eman ta zabal zazu

ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA INDUSTRIAL DE BILBAO

GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

2013/ 2014

MÓDULO DE IDENTIFICACIÓN DE PASOS Y SITUACIONES (MIPS)

MEMORIA

DATOS DE LA ALUMNA O DEL ALUMNO	**DATOS DEL DIRECTOR O DE LA DIRECTORA**
NOMBRE: RUBÉN APELLIDOS: AGUDO SANTOS FDO.: FECHA: 13/06/2014	NOMBRE: MIKEL APELLIDOS: VILLAMAÑE GIRONÉS DEPARTAMENTO: LSI FDO.: FECHA: 13/06/2014

Anexo II

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

El Trabajo de Fin de Grado (a partir de ahora TFG) aquí presentado fue propuesto por un profesor de la universidad. Las razones de la aceptación del TFG fueron que se veía una continuidad, es decir, que se podía utilizar el TFG como entrada a un grupo de investigación y/o continuar con el trabajo durante un máster.

El software ha sido liberado bajo licencia GPLv3 o posterior, y puede encontrarse en `https://github.com/RubenAgudo/MIPS`.

Figura 1.1: MIPS por Rubén Agudo está licenciado bajo una licencia Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional.

1.1. Planteamiento del problema

El problema que el software ULISES, diseñado por el grupo de investigación GaLan en colaboración con la universidad de Navarra, intenta resolver, es enseñar a alumnos habilidades. Es decir, ayudar a instruir a las personas cosas que no son fácilmente evaluables mediante un examen escrito.

ULISES es una metodología que permite unir un sistema interactivo que exista con un sistema educativo que también exista para poder evaluar la realización de tareas procedimentales o la adquisición de habilidades.

Un ejemplo de aplicación de ULISES es el software INTRASIM [1], en el cual el sistema interactivo es un simulador de camiones y el sistema educativo es un software llamado DETECTIVE.

Los datos capturados en bruto, es decir, las señales generadas por el sistema interactivo son trasladados a objetos del propio software, que son las Observaciones y las Propiedades.

De manera resumida, una Observación es un hecho interesante en el sistema y que quiere ser observado, y está compuesta por propiedades, que son las características que la definen. En el apartado de Antecedentes todo se explicará con mayor detalle, aquí solo se pretende rascar la superficie para conseguir una idea general.

Una vez creadas las observaciones con sus propiedades, se crean Situaciones y Pasos. Siendo una Situación el contexto en el que suceden las observaciones, y los Pasos cómo se comportan ciertas observaciones en el tiempo.

Finalmente, con esos Pasos y Situaciones, el sistema mediante distintas técnicas de diagnóstico es capaz de dar un veredicto a la habilidad que se quiere aprender, e indicándole qué ha hecho mal.

1.2. Justificación y propósito

Actualmente, la selección de pasos y situaciones se hace manualmente y con un gran componente de intuición, por lo que se hace evidente la necesidad de un software para facilitar la creación de Pasos y Situaciones porque no es una tarea trivial. La creación de relaciones puede ser muy compleja, y el software MIPS desarrollado en este TFG tiene como objetivo simplificar en la medida de lo posible esa tarea, aprovechando al máximo las capacidades cognitivas del cerebro humano para identificar patrones complejos.

Pese a visualizarse gráficamente, la identificación de esos patrones sigue sin ser algo sencillo. Es por ello que la tarea principal del software es facilitar la identificación de intervalos donde se producen los pasos y situaciones para posteriormente extraer de manera semiautomática las relaciones entre las observaciones y propiedades que forman parte de cada paso o situación. Esos pasos y situaciones vienen identificados por el experto en el dominio que es quien identifica qué hechos son relevantes. Por ejemplo, si se estuviera trabajando en el dominio del tenis, un experto podría indicar que para determinar

[1] `http://www.ceit.es/index.php?option=com_content&view=article&id=112&Itemid=132`

si la realización de un saque es correcto o no, un hecho relevante a tener en cuenta sería “levantar el brazo”, ya que dependiendo de la manera que se levante el brazo, el saque puede ser correcto o no. De este modo “levantar brazo” se convertiría en un paso en ULISES.

1.3. Definiciones, acrónimos y abreviaturas

TFG: Trabajo de fin de grado.

BD: Base de datos.

CVS: Control Version System.

MVVM: Model-View-ViewModel.

UI: User Interface. Interfaz de usuario.

MVC: Model-view-controller.

IDE: Integrated Development Environment. Entorno de desarrollo integrado.

W3C: World Wide Web Consortium.

Capítulo 2

Objetivos del proyecto

2.1. Objetivos

Los objetivos del proyecto consisten en crear un software que extienda la funcionalidad actual del sistema ULISES desarrollado por la universidad de Navarra y del grupo de investigación GALAN perteneciente a la UPV/EHU.

El software, debe permitir cargar observaciones y propiedades previamente capturadas con diversos sistemas interactivos (por ejemplo, con un dispositivo Kinect) y visualizarlas gráficamente. Esas observaciones y propiedades se cargarán en un panel lateral con forma de árbol y podrán seleccionarse para ser visualizadas. Los gráficos pueden ser reorganizados como el usuario quiera, para verlos al mismo tiempo.

También permitirá la reproducción de vídeos si están disponibles. El fin de la herramienta es facilitar al usuario experto la identificación de intervalos donde se producen los pasos y situaciones para posteriormente extraer de manera semiautomática las relaciones entre las observaciones y propiedades que forman parte de cada paso o situación.

2.2. Alcance

Se ha decidido utilizar las metodologías ágiles de desarrollo, como Scrum y Kanban en detrimento de los desarrollos clásicos como puede ser el desarrollo en cascada [1]. Al utilizar este tipo de metodologías se hace frente a uno de los mayores problemas del desarrollo de software: la incertidumbre y los cambios

[1] `www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf`

no contemplados inicialmente, facilitando la reacción ante los cambios al ser mucho más flexible.

Al utilizar un modelo de desarrollo ágil, el ciclo de vida que va a utilizarse, será iterativo e incremental.

Antes de exponer el alcance, es necesario saber en que consiste Scrum y Kanban.

2.2.1. Scrum

Scrum es una serie de herramientas (framework) para la gestión y desarrollo de software basado en un proceso iterativo e incremental (Scrum.org, s.f.).

En la figura 2.1 [2] se puede ver el método de trabajo según Scrum, en el cual se observa lo importante que son los entregables.

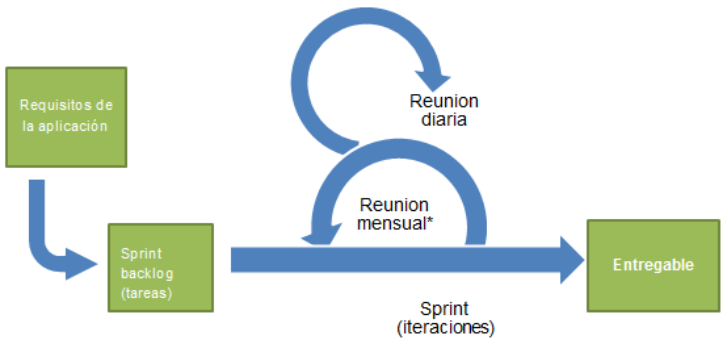


Figura 2.1: Diagrama Scrum

Scrum, de manera resumida consiste en lo siguiente:

1. Obtener la lista de tareas (Scrum backlog).
2. Mientras haya tareas en el Scrum backlog.
   a) Crear sprint backlog, es decir, las tareas de ese sprint.
   b) Programar las tareas del sprint.
   c) Reunión de retrospectiva para ver qué ha ido mal y mejorarlo.

[2] `https://upload.wikimedia.org/wikipedia/commons/e/e5/Scrumm.PNG` Autor: Maxie Ayala Licenciado bajo CC BY-SA 3.0

d) Enviar entregable al cliente.

e) Hablar con el cliente para ver si hay nuevos requisitos.

f) Repetir.

Es fundamental que entre sprint y sprint el valor del producto haya aumentado. Lo importante es el producto, no el avance del propio proyecto. Se podría avanzar mucho en el diseño del software, pero si eso es algo que el cliente no puede ver, se estará fallando en el objetivo de ser ágiles, ya que no le aporta nada.

Por otro lado, si se comete algún error será muy sencillo corregirlo ya que si los sprints son, por ejemplo, de dos semanas, no se habrá perdido apenas tiempo. Y además para el cliente los entregables serán predecibles en el tiempo, y se podrá estimar mejor cuándo una funcionalidad presente en el *backlog* será introducida en el software. Esto último es cierto porque se sabe la cantidad media de requisitos que se entregan en cada sprint, por lo que se puede realizar una predicción rápida y razonablemente precisa.

Scrum es también muy interesante, y se posiciona como una alternativa muy potente frente al desarrollo en cascada, porque en la creación de software hay un componente de incertidumbre muy grande. No se sabe cómo, ni cuándo pueden cambiar los requisitos de un software. Por ello, toman de nuevo importancia los sprints.

En los modelos antiguos, si ya se había terminado la fase de análisis y diseño, y se requería una nueva funcionalidad era necesario paralizar el desarrollo del software y volver a analizar y diseñar. En cambio, con Scrum es posible integrar ese cambio en el siguiente sprint.

2.2.2. Kanban

Kanban es un método de organización del conocimiento del trabajo que se está realizando, con un gran énfasis en la entrega justo a tiempo sin sobrecargar al equipo (Scotland, s.f.).

Kanban se centra mucho en que lo importante no es empezar muchas cosas, sino que se finalice aquello que está empezado. Sirve sobre todo para ver de una manera visual:

- Qué hay pendiente
- Qué se está haciendo ahora mismo
- Qué se ha terminado.

El método Kanban es algo muy extenso, pero en este proyecto, simplemente se ha aplicado el tablero Kanban mezclado con Scrum. Cada mes, se crea una lista de tareas que se debe completar (Sprint Backlog) y se coloca en la columna de "Tareas pendientes", ordenadas por prioridad. Después, se coloca en la lista de "En progreso" como máximo dos tareas. Y no se empieza ninguna otra hasta que esas dos hayan sido pasadas a la columna de "Tareas finalizadas".

