un método get que devuelve el tipo correspondiente al listener.

Así cuando durante la ejecución de nuestra aplicación necesitemos alguno de los datos, solo tenemos que realizar una espera activa comprobando si tenemos algún dato nuevo en el listener correspondiente al objeto que necesitamos, al que podemos acceder mediante los métodos provistos en la clase contenedor de comunicaciones.

4.2.3. IMPLEMENTACIÓN DEL MODELO SIMPLE

DBEBEMOS trabajar haciendo uso de un modelo que simule el comportamiento de una aeronave no tripulada, la estructura de ésta aeronave simulada será por un lado

un control que hará de autopiloto, y por otro lado un modelo en Matlab Simulink que será el que haga las veces de sensores y respuestas del avión. Para familiarizarnos con el comportamiento y uso de éstas tecnologías vamos a implementar un llamado *modelo bicicleta* o modelo simple.

PRIMERA APROXIMACIÓN

Simulink nos autogenera un código en C++ con el modelo y una serie de métodos para introducir los parámetros de entrada del modelo y un método para ejecutar un paso de simulación. La entra de este modelo simple es el *waypoint* o coordenada al que debe dirigirse y como salida irá mostrando su posición.

Para ésta prueba se realizó un nuevo proyecto que creaba una envoltura al código generado por Simulink, de manera que se le proporcionaba una serie de waypoints y un módulo secuenciador era el encargado de modificar el waypoint objetivo al modelo, ejecutar los pasos de simulación y enviarle el siguiente waypoint cuando hubiera llegado al anterior.

Una vez tuvimos éste sistema funcionando ampliados su funcionamiento para poder usar órdenes externas a través de ánimo, de forma que el control de la ejecución del paso de simulación fuera activada exteriormente, así como la información de la misión que debía realizar.

CAMBIOS PARA IMPLEMENTACIÓN EN EL SISTEMA

La envoltura que se realizó fue una clase llamada AircraftContainer la cual creaba e inicializaba las clases autogeneradas por Matlab Simulink y se encargaba de almacenar los objetos necesarios para el manejo de éste modelo, como por ejemplo el contenedor de comunicaciones, el controlador de ejecución, la clase encargada de secuencia los waypoints al modelo, el controlador de la posición y la encargada del manejo de la carga de misión en nuestro proyecto.

Tiene un método para ejecutar el step del modelo, de forma que primero comprueba si se ha enviado una orden de ejecución desde fuera ya que la ejecución será controlada desde otro módulo. Cuando nos llega una orden de ejecución del paso de simulación, llamamos a la ejecución del controlador de posición que se encargará de modificar el estado de la aeronave y de llamar al guionado que en última instancia será el que compruebe si hemos llegado al waypoint y luego, el controlador de posición llamará a la ejecución del modelo.

Otro método llamado checkMissionRequest comprueba que haya una petición de inicio de carga de misión, el cuál comprobará que no se haya terminado de cargar una misión en curso, y, si se dan las condiciones, inicializar el control de posición.

Por último, checkMissionStartCommand comprueba si ha llegado un comando de inicio de misión, en cuyo caso devolverá verdadero.

La carga de misión se realizará siguiendo el proceso de carga definido por el protocolo Mavlink y será gestionado por la clase MissionLoader, la cual será ejecutada por el contenedor de la aeronave.

Así pues, a cada paso de ejecución, si no hay una misión cargada y llega una solicitud de carga de misión se inicia el proceso de petición de waypoints, una vez está cargada la misión se espera a que se envía una señal de inicio de misión, cuando se cumplen estas dos premisas se comienza la ejecución del modelo cuyos datos recibe mediante el módulo de guiado.

El siguiente paso que realizaremos será el de crear la primera versión del módulo central de control de la simulación, el cuál enviará el paso de simulación y mediante un programa de prueba que haremos podrá validar el funcionamiento de la implementación del modelo simple.

4.2.4. DIAGRAMA DE BURNDOWN

Para este sprint se estimaron 8 puntos, de los cuales se han realizado 11, debido a que una historia de usuario que tenía 3 puntos, concretamente la de set up del entorno se ha realizado en este segundo sprint.