2.3. Descripción de las tareas

El alcance del proyecto va a definirse mediante un EDT, en el que se muestran las tareas de las que se compone este TFG.

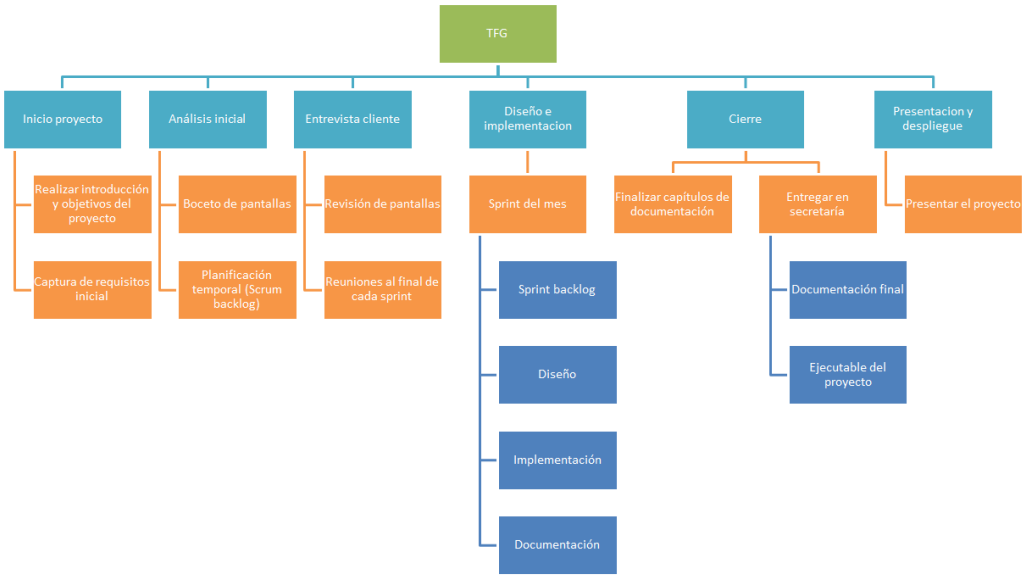


Figura 2.2: EDT

Tal y como se ve en la Figura 2.2 cada una de las tareas de los grupos "Análisis inicial", "Entrevista cliente" y "Diseño e implementación" van a realizarse en cada uno de los sprints. Para cada tarea se ha realizado una pequeña tabla en la que se detalla qué se va a hacer, la duración y esfuerzo aproximados, así como otros datos de interés.

2.3.1. Inicio proyecto

Realizar introducción y objetivos del proyecto

Duración (Horas / Persona)	7.
Descripción	Realizar las secciones de la memoria de la Introducción y los objetivos del proyecto.
Entradas	Scrum backlog
Salidas / Entregables	Los capítulos de introducción y objetivos del proyecto.
Recursos necesarios	PC, LaTeX, TeXStudio.
Precedencias	Ninguna.

Captura de requisitos inicial

Duración (Horas / Persona)	7.
Descripción	Entrevistarse con el cliente para obtener una captura de requisitos inicial.
Entradas	Ninguna.
Salidas / Entregables	Diagrama de casos de uso y modelo de dominio.
Recursos necesarios	PC, LaTeX, TeXStudio.
Precedencias	Ninguna.

2.3.2. Análisis inicial

Boceto de pantallas

Duración (Horas / Persona)	4.
Descripción	Diseño de pantallas inicial en base a las ideas obtenidas de la captura de datos inicial, a fin de obtener dudas sobre funcionalidades.
Entradas	Ninguna
Salidas / Entregables	Boceto dibujado a mano con el diseño de las pantallas
Recursos necesarios	Papel y lápiz
Precedencias	Ninguna

Planificación temporal (Scrum Backlog)

Duración (Horas / Persona)	7.
Descripción	Creación de todas las tareas para crear una pila.
Entradas	Requisitos del cliente.
Salidas / Entregables	Un listado con todas las tareas a completar
Recursos necesarios	PC con un editor de textos.
Precedencias	Requisitos del cliente.

2.3.3. Entrevista cliente

Revisión de pantallas

Duración (Horas / Persona)	2.
Descripción	Mostrar al cliente, un prototipo inicial de pantallas para así poder mejorar el diseño para que sea más fácil de usar.
Entradas	Requisitos del cliente.
Salidas / Entregables	Prototipo realizado en WPF que muestra el comportamiento y forma de las pantallas.
Recursos necesarios	Visual Studio 2013, Avalon Dock, un PC.
Precedencias	Captura de requisitos inicial.

Reuniones al final de cada sprint

Duración (Horas / Persona)	2.
Descripción	Reunión para que el cliente vea como va el trabajo, aclarar dudas etc.
Entradas	El trabajo realizado hasta el momento.
Salidas / Entregables	Ninguno / acta de reunión
Recursos necesarios	PC con el programa funcional.
Precedencias	Ninguno.

2.3.4. Diseño e implementación

Sprint backlog

Duración (Horas / Persona)	1
Descripción	Elegir las tareas que van a conformar el sprint del mes.
Entradas	La lista de tareas completa
Salidas / Entregables	Post-it con las tareas seleccionadas para ese sprint
Recursos necesarios	Post-it, bolígrafo y lista de tareas.
Precedencias	Scrum backlog

Diseño

Duración (Horas / Persona)	10.
Descripción	Realizar el diseño de software, clases, Interfaces necesarias, qué patrones van a usarse etc en el sprint del mes.
Entradas	Tarea a realizar y la interfaz de usuario.
Salidas / Entregables	Diseño de software que guiará el sprint del mes y la documentación habrá sido actualizada en consecuencia.
Recursos necesarios	La documentación realizada hasta el momento, Visual Studio 2013, Git y un PC.
Precedencias	Sprint Backlog.

Implementación

Duración (Horas / Persona)	20.
Descripción	Implementar lo diseñado, lo que describe la tarea en el post-it del sprint.
Entradas	Diseño realizado sobre la tarea, post-it con el sprint del mes.
Salidas / Entregables	La pieza de software con las nuevas funcionalidades implementadas, y la documentación actualizada.
Recursos necesarios	La documentación realizada hasta el momento, Visual Studio 2013, Git y un PC.
Precedencias	Diseño.

Documentación

Duración (Horas / Persona)	10.
Descripción	Revisar y actualizar la documentación realizada durante el sprint.
Entradas	La documentación realizada hasta el momento.
Salidas / Entregables	La documentación actualizada.
Recursos necesarios	La documentación.
Precedencias	Implementación.

2.3.5. Cierre

Finalizar capítulos de documentación

Duración (Horas / Persona)	30.
Descripción	Realizar los capítulos adicionales: Verificación y evaluación, conclusiones y trabajo futuro etc.
Entradas	La documentación realizada.
Salidas / Entregables	La documentación terminada.
Recursos necesarios	LaTeX, Git, y un PC.
Precedencias	Diseño e implementación (categoría).

Documentación final

Duración (Horas / Persona)	1
Descripción	Entregar la documentación en secretaría
Entradas	La documentación.
Salidas / Entregables	Ninguna.
Recursos necesarios	La documentación impresa.
Precedencias	Finalizar capítulos de la documentación.

Ejecutable del proyecto

Duración (Horas / Persona)	1.
Descripción	Entregar el ejecutable del proyecto realizado.
Entradas	El código fuente del programa.
Salidas / Entregables	El ejecutable binario.
Recursos necesarios	PC.
Precedencias	Finalizar capítulos de la documentación.

2.3.6. Presentación y despliegue

Presentar el proyecto

Duración (Horas / Persona)	20 minutos.
Descripción	Presentar el proyecto y defenderlo ante el tribunal.
Entradas	Slides de la presentación.
Salidas / Entregables	Ninguna.
Recursos necesarios	PC con LibreOffice Impress.
Precedencias	Categoría cierre.

2.4. Planificación temporal

Al seguirse una metodología ágil de desarrollo, la planificación se ha dividido en sprints. Ciertamente, detallar los sprints de antemano, no es una buena idea, ya que se estaría cayendo en el mismo error de el desarrollo en cascada. Es decir, el sprint 1, nos lleva al sprint 2, y el 2 al 3 y así sucesivamente.

Lo que se hace normalmente, es elegir lo que se va a hacer en el primer sprint, y hasta que no se termina, no se elige qué se va a hacer en el segundo, aumentando la flexibilidad a los cambios y los imprevistos.

Por lo tanto, se ha detallado una planificación de los sprints como algo orientativo, nadie garantiza que lo aquí descrito vaya a ser igual a lo que se va a realizar.

2.4.1. Mi planificación

Cada sprint tiene una duración de un mes, empezando desde Octubre del 2013. El esfuerzo en horas que se va a realizar se decidirá al inicio de cada sprint, en función de algunos parámetros, por ejemplo, si se va con retraso, si hay entregables de la universidad pendientes, otros trabajos etc. Pese a ese componente variable, la cantidad media de horas que se trabajará en el proyecto por sprint, será de 60 horas. Es decir, 3 horas diarias de lunes a viernes.

Gantt

Se ha creado un diagrama de Gantt para que de una manera visual pueda verse la distribución de los sprints, y más adelante se detallará en que consiste cada sprint.

Tarea	Inicio	Fin
Sprint 1	01/10/13	31/10/13
Sprint 2	01/11/13	30/11/13
Sprint 3	01/12/13	31/12/13
Sprint 4	01/01/14	31/01/14
Sprint 5	01/02/14	28/02/14
Sprint 6	01/03/14	31/03/14
Sprint 7	01/04/14	30/04/14

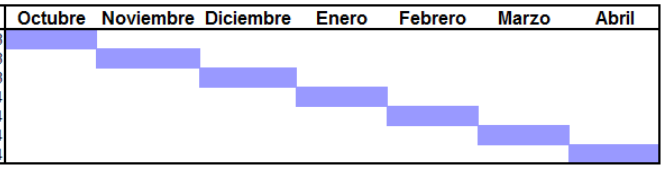


Figura 2.3: Diagrama de Gantt

Sprint 1

Inicio del proyecto, establecer prioridades, documentarse etc. Lo que compone este sprint es:

- Crear y configurar el repositorio Git y los clientes en los dos ordenadores.
- Leer la documentación existente y obtener el código fuente original.

- Iniciar diseños gráficos preliminares en papel para elegir las bibliotecas necesarias.
- Buscar y documentarse sobre las bibliotecas que cumplan los requisitos del punto anterior.

Sprint 2

Una vez configurados los ordenadores y elegidas las bibliotecas necesarias:

- Reunirse con el cliente para realizar una captura de requisitos.
- Diseñar unos prototipos funcionales.
- Documentar lo que se había realizado hasta el momento, es decir, el tipo de planificación que se había elegido, tecnologías que se iban a usar etc.

Sprint 3

El siguiente paso consiste en poder cargar vídeos, tantos como se quieran, desde el sistema de archivos.

- Crear el control de cargar vídeos (contenedor de vídeos). El contenedor debe permitir:
  - Cargar distintos vídeos desde el sistema de archivos.
  - Cada vídeo debe disponer de sus controles individuales. Play, pausa, stop y avanzar de tantos segundos en tantos segundos, que esa funcionalidad sea configurable por el usuario mediante un archivo de configuración y poder silenciar el vídeo.
  - El contenedor de vídeos debe disponer de unos controles generales que ofrezcan idénticas funcionalidades que las del punto anterior.
  - Debe permitir establecer para cada vídeo el momento de inicio del vídeo manualmente para mantener la sincronización.

Sprint 4

Después de terminar la visualización de vídeos, lo siguiente en la lista de tareas es crear un visualizador de datos.

- Crear el contenedor que nos permita ver datos. El control debe permitir:
  - Visualizar datos continuos y discretos.

- Seleccionar un rango en la gráfica, estableciendo un inicio y un fin para las propiedades.
- Que muestre el progreso, y que esté sincronizado con el vídeo.
- Guardar el inicio y fin seleccionado.

Sprint 5

Ya se tiene todo preparado para funcionar, ahora hay que poder guardar datos.

- Elegir el método de guardado de los datos.
- Configurar la base de datos (si aplica).
- Crear el diagrama Entidad-Relación que representa el modo de guardado de datos (si aplica).
- Añadir al código existente las llamadas necesarias a la base de datos (si aplica).
- Permitir cambiar la ubicación de la base de datos mediante un archivo de configuración (Si aplica).

Sprint 6

En este momento se dispone de una aplicación completamente funcional pero con datos de prueba. Hay que poder cargar datos desde un origen de datos.

- Crear la clase para interactuar con el origen de datos.
- Hablar con el cliente para saber como están guardados los datos.
- Utilizando los datos de origen cargar la aplicación.

Sprint 7

Este último sprint es sobre todo para hablar con el cliente para ver si sus requisitos iniciales satisfacen sus necesidades actuales y terminar la documentación así como una revisión general del código para posibles optimizaciones, refactorizado final, añadir o quitar comentarios en función de su necesidad.

También se finalizará la documentación y memoria para dejarla para ser presentada.

Este sprint final es para dar por cerrado el proyecto.

2.5. Herramientas

El software MIPS al requerir de una cierta compatibilidad con el proyecto ULISES, no se ha tenido libertad total en la elección de herramientas.

Las herramientas que se han utilizado para llevar a cabo este proyecto han sido:

- **Visual Studio**

  El único software en el que no se ha tenido libertad de elección. Debido a que el software ULISES, en el que MIPS será integrado, está desarrollado usando Visual Studio, se ha utilizado por compatibilidad. Además, se estaba valorando la posibilidad de portar ULISES a WPF por lo que Visual Studio se convertía en una elección obvia ya que es un software muy potente diseñado para trabajar e integrarse perfectamente con esas tecnologías.

  La versión utilizada ha sido Visual Studio 2013 Ultimate, proporcionado de manera gratuita gracias al acuerdo que mantiene la UPV/EHU con Microsoft.

- **TeXstudio y LaTeX**

  Dos herramientas gratuitas y de código abierto para generar documentos con el lenguaje de programación LaTeX.

- **Git y GitHub**

  Fundamental en los proyectos que se basen en metodologías ágiles. Git es un software de gestión de control de versiones distribuido, esto es, que cada desarrollador dispone de una copia completa del repositorio. Desarrollado por Linus Torvalds y Junio Hamano en 2005 y similar a SVN.

  GitHub es un repositorio online, gratuito para proyectos Open Source desarrollado y mantenido por GitHub Inc. Se ha elegido por ser la alternativa más conocida así como punto de referencia de los desarrolladores a la hora de alojar proyectos.

2.6. Gestión de riesgos

Como en todo proyecto de software, existen una serie de riesgos que hay que tener en cuenta y en la medida de lo posible, evitarlos. Es por ello que se necesita tener en cuenta las probabilidades de los sucesos que pueden ocurrir y que puedan retrasar el trabajo de forma notable. Una vez listados, se puede proceder a crear un plan de prevención para evitarlos. Y en caso de que la prevención no funcionase, un plan de contingencia que pueda amortiguar las consecuencias de ese riesgo. A continuación se listan de forma detallada los riesgos que pueden aparecer durante el transcurso del proyecto.

2.6.1. Planificación

En esta categoría se engloban los riesgos que pueden causados por una mala planificación temporal.

Mala concepción del proyecto

Descripción	Por decisiones erróneas, como puede ser elegir utilizar una base de datos, puede que el trabajo deba ser reconstruido desde el principio.
Prevención	Una buena planificación y conocimiento exacto de lo que quiere y necesita el cliente.
Plan de contingencia	Un programa flexible a cambios.
Probabilidad	Improbable.
Impacto	Muy alto.

Error en la planificación temporal

Descripción	Por una mala comprensión y falta de experiencia, el cálculo de tiempo estimado de una de las tareas es erróneo y hay que reajustar todo el proyecto.
Prevención	Realizar la estimación lo mejor posible y tener una buena comunicación con el cliente para no tener fallos de comprensión. Utilizar de manera correcta las metodologías ágiles.
Plan de contingencia	Margen de error en los tiempos.
Probabilidad	Muy probable.
Impacto	Medio.

2.6.2. Entorno de desarrollo

Incompatibilidad de versiones

Descripción	Al utilizar dos ordenadores, una workstation y un portátil, puede que no se disponga de las mismas versiones del software o que no se satisfagan todas las dependencias.
Prevención	Asegurarse de disponer el mismo software con las mismas versiones.
Plan de contingencia	Resolver manualmente las dependencias.
Probabilidad	Probable
Impacto	Bajo

Problemas con GIT

Descripción	Al ser algo no enseñado en profundidad en la universidad puede causar problemas al usar las funciones avanzadas.
Prevención	Tener un buen manual y tener una copia local en otra carpeta para prevenir perdidas de información.
Plan de contingencia	Desechar el software de control de versiones y buscar alternativas como Dropbox.
Probabilidad	Poco probable
Impacto	Bajo

Problemas con las librerías

Descripción	Que las librerías escogidas no cumplan los requisitos necesarios
Prevención	Mirar con anterioridad que sí los cumplan.
Plan de contingencia	Buscar una alternativa o intentar parchear la actual, ya que son de código abierto.
Probabilidad	Muy probable
Impacto	Alto

2.6.3. Usuarios finales y clientes

No agrado del producto

Descripción	Después de la entrega del proyecto puede suceder que el cliente no esté satisfecho.
Prevención	Una buena comunicación con el cliente y reuniones progresivas para ir mostrando la evolución del proyecto.
Plan de contingencia	Blindar las características, solo se realizarán las estipuladas durante el proyecto.
Probabilidad	Probable
Impacto	Muy alto

Manejo poco intuitivo

Descripción	Los programadores al tener un conocimiento informático superior no tendrán problemas con el uso, pero los usuarios sí pueden tenerlos.
Prevención	Un diseño de interfaz sencillo y probar versiones con usuarios con conocimientos informáticos bajos.
Plan de contingencia	Adaptar la interfaz en el siguiente sprint.
Probabilidad	Probable.
Impacto	Alto

2.6.4. Personas

Bajas por enfermedad

Descripción	El único componente del equipo cae enfermo y no puede trabajar.
Prevención	No existe prevención.
Plan de contingencia	Recuperarse lo antes posible y manejar cierta holgura con las fechas.
Probabilidad	Probable.
Impacto	Muy alto

2.6.5. Requisitos

Nuevos requisitos

Descripción	Debido a que el cliente tiene nuevas necesidades solicita la implementación de funcionalidades extra o la desaparición de alguna que se haya podido volver innecesaria.
Prevención	Realizar un buen diseño para que la posibilidad de añadir funcionalidades no suponga un coste muy elevado.
Plan de contingencia	Evaluar los nuevos requisitos para saber si se pueden abordar o no y si fuese que sí, estudiar la situación y realizarlo.
Probabilidad	Probable
Impacto	Alto

Requisitos no contemplados (ocultos)

Descripción	Requisitos explícitos desde el principio, pero que no fueron vistos en el primer análisis y al verlos en un futuro, provocan cambios en todos los niveles.
Prevención	Utilizar correctamente Scrum.
Plan de contingencia	Añadirlo como una tarea al siguiente sprint.
Probabilidad	Muy probable.
Impacto	Medio-Bajo.

2.7. Planificación económica

Debido a que el software creado va a ser donado a un grupo de investigación, todos los datos utilizados en esta sección de la planificación económica se han realizado suponiendo que el proyecto se realiza al amparo del grupo de investigación como personal contratado y que se va a pedir financiación al Gobierno Vasco.

Al realizar esta suposición, es necesario crear una estimación de los costes para poder pedir la ayuda al gobierno y de esta forma llevar a cabo el proyecto.

2.7.1. Ingresos

No se obtendrá beneficio económico del proyecto ya que se realiza con carácter comunitario y con la intención de ayudar a un grupo de investigación. El código está disponible y será liberado bajo licencia GPLv3.