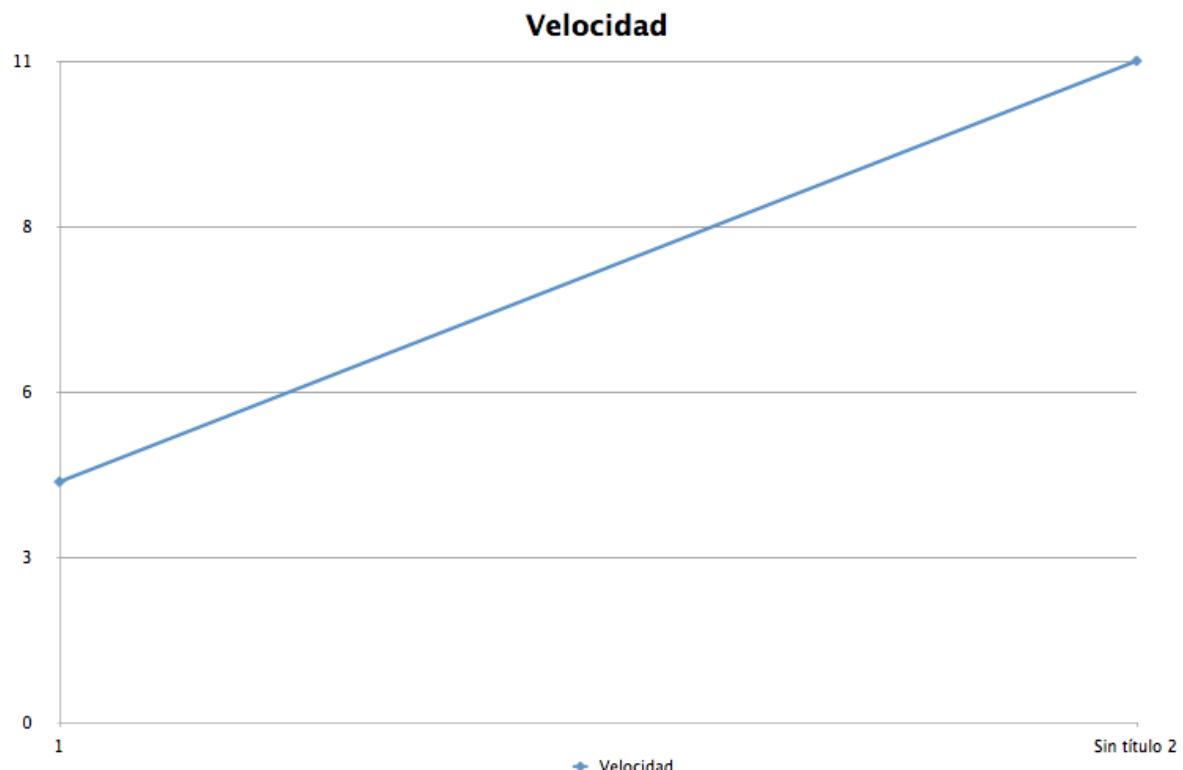


Figura 4.11: Diagrama de velocidad para el segundo sprint.

4.2.5. DIAGRAMA DE EVOLUCIÓN

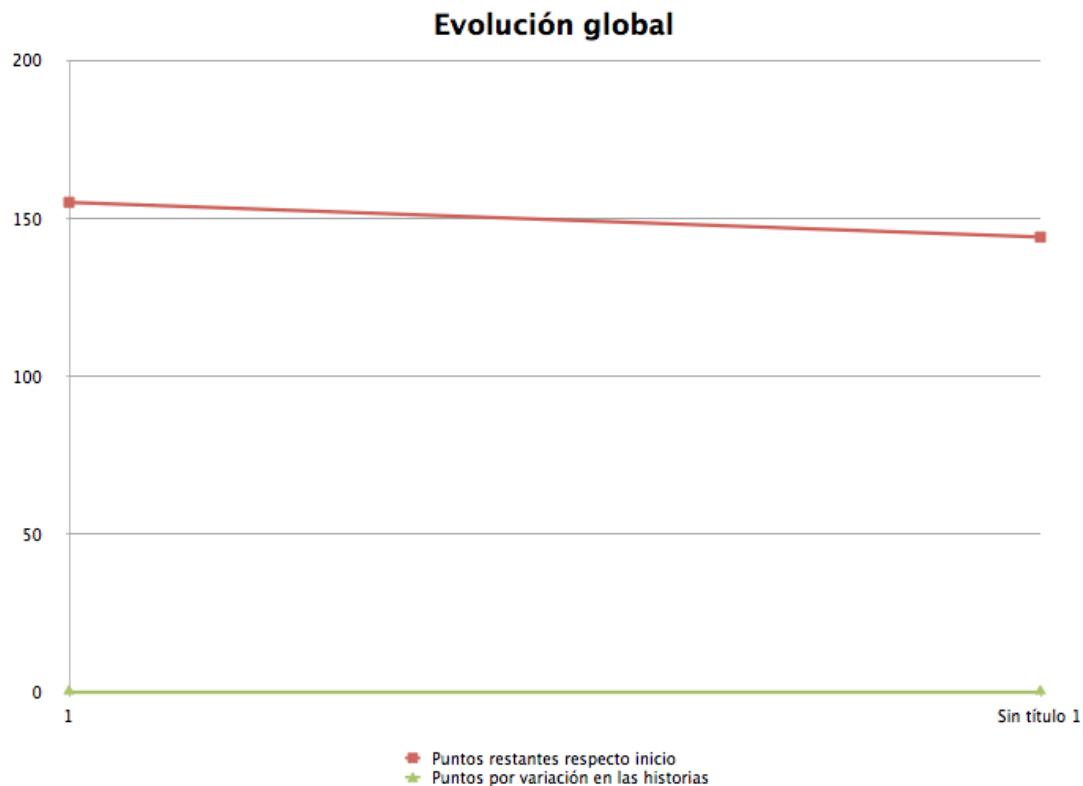


Figura 4.12: Diagrama de evolución para el segundo sprint.

4.3. SPRINT 3: CREACIÓN DEL SIMCORE

4.3.1. DIAGRAMA DE BURNDOWN

4.4. SPRINT 4: IMPLEMENTACIÓN DEL MODELO DE LOCOMOVE

4.4.1. DIAGRAMA DE BURNDOWN

4.5. SPRINT 5: GESTIÓN DE CARGA DE LA MISIÓN

4.5.1. DIAGRAMA DE BURNDOWN

4.6. SPRINT 6: PUESTO DE INSTRUCTOR

4.6.1. DIAGRAMA DE BURNDOWN

4.7. SPRINT 7: PRIMERA VERSIÓN Y CAMBIOS EN EL OBJETIVO

4.7.1. DIAGRAMA DE BURNDOWN

4.8. SPRINT 8: CAMBIOS EN EL DISEÑO

4.8.1. DIAGRAMA DE BURNDOWN

Parte IV

Conclusiones

Parte V

Apéndices