Por lo tanto el coste de MIPS será de 0€.

2.7.2. Salario del investigador responsable

Teniendo en cuenta la complejidad, y el tiempo disponible para realizar el software, se ha decidido dar un salario compensatorio de 30 € / hora al responsable desarrollador del proyecto.

Para calcular el salario total a percibir, se ha utilizado la cantidad estimada media de horas que se le dedicará al proyecto, que son 60 horas / mes. Por tanto:

$$Total\ horas = 60horas/mes * 7meses = 420horas$$

$$Total\ a\ percibir(€) = 420horas * 30€/hora = 12600€$$

Lugar de trabajo

El lugar de trabajo es el domicilio familiar y no hay que pagar alquiler ya que está en propiedad. Por lo que el gasto de alquiler/hipoteca es 0€.

2.7.3. Amortización material informático

Con la amortización cargamos al cliente con el desgaste que supone utilizar esos dispositivos para esa tarea. Como el proyecto ha durado siete meses, la amortización correspondiente a este año deberá ser multiplicada por $\frac{7meses}{12meses} = 0{,}582$ para cobrar de manera correcta por ese desgaste. Este factor se le aplicará tanto al software de pago como a los dispositivos informáticos utilizados.

Material informático

El material informático que se va a usar es un portátil Acer TravelMate 5742G comprado en Diciembre de 2010 y usado intensivamente durante estos 4 años. El portátil está pensado para ser amortizado en 8 años.

Por lo tanto:

El coste del portátil fue de 549€ IVA incluido.

$$Amortizacion\ anual(\,\%) = \frac{100}{8} = 12{,}5\,\% \tag{2.1}$$

$$Amortizacion\ anual(€) = \frac{Amortizacion\ anual(\,\%) * Precio\ dispositivo}{100} \tag{2.2}$$

$$Amortizacion\ anual(€) = \frac{12{,}5 * 549}{100} = 68{,}625€ \tag{2.3}$$

$$Amortizacion\ a\ cargar(€) = 68{,}625 * 0,583 = 39{,}79€ \tag{2.4}$$

$$Amortizacion\ acumulada(€) = 68{,}625 * 4 = 274{,}5€ \tag{2.5}$$

$$Amortizacion\ acumulada(2014) = 274{,}5 + 68{,}625 = 343{,}125€ \tag{2.6}$$

También se va a usar un PC genérico comprado a piezas en 2012, y pensado para ser amortizado en 10 años. El precio fue 277,65 € IVA incluido. Por lo que su amortización hasta el día de hoy ha sido:

$$Amortizacion\ anual(\,\%) = \frac{100}{10} = 10{,}0\,\% \tag{2.7}$$

$$Amortizacion\ anual(€) = \frac{Amortizacion\ anual(\,\%) * Precio\ dispositivo}{100} \tag{2.8}$$

$$Amortizacion\ anual(€) = \frac{10{,}0 * 277{,}65}{100} = 27{,}765€ \tag{2.9}$$

$$Amortizacion\ a\ cargar(€) = 27{,}765 * 0{,}583 = 16{,}187€ \tag{2.10}$$

$$Amortizacion\ acumulada(€) = 27{,}765 * 2 = 55{,}53€ \tag{2.11}$$

$$Amortizacion\ acumulada(2014) = 55{,}53 + 27{,}765 = 83{,}295€ \tag{2.12}$$

2.7.4. Software

Debido a la previa suposición de que se está trabajando de manera conjunta con la universidad como personal contratado, se hace necesario contabilizar como gasto las licencias de software.

Visual Studio

La versión de Visual Studio utilizada es la 2013 Ultimate, la más completa y a la vez más cara, y que su Precio de Venta al Público es de 14.237,91€ [3].

Como es un software muy caro, se le va a aplicar una amortización de 4 años. Parece poco tiempo, pero en términos de desarrollo es un periodo bastante largo, y además Microsoft ofrece unos precios razonables de actualización.

$$Amortizacion\ anual(\,\%) = \frac{100}{4} = 25{,}0\,\% \tag{2.13}$$

$$Amortizacion\ anual(€) = \frac{Amortizacion\ anual(\,\%) * Precio\ software}{100} \tag{2.14}$$

$$Amortizacion\ anual(€) = \frac{25{,}0 * 14237{,}91€}{100} = 3559{,}4775€ \tag{2.15}$$

[3] `http://www.visualstudio.com/en-us/products/visual-studio-ultimate-with-msdn-vs#Fragment_PricingHeader`

$$Amortizacion\ a\ cargar(€) = 3559{,}4775 * 0{,}583 = 2075{,}17€ \tag{2.16}$$

$$Amortizacion\ acumulada(€) = 3559{,}4775 * 0 = 0€ \tag{2.17}$$

$$Amortizacion\ acumulada(2014) = 0 + 3559{,}4775 = 3559{,}4775€ \tag{2.18}$$

Resto del software

El resto del software utilizado es gratuito y de código abierto, por lo que no suponen un gasto.

2.7.5. Otros gastos

Los gastos comunes del proyecto (Agua, Internet, Luz, Gas) supondrán un 5 % del coste total del proyecto.

2.7.6. Gastos totales

La tabla 2.1 resume los gastos estimados del proyecto, aquellos gastos que son 0€ no se añaden a la tabla.

Concepto	Gastos en €
Amortización material	55,977
Gastos personal (salario)	12.600,000
Software	2.075,170
Sub total	14.062,290
Gastos comunes	1.203,115
Total	15.265,405

Cuadro 2.1: Gastos totales

Capítulo 3

Antecedentes

3.1. Situación actual

El proyecto ULISES desarrollado por el grupo de investigación GaLan, en colaboración con la universidad de Navarra, consiste en un software que permite la integración de cualquier sistema interactivo con cualquier sistema educativo. Es decir, sirve para enseñar habilidades que necesitan de un profesor experto que los evalúe, ya sea en tiempo real, o a través de un vídeo.

La parte de los sistemas interactivos puede estar formada, por ejemplo, por un simulador de conducción de camiones, un sistema de captura de movimientos, un kinect, etc.

Por otro lado, el sistema educativo puede ser moodle, DETECTIVE o cualquier software que permita evaluar algo.

El objetivo principal es conseguir un sistema en el que la gente entrene una habilidad concreta, por ejemplo, aprender a sacar correctamente en tenis, y que el sistema sea capaz de evaluar, determinar dónde se ha equivocado, y en definitiva, ayudar a esa persona a aprender a realizar un saque de tenis correctamente.

Para poder realizar un diagnóstico, se han definido tres niveles.

1. Nivel de observación
2. Nivel de interpretación
3. Nivel de diagnóstico

3.1.1. Nivel de observación

En el nivel de observación lo que se hace es transformar los datos en bruto, es decir, las señales y valores generados por el sistema interactivo a los objetos que han sido definidos en el propio sistema: Observaciones y Propiedades.

Para poder determinar de manera correcta esas propiedades y observaciones, es necesario contar con un experto en el sistema interactivo, y un experto del dominio, que es quien define las observaciones y las propiedades que han de tenerse en cuenta. Nótese que el experto del dominio define las observaciones y las propiedades en un lenguaje natural, técnico en su campo del conocimiento, pero no en el ámbito del propio sistema. Por su parte, el experto en el sistema interactivo es quien define las observaciones y sus propiedades en el propio sistema en base a los datos, valores y señales que produce el sistema interactivo.

Observaciones y propiedades

Una observación representa un hecho interesante para el diagnóstico, y que además tiene sentido. Suelen ser lo suficientemente pequeñas para que sean fácilmente diagnosticables. Si la observación fuese "Realizar saque", sería una secuencia de varias acciones, por ejemplo: "lanzar pelota", "echar raqueta hacia atrás" y "golpear pelota". De esta manera se es mucho más granular y se pueden dar mejores diagnósticos (CEIT, 2013).

Cada observación contiene una serie de propiedades, que es lo que caracteriza una observación. Siguiendo con el ejemplo del tenis, unas posibles propiedades para la observación "pelota en movimiento" pueden ser la velocidad y la dirección de la pelota.

3.1.2. Nivel de interpretación

Este es un nivel clave, en el que se crean los Pasos y las Situaciones. Ambos objetos se crean en base a relaciones entre observaciones. Estas relaciones son necesarias, ya que no todas las observaciones que se capturan influyen en las demás, o no aportan información útil o relevante, y puede que incluso creen ruido, empeorando el diagnóstico.

Paso: Es el comportamiento de un conjunto de observaciones en el tiempo. Con un paso se define cómo se tienen que comportar las observaciones y sus propiedades en un periodo de tiempo concreto. Al igual que con las

observaciones, se suelen definir pasos pequeños, como por ejemplo, “levantar raqueta”, para realizar los diagnósticos más precisos.

Situacion: Define el contexto en el que suceden las observaciones y que influyen al Paso. Hay que tener en cuenta que sólo se definen las situaciones que tienen influencia en el paso para poder diagnosticar si lo que está haciendo el alumno es correcto o no. Por ejemplo, a la hora de tirar la pelota, habría que tener en cuenta el viento, para compensar su trayectoria. Si el alumno tirase la pelota demasiado hacia delante, y no hay viento, eso nos indicaría que está realizando la acción incorrectamente. Por el contrario, si hiciese viento en contra, la acción tendría un veredicto correcto porque se está lanzando hacia delante para compensar la fuerza del viento.

Actualmente, para crear las relaciones entre pasos y situaciones, hay una persona que lo realiza a mano. En muchos casos es un trabajo de prueba y error y además con resultados subóptimos ya que no siempre se ven claramente las relaciones, e incluso pueden crearse relaciones erróneas y que empeoren los futuros diagnósticos.

En esta situación es donde cobra sentido la realización de este TFG. El software MIPS, es un primer paso para facilitar el trabajo a la persona que crea esas relaciones. Es una herramienta de experto que de manera gráfica, permite acotar los datos procesados en el nivel de observación. Es decir, el software no recibe datos capturados por el sistema de realidad virtual, sino las propiedades y observaciones, con sus valores en el tiempo durante una sesión de trabajo.

Al acotar esos datos en distintos rangos se pueden definir pasos y situaciones tomando como referencia el tiempo y los vídeos en caso de existir, en qué momentos de la sesión de trabajo se está produciendo ese paso o esa situación que se quiere definir, de manera que se obtenga como resultado un fichero XML donde estarán almacenados todos los datos de aquellas observaciones y propiedades que el usuario haya seleccionado porque crea que son importantes para ese paso o situación.

Una vez creados los pasos y situaciones, se utilizarán algoritmos de *Machine Learning* y minería de datos para realizar un *Feature Selection* [1], y establecer las relaciones de manera automática.

[1] `http://jmlr.org/papers/volume3/guyon03a/guyon03a.pdf`

3.1.3. Nivel de diagnóstico

Finalmente, en este nivel, mediante distintas técnicas de diagnóstico, como por ejemplo el *Clustering* y algoritmos de clasificación supervisada se determina si el alumno ha adquirido la habilidad. Si aun no domina lo que deseaba aprender, se le proporciona el *feedback* necesario para saber donde tiene que incidir para poder mejorar.

3.2. Estudio de diferentes alternativas

Como hemos visto en la sección anterior, la aplicación debe poder visualizar tanto gráficos como vídeos capturados durante una sesión de trabajo. Si se decide añadir uno o varios vídeos, irán sincronizados con los gráficos, para poder identificar más fácilmente en que momentos está sucediendo algo que podría ser un paso o una situación.

Para conseguir este objetivo, se barajaron distintos tipos de acercamientos al problema.

3.2.1. Múltiples ventanas dentro de una ventana maestra (MDI)

La primera opción barajada fue crear una interfaz MDI (Multiple Document Interface). Como ventajas se disponía de cierta experiencia creando aplicaciones MDI en .NET.

Pero, estamos en 2014 y las ventanas MDI no son muy cómodas de utilizar cuando se disponen de varias ventanas abiertas, ya que no se acoplan automáticamente a los lados, ni se pueden crear pestañas de manera dinámica. Por otro lado, las ventanas MDI fueron desaprobadas en WPF, que es el nuevo sub sistema gráfico para renderizar interfaces de usuario de Microsoft.

3.2.2. Crear una interfaz tipo IDE

La mayoría de los IDEs conocidos, como Eclipse, Visual Studio, NetBeans IntelliJ Idea... disponen de un sistema de ventanas acoplables dinámicas. Pueden crearse grupos de pestañas, redimensionarlas, ponerlas lado a lado etc.

Debido a que el software va a desarrollarse con Visual Studio, era factible pensar que crear una interfaz similar al propio Visual Studio sería sencillo, con controles gráficos que facilitaran la tarea.

La realidad es bien distinta, ya que se necesita de componentes de terceros para crear ese tipo de interfaz. Pese a ese pequeño inconveniente hay distintas librerías de código abierto que proporcionan la nombrada funcionalidad.

Por tanto la alternativa que se tomó fue esta. Implementar una interfaz tipo IDE que permita el acoplamiento de ventanas, crear grupos de pestañas etc.

Capítulo 4

Captura de requisitos

4.1. Diagrama de casos de uso

En el diagrama de casos de uso, se puede ver que acciones puede realizar el usuario en cualquier momento, y cuales están supeditadas a ciertas condiciones. Nótese que las condiciones de los "extend" son los comentarios añadidos a los subcasos de uso.

Los casos de uso extendidos se encuentran en el anexo B

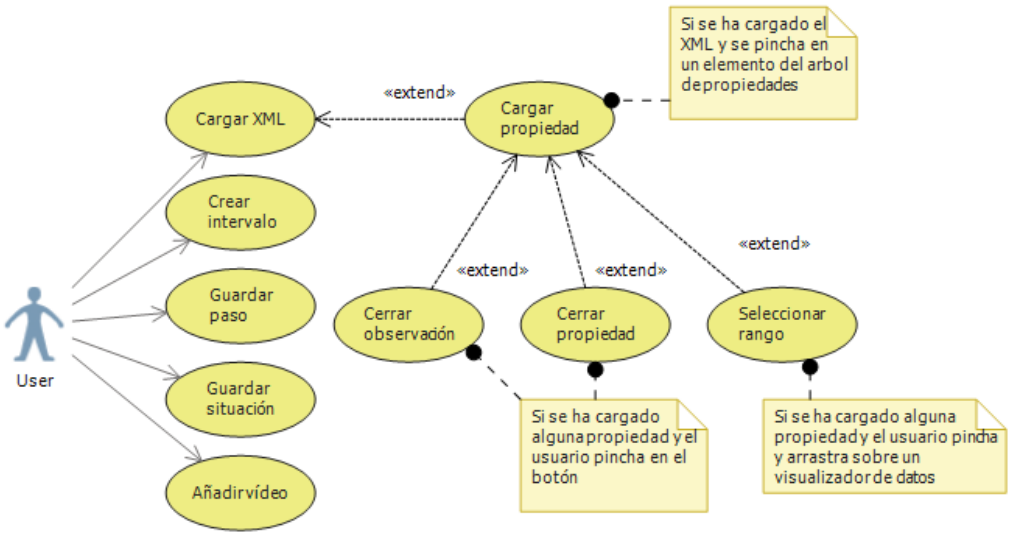


Figura 4.1: Diagrama de casos de uso

4.1.1. Cargar XML

Carga un XML que contiene las propiedades y las observaciones en el sistema. Únicamente se cargan si el fichero valida contra el XSD.

4.1.2. Añadir vídeo

Abre un vídeo y lo añade al contenedor de vídeos si ya había alguno cargado previamente. Si no, añade también un contenedor de vídeos.

4.1.3. Cargar propiedad

Carga una propiedad en el contenedor de gráficos correspondiente a la observación a la que pertenece. También se pueden añadir propiedades en grupo, haciendo doble click sobre el nombre de la observación. Si el contenedor de esa observación o propiedad no está creado, lo crea.

4.1.4. Seleccionar rango

Al pinchar y mover el ratón sobre un visualizador de datos, se seleccionará ese rango en todos los visualizadores de datos cargados en el sistema.

4.1.5. Cerrar observación

Elimina, tanto de la interfaz gráfica como de memoria, la observación seleccionada y todos sus visualizadores de datos (propiedades) asociadas.

4.1.6. Cerrar propiedad

Elimina, tanto gráficamente, como de memoria, la propiedad seleccionada.

4.1.7. Crear intervalo

Añade el intervalo seleccionado a los intervalos a guardar en el XML si no se superpone con los que ya están guardados.

4.1.8. Guardar paso

Guarda todos los intervalos creados a disco con un elemento raíz “step”.

4.1.9. Guardar situación

Guarda todos los intervalos creados a disco con un elemento raíz “situation”.

4.2. Modelo de dominio

La Figura 4.2 representa tanto los datos de entrada del software, tanto los datos de salida.

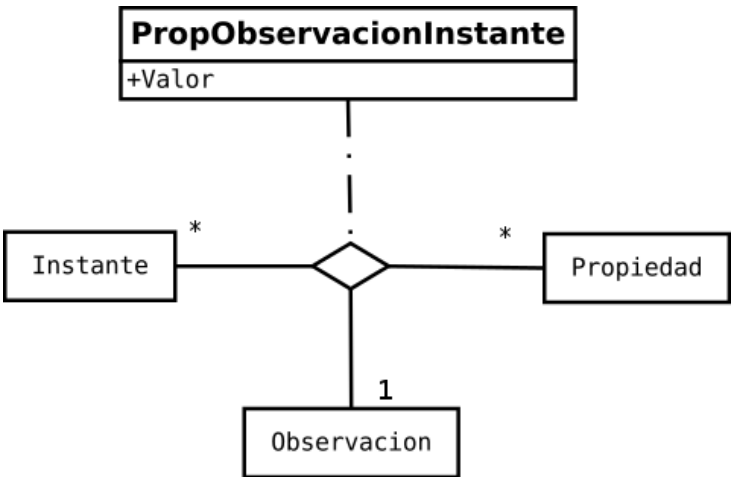


Figura 4.2: Modelo de dominio

4.2.1. Observación

La entidad observación la compone una lista propiedades y un nombre único, que describe la observación, por ejemplo “pelota en movimiento”.

4.2.2. Propiedad

Las propiedades están formadas por una serie de instantes, y un nombre de propiedad, que detalla el significado de la propiedad, por ejemplo,

“velocidad”.

4.2.3. Instante

Los instantes se componen de un atributo “instante de simulación”, que indica en que instante está sucediendo dicho Instante.

4.2.4. PropObservacionInstante

Tal y como se ve en la Figura 4.2 para una observación y una propiedad, hay muchos instantes, para un instante concreto y una observación hay múltiples propiedades, y para una propiedad y un instante, hay una única observación.

Podría pensarse en no utilizar una relación ternaria, sino dos binarias. Para explicar porque no puede ser, lo mejor es utilizar un ejemplo. Supongamos que tenemos las observaciones Obs1 y Obs2, así como las propiedad Prop1, Prop2 y Prop3, y los instantes Ins1, Ins2.

Obs1 contiene: Prop1 y Prop3. Obs2 contiene: Prop1 y Prop2.

Nótese que las propiedades aunque tengan el mismo nombre, no son la misma propiedad si están en observaciones distintas. Llegados a este punto, estaríamos incumpliendo una de las restricciones del modelo de dominio, que dice que en una relación binaria, una pareja de datos solo puede darse una vez, y es factible pensar que existirá tanto (Prop1, Ins1) para la Obs1 como para la Obs2.

Como esa restricción no puede incumplirse, se utiliza una relación ternaria, en la que se tiene la certeza que la tripleta (Obs1, Prop1, Ins1) no va a repetirse. Finalmente, para esa tripleta se guarda el valor que se ha seleccionado.

Capítulo 5

Análisis y diseño

5.1. Análisis

Los datos que ha de guardar el software estarán en un XML. El XML se divide en:

- Observaciones
- Propiedades
- Instantes

Por tanto se ha decidido guardar la lista de Observaciones, cada observación contiene la lista de Propiedades y cada Propiedad una lista de Instantes en los que se guardarán los valores.

El rango seleccionado en cada propiedad estará sincronizado con el resto de propiedades cargadas. Ya que no tiene sentido que por ejemplo, la observación "coche", que tiene como propiedades "velocidad" y "aceleración" tengan distintos rangos seleccionados, ya que si en "velocidad" seleccionamos [1, 3] y en "aceleración" [2, 4], ¿Qué guardamos? ¿El primero o el segundo? Por tanto, ya que todas las propiedades suceden al mismo tiempo se sabe que están relacionadas, por lo que es necesario que los rangos estén sincronizados.

A la hora de guardar, tampoco podrán guardarse distintos rangos que se solapen con los previamente creados para un mismo paso o situación. La razón por la que no se pueden solapar es que no tiene sentido definir que algo está ocurriendo dos veces exactamente en el mismo instante de tiempo. Si se dice que "saltar"se produce en el intervalo [1, 3] y en el [2, 5], en los momentos 2 y 3... ¿Se está saltando dos veces? ¿Es el mismo salto? ¿Se está saltando sobre un salto? ¿Cómo puede ser que en medio del primer salto

se defina otro salto distinto? Esa es la razón principal por la que dos cosas iguales no pueden ocurrir a la vez. En cambio, sí que podría suceder que en un mismo intervalo de tiempo estén sucediendo dos cosas al mismo tiempo. Por ejemplo, "saltar" y "flexionar rodillas", pero se definirían como pasos diferentes.

Gráficamente las clases se dividen de dos maneras: Contenedores y clases "hoja". El programa puede verse como una estructura de árbol, en las que de la ventana principal, cuelgan los contenedores de vídeo y gráficos, y de los de gráficos cuelga el visualizador de datos.

Los vídeos tienen la misma estructura. También disponen de un contenedor en el que se guardan todos los vídeos. Esos vídeos, irán sincronizados con los gráficos, mostrando el progreso, para que sea más intuitivo seleccionar los rangos.

5.2. Diseño

No se ha seguido un proceso de diseño clásico, en el cual antes de empezar a programar se realiza un diseño exhaustivo, en el que se modelan las clases, atributos, incluso algunos métodos y llamadas entre métodos.

Se ha seguido un proceso iterativo de diseño. En cada sprint, se iba construyendo sobre lo anterior, y refactorizando y mejorando lo que había previamente. Ha sido un proceso en el que se ha sido muy crítico con lo que ya estaba hecho, en ningún momento se ha dejado algo porque "funciona", siempre se ha intentado mejorar lo que ya había.

En cada iteración se iba consolidando el trabajo previo. Y si se llegaba a una situación en la que había que rediseñar partes grandes se hacía sin miedo.

5.2.1. Librerías usadas

- **AvalonDock:** Es una librería que permite crear ventanas acoplables en WPF, al más puro estilo Eclipse o Visual Studio. Es de código abierto y gratuita, pese a que también dispone de una versión profesional que es de pago. La usada en este proyecto ha sido la versión gratuita o *community*. La versión usada esta licenciada bajo licencia BSD.
- **OxyPlot:** Biblioteca de código abierto para realizar gráficos. Permite visualizar gráficos contínuos, discretos, selección de rangos. Licenciada

bajo los términos de la licencia MIT.

5.2.2. Patrones utilizados

MVVM

MVVM es un patrón de diseño creado por Microsoft. Intenta conseguir las ventajas de Model-View-Controller y además una separación total de la vista del controlador. De esta forma los diseñadores de UI pueden centrar todos sus esfuerzos en crear la interfaz sin preocuparse del código de la aplicación, ya que se asume que la interfaz va a cambiar mucho durante el ciclo de vida de la aplicación. Pero, ¿Si separamos totalmente la vista del modelo, como se comunican? En MVC el encargado de eso es el controlador, pero un detalle importante es que el código referente a la vista debe tener llamadas a funciones en el lenguaje utilizado.

Y en este punto es donde la parte VM [1] toma sentido. El View-Model es una abstracción de la vista, y que es el objetivo de los enlaces de datos. El View-Model expone los datos del modelo para que el creador de interfaces pueda enlazar los elementos de la UI con los datos del modelo (bien utilizando orientación a objetos o exponiendo los datos de la BD) sin escribir una sola línea de código .NET.

Elementos de MVVM

- **Model:** Dentro del patrón MVC corresponde al modelo de dominio utilizando orientación a objetos, o a los datos representados por una BD.
- **View:** La interfaz de usuario con sus botones, cajas de texto etc.
- **View Model:** Es la *Vista del Modelo*, siendo una abstracción que sirve de mediadora y que expone los datos del modelo a la vista, para que esta última pueda utilizarlas mediante enlaces de datos simples que no requieren de código, sino que se crean mediante XAML. Realmente, no sólo expone los datos, si no que además convierte los datos del modelo, en datos de vista, listos para ser visualizados.

Críticas Es curioso que las principales críticas provengan de su creador, Josh Gossman, el cual dice que utilizar a la fuerza MVVM para operaciones

[1] View-Model

simples de UI es una exageración y que si los enlaces de datos si no se gestionan correctamente puede llevar a un consumo excesivo de memoria de la aplicación (Gossman, 04/03/2006)

Uso en MIPS El patrón MVVM no se ha utilizado tal y como fue concebido. Pero sí se ha mantenido la separación entre las tres partes del software. Se han creado los ViewModels tal y como se recomienda en el patrón, pero no se han podido utilizar los enlaces de datos, ya que para poder usarlos hay que establecer un contexto de datos y en este caso es dinámico. Un contexto de datos determina de dónde van a obtenerse los datos para los enlaces. Un contexto de datos puede ser el propio objeto, una lista, una consulta a una base de datos... Y en este caso, el contexto de datos al ser dinámico los enlaces de datos no se pueden establecer fácilmente en el XAML, hay que hacerlo programáticamente. Además se necesitan más cosas que simplemente visualizar unos datos. Pero pese a todo, el software se ha construido en torno a la idea del MVVM aunque no se haya podido aplicar exactamente como se recomienda.

Iterator

Si bien el titulo alude al patrón *Iterator* .NET no dispone como Java de las interfaces *Iterable* ni *Iterator*. En cambio, dispone de las interfaces llamadas *IEnumerator<T>* e *IEnumerable<T>* que ofrecen una funcionalidad equivalente a las de Java.

Se ha decidido utilizar este patrón por los siguientes motivos:

1. **Abstracción:** Al utilizar este patrón se pueden obtener los elementos de un contenedor, que puede ser un *array*, una *LinkedList<>*... cualquier clase que implemente la interfaz *IEnumerable<T>* sin exponer su representación interna, aumentando la seguridad y previniendo que una clase tenga acceso a cosas que no deba.
2. **Facilidad de uso:** Siempre es preferible utilizar cosas que nos provea el *framework*, ya que no es necesario perder el tiempo en banalidades como recorrer una lista, y probablemente lo haremos menos eficientemente que lo actualmente programado. De todas formas, si necesitamos recorrerlo de una manera particular, y que sabemos hacerlo muy eficientemente, no hay problema en no seguir el patrón, ya que no son normas, si no recomendaciones.
3. **Prevención de errores:** Relacionado con lo anterior. Si se pierde el

tiempo añadiendo líneas extra la probabilidad de que se cometa un error aumenta. Y aunque no haya sido probado empíricamente, se tiende a pensar que *¿Como me voy a equivocar en recorrer un array? ¡Pero si está perfecto!*. Y ello llevará a malgastar más aun algo tan preciado como el tiempo. Pero al menos servirá para darse cuenta del error y se verá lo útiles que son los patrones.

Uso en MIPS Este patrón ha sido muy útil en todo aquello que tenga que ver con recorrer datos. Por ejemplo a la hora de sincronizar el rango seleccionado entre los visualizadores de datos. Además en C# se realiza de una manera muy semántica.

```
1   public partial class UC_ChartContainer : UserControl
2   {
3     private SortedSet<UC_DataVisualizer>
        datavisualizers;
4
5     internal void updateSelections(double[]
        selectedRange)
6     {
7       foreach(UC_DataVisualizer datav in
          datavisualizers)
8       {
9         datav.updateRangeSelection(selectedRange);
10      }
11    }
12  }
```

Como se puede ver en el trozo de código, se ha podido iterar a través de todos los elementos de *datavisualizers* sin tener que comprobar si hay un elemento siguiente, el tamaño del elemento sobre el que se quiere iterar... Y además como se ha dicho antes, este código se ha vuelto mucho mas legible, ya que puede leerse de la siguiente manera: "para cada datav en datavisualizers, actualizar el rango seleccionado de datav con el rango obtenido por parámetro". Concretamente, el ejemplo hace referencia al método de sincronización de rangos de las observaciones. Lo que se ha hecho es recorrer todas las propiedades cargadas de una observación para sincronizarlo con el rango dado como parámetro.

Singleton

Patrón de diseño básico que consigue que una clase tenga una única instancia durante toda la ejecución del software, siendo accesible por toda la aplicación.

En general, hay dos maneras de crear una clase Singleton.

Lazy La inicialización *Lazy* o *perezosa* únicamente crea la instancia cuando la necesita, es decir, la primera vez. Para conseguirlo, cada vez que se intenta obtener la instancia se comprueba si ya había sido previamente instanciada. Uno de los problemas de la inicialización perezosa es que en entornos multihilo da problemas si no se obtiene un bloqueo exclusivo de la clase.

Una inicialización perezosa con comprobación doble de bloqueo se realiza así:

```
1   public class GraphicsActions
2   {
3       private static object myLock = new object();
4       private static GraphicActions myGraphicActions
            ;
5
6       public static GraphicActions
            getMyGraphicActions()
7       {
8           if (myGraphicActions == null)
9           {
10              lock (myLock)
11              {
12                  if (myGraphicActions == null)
13                  {
14                      myGraphicActions = new
                            GraphicActions();
15                  }
16              }
17          }
18
19          return myGraphicActions;
20      }
21  }
```

En .NET 4.0 fue incluida la clase *Lazy<T>* que realiza ese tipo de inicialización.

Eager En la inicialización *Eager* o *ansiosa* se crea la nueva instancia siempre. En este método no hay problemas en entornos multihilo, pero tiene un mayor coste computacional si la constructora es muy compleja.

Una inicialización ansiosa se realiza de la siguiente manera:

```
1       public class GraphicsActions
2       {
3           private static GraphicActions myGraphicActions
                = new GraphicActions();
4
5           public static GraphicActions
                getMyGraphicActions()
6           {
7               return myGraphicActions;
8           }
9       }
```

Críticas Pese a haberse utilizado el patrón Singleton, muchos desarrolladores desaconsejan su uso, ya que implican una violación del Principio de Responsabilidad Única del que se hablará más adelante. Y lo viola porque la clase Singleton tendría dos responsabilidades, controlar que solo exista una única instancia y además toda su lógica de negocio (Button y Densmore, 25/05/2004).

Por otro lado, también tiene como inconveniente una de las máximas de la programación: Las variables globales son malas. Pues por si no fuera poco, un Singleton es una clase global (Button y Densmore, 25/05/2004).

Uso en MIPS Pese a las críticas del párrafo anterior, se ha utilizado en aquellos elementos que solo iban a tener una instancia. Concretamente *VideoActions*, *GraphicActions* y *XMLExport*. Al hacerlos accesibles desde todo el código, se ha facilitado mucho la tarea al desarrollador para utilizarlo.

5.2.3. SOLID

SOLID es un acrónimo mnemónico acuñado a comienzos de la década de los 2000 por Robert C. Martin (Martin, s.f.-b) que representa cinco principios básicos de la programación orientada a objetos y del diseño de software.

SOLID merece un apartado en esta memoria ya que se ha intentado cumplir en el mayor grado posible los cinco principios.

Los principios son los siguientes:

Single responsibility principle

Traducido al castellano: Principio de responsabilidad única. El nombre puede llevar a engaño ya que Martin define como *responsabilidad* a una *razón de cambio* (Martin, s.f.-c) y concluye que una clase, debiera tener, una, y solamente una razón para cambiar.

Por ejemplo, imaginemos que disponemos de una clase, o módulo que compila e imprime un informe. Ese informe puede cambiar por dos razones, de estilo, o de contenido.

El principio de responsabilidad única dice que una clase solo debiera cambiar por un motivo, y por tanto tendríamos que separar el módulo en dos: El que compila el informe, y el que lo imprime.

Utilizando este principio volveremos nuestro código más robusto, ya que un cambio en una clase, no debería afectar a la salida, y por tanto los siguientes módulos que se alimenten de esa clase.

Uso en MIPS Si bien el principio de responsabilidad única no es algo que se aplique de manera directa, es decir, no es si sucede A entonces haz B. Pero hay en algunas situaciones en las que se cumple este principio. Por ejemplo, en la clase *UC_ChartContainer* se mantenía un repositorio de Observaciones y además de propiedades. Se ve, que esta clase tiene dos razones de cambio. Una es por las propiedades (añadir, borrar etc) y la otra por las observaciones (añadir, borrar etc). Al incumplirse el principio, la clase ha de separarse en dos. En un repositorio de propiedades, y en un repositorio de observaciones.

En este caso concreto, no se separó en dos clases, sino que la colección de propiedades, pasó a ser responsabilidad de cada *UC_ChartContainer*.

Open/Closed principle

El principio abierto / cerrado (Martin, Enero 1996) implica que una clase está abierta a la extensión pero cerrada a la modificación. Ya que un cambio en el funcionamiento de una clase haría que se tambalease la estabilidad de nuestro software.

Uso en MIPS Ciertamente este principio no se ha utilizado, ya que está más pensado para ser usado en código que ya está en producción, no mientras el software está en desarrollo.

Liskov's substitution principle

El principio de sustitución de Liskov (Martin, Marzo 1996b) [2] sostiene que cualquier subtipo de una clase debe poder sustituir a su supertipo sin que haya un cambio de comportamiento.

Más formalmente: Si **S** es un subtipo de **T** entonces **T** puede sustituirse por **S** y conservar su exactitud, tarea que realiza etc.

Uso en MIPS Este principio se cumple claramente, ya que para incumplirlo habría que sobreescribir algún método de la superclase y que funcionase diferente de como funciona en la superclase.

Tómese como ejemplo las clases *XMLValidation* y *XMLLoader*. Si *XMLValidation* no fuera abstracta, podría ser sustituida por *XMLLoader* sin alterar su funcionalidad base, ya que *XMLLoader* no sobreescribe el método de la clase base. Lo mismo sucede con los ViewModels. Ambas clases derivadas podrían sustituir a la clase abstracta sin cambiar su funcionalidad. Es decir, aunque las clases derivadas hayan sustituido a la clase padre, las clases derivadas siguen teniendo la funcionalidad de la clase base intacta.

Interface segregation principle

El principio de segregación de interfaces dice que es mejor tener muchas interfaces específicas, que una gran interfaz que contenga todo (Martin, s.f.-a).

De esta manera evitamos que haya partes del software que dependan de métodos que no influyen en esa clase.

[2] `http://pmg.csail.mit.edu/~liskov/`

Uso en MIPS Al solo tener una interfaz propia, y que además tiene un único método, se cumple siempre, ya que cada vez que una clase implemente esa interfaz es porque se requiere ese método, si no, no tendría sentido implementarla.

Dependency inversion principle

El principio de Inversión de dependencia (Martin, Marzo 1996a) dice que:

1. Los módulos de alto nivel no deben depender en módulos de bajo nivel. Ambos deben depender de abstracciones
2. Las abstracciones no deben depender en los detalles. Los detalles deben depender de las abstracciones.

Uso en MIPS Este principio es el único que no se cumple. Ya que los módulos de alto nivel, como por ejemplo *UC_ChartContainer* no depende de abstracciones, sino de tipos concretos (*UC_DataVisualizer*). Para arreglarlo, habría que definir una serie de interfaces que describan los métodos necesarios que debe tener un visualizador de datos. Y de esa manera el contenedor depende de esas abstracciones (interfaces). Esto mismo habría que aplicárselo a todos los elementos del proyecto.

Excepto en casos muy concretos, los métodos, deberían tomar como parámetros, o bien interfaces, o bien objetos que implementen esas interfaces.

5.2.4. Clean Code

Otra de las cosas en las que incide mucho Martin es que evitemos realizar comentarios en el código, ya que el hecho de realizar un cambio en el código no garantiza que vayamos a cambiar el comentario, y puede llevar a errores futuros o a inconsistencias en la documentación. Además, todo el mundo sabe que programar es un arte (Geek y Poke, 2013), y es por ello que el código de MIPS tiene los comentarios justos y necesarios para hacer entender los métodos que no se entienden o que tienen una lógica ligeramente mas compleja de seguir.

5.2.5. Diagrama de clases

El diagrama de clases 5.1 tiene todos los métodos, propiedades y atributos ocultos para que entrase en el propio documento. En el anexo A puede en-

contrarse la versión completa. Las clases que no son descritas más adelante es porque o bien son clases propias de .NET o porque son clases generadas por Visual Studio.

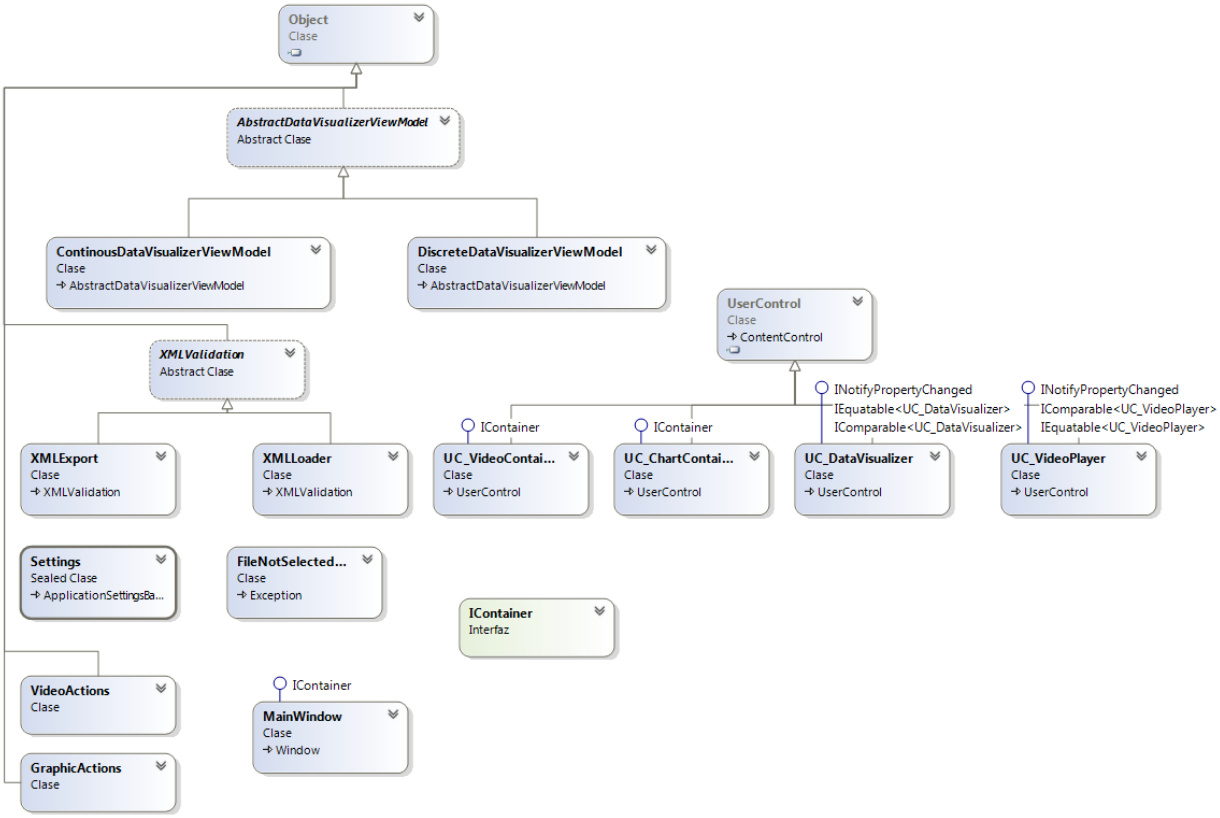


Figura 5.1: Diagrama de clases compacto

AbstractDataVisualizerViewModel

Clase abstracta que describe e implementa todos los métodos y propiedades que han de tener todos los ViewModels. Es decir, define todo aquello que podrá ser cargado por un *UC_DataVisualizer*. Esta clase expone el método *potected abstract PlotModel createModel(List<DataPoint>points)* que todas sus subclases han de implementar.

ContinousDataVisualizerViewModel

Subclase de *AbstractDataVisualizerViewModel* que se especializa en tratar los datos contínuos.

DiscreteDataVisualizerVIewModel

Subclase de *AbstractDataVisualizerViewModel* que se especializa en tratar los datos discretos. Esta clase requiere de datos adicionales, concretamente los elementos del eje de ordenadas.

XMLValidation

Clase abstracta que sirve como base a todas aquellas clases que requieran de validar un XML contra un XSD.

XMLExport

Subclase de *XMLValidation* que se encarga de gestionar todo lo relacionado con exportar un XML. Creación y validación de intervalos y guardar a disco el paso o situación principalmente.

XMLLoader

Subclase de *XMLValidation* que tiene como responsabilidad cargar los datos del XML, tanto obtener los datos para mostrar en el árbol como de crear los ViewModels que después serán utilizados por los *UC_DataVisualizer*.

VideoActions

Clase Singleton que proporciona los métodos necesarios para gestionar los vídeos cargados.

GraphicActions

Clase Singleton que gestiona los *UC_ChartContainer*. Añadir, borrar, sincronizar rangos etc.

UC_VideoContainer

Control de usuario derivado de *UserControl* y que implementa la interfaz *IContainer*. Dispone de la interfaz y los métodos necesarios para gestionar los *UC_VideoPlayer* (añadir, borrar etc) así como unos controles globales para controlar su reproducción.

UC_ChartContainer

Control de usuario derivado de *UserControl* y que implementa la interfaz *IContainer*. Esta clase contiene la interfaz gráfica y los métodos requeridos para gestionar los *UC_DataVisualizer*. Representa una observación que dentro tiene sus propiedades. También es importante añadir que es esta clase la que se suscribe al evento que notifica que el rango seleccionado ha cambiado, para poder inicial el proceso de sincronización.

UC_DataVisualizer

Control de usuario que se encarga de mostrar los datos de las propiedades y una de las funcionalidades principales consiste en notificar a su padre (un *UC_ChartContainer*) de que el rango seleccionado ha cambiado.

Esta clase extiende a *UserControl* e implementa las interfaces *INotifyPropertyChanged*, *IEquatable<T>* e *IComparable<T>*. Es necesario implementar esas interfaces para que el árbol dónde se almacenan los objetos funcione correctamente, ya que requiere ordenar las instancias de alguna manera.

UC_VideoPlayer

Control de usuario que se encarga de reproducir vídeos.

Al igual que *UC_DataVisualizer* extiende *UserControl* e implementa las interfaces *INotifyPropertyChanged*, *IEquatable<T>* e *IComparable<T>*. Los motivos de su implementación son los mismos que en *UC_DataVisualizer*.

IContainer

Esta interfaz define aquellos métodos que deben ser implementados por las clases que vayan a ser contenedores de algún tipo de dato.

FileNotSelectedException

Excepción personalizada pensada para ser lanzada en los diálogos de apertura de ficheros.

MainWindow

Punto de entrada al programa. Al ser un contenedor de contenedores, concretamente de *UC_ChartContainer* y *UC_VideoContainer*, también implementa la interfaz *IContainer*.

Esta clase, principalmente se encarga de realizar las llamadas a cargar los datos y añadir las observaciones y el contenedor de vídeos.

Capítulo 6

Desarrollo

6.1. Qué se ha hecho

Se ha desarrollado una aplicación que dados unos datos organizados de cierta manera permite la visualización de los mismos y la selección de unos rangos para guardar y especificar que en ese rango de tiempo está sucediendo una acción determinada, ya sea un Paso o una Situación.

Los datos de entrada representan Observaciones y Propiedades. Cada observación tiene una serie de propiedades que son las que se visualizan. Estos datos, cuando fueron capturados, puede que tengan uno o varios vídeos asociados. La visualización de los vídeos es completamente optativa, y en caso de visualizarlos, van sincronizados con los gráficos a partir del instante que el usuario diga.

Es decir, en cada gráfico habrá una línea de progreso para que sea fácil determinar en que ciclo de simulación estamos [1].

Una vez hemos determinado cual es el problema a resolver, se hace necesario tomar ciertas decisiones. Desde un principio se sabía que había que programar en un entorno Windows, utilizando Visual Studio 2013. La razón de esta decisión fue que el proyecto actual está desarrollado utilizando Windows Forms, y el director del proyecto comentó que existía una intención de portarlo a WPF.

Pese a conocer esa intención, hubo que valorar qué merecía más la pena, si utilizar WPF, una tecnología moderna en la que Microsoft está poniendo todo

[1] Un ciclo de simulación es la unidad de tiempo elegida para la captura de datos, no necesariamente es un segundo.

su empeño, o bien Windows Forms, que es un viejo conocido del desarrollo .NET con un camino muy largo de desarrollo.

Para decidir de la manera más objetiva posible, se confeccionó la tabla 6.1 en la que se sitúan las características de cada tecnología. La tabla es una elaboración propia confeccionada a partir de una comparativa online (Chapell, Septiembre 2006a).

	Windows Forms	WPF
Formularios y controles	Si	Si
Documentos en pantalla	Si	Si
Documentos de formato fijo (XPS, PDF)	No	Si
Imágenes	No	Si
Vídeo y audio	No	Si
Gráficos 2D	No	Si
Gráficos 3D	No	Si
Interfaz compatible con altas resoluciones	No, basada en BMP	Si, basada en vectores
Creación de la interfaz	Arrastrando y soltando los elementos	Interfaz declarativa tipo XML
Multilenguaje	Mediante archivos de recursos, fácil y bien documentado	Mediante DLLs satélite, poco documentado y las herramientas aun no están listas para un entorno de producción.

Cuadro 6.1: Comparativa Windows Forms y WPF

Como puede observarse, Windows Forms se ha quedado obsoleto para un desarrollo de aplicaciones moderno. Pese a todo, WPF, tiene algunos puntos débiles. Entre ellos, el soporte multilenguaje, y que es un proyecto mucho menos maduro, ya que fue lanzado en 2006 junto con .NET Framework 3 (Chapell, Septiembre 2006b). Windows Forms, por su parte se presentó junto con la primera version de .NET Framework, en 2002.

Esta elección condiciona el resto de decisiones que fueron tomadas, ya que las bibliotecas gráficas de Windows Forms no son compatibles con WPF. Existe una capa de compatibilidad en la que es posible embeber dentro de un host WPF un control Windows Forms, pero no es recomendable, ya que

no deja de ser una capa de compatibilidad y no es posible sacarle toda la potencia a Windows Presentation Foundation.

Una de las primeras decisiones que se tomaron, y además sin demasiado debate, fue que tipo de interfaz se deseaba. Se eligió una interfaz tipo IDE, con ventanas acoplables, tal y como se ha detallado en el capítulo de Antecedentes.

En este caso en concreto no hubo que buscar mucho, ya que no hay mucho donde elegir. La biblioteca seleccionada fue AvalonDock.

AvalonDock es una biblioteca escrita íntegramente para WPF, con soporte para MVVM. Permite todo lo que se espera de un proyecto como ese: Ventanas acoplables a los lados, mover pestañas etc, fue curioso que durante desarrollo se descubrió un bug en el software.

Dicho bug, lanzaba una excepción cuando un elemento "acoplable", que tenía un Menú superior, al ponerlo en una pestaña, al cambiar de pestaña fallaba. El bug ya estaba reportado, pero aun así se le proporcionó al desarrollador más información sobre el bug [2], y a día de hoy sigue sin estar resuelto.

Antes de explicar como es la interfaz que se ha diseñado, primero hay que saber como trabaja internamente AvalonDock y como se usa para poder crear esas pestañas y ventanas acoplables.

Explicado de manera muy sencilla, AvalonDock se compone de cuatro partes principales: el *DockingManager*, el *LayoutPanel*, el *LayoutAnchorablePane*, y el *LayoutAnchorable*. También tiene otros elementos pero no han sido usados en este proyecto.

DockingManager: Es el núcleo de AvalonDock, donde todos los elementos WPF son añadidos, necesario siempre para poder trabajar con AvalonDock.

LayoutPanel: Este elemento organiza todos los hijos, poniendo un separador entre ellos, establecido usando la propiedad *Orientation*.

LayoutAnchorablePane: Estos elementos son la clave. Ya que es el panel que permite que los objetos *LayoutAnchorable* pueden ser arrastrados, puestos en pestañas, lado a lado etc.

[2] `https://avalondock.codeplex.com/discussions/429063`

LayoutAnchorable: Estos elementos siempre deben ir dentro de un *Pane*, bien en un *LayoutAnchorablePane* bien en un *LayoutDocumentPane* que no ha sido utilizado. Estos elementos son los que pueden reorganizarse como uno quiera, anclarse a un lado, poner que se oculten etc. Y dentro pueden tener aquello que se desee.

Por tanto, para tener una instancia funcional de AvalonDock, tendremos que tener un *DockingManager* y un *LayoutRoot* dentro de ese dockingmanager. Y dentro de ese *LayoutRoot* un *LayoutAnchorablePane*.

Una vez tenemos eso, añadir a un *LayoutAnchorable* un elemento es muy sencillo, no hay mas que añadirlo dentro de la propiedad Content. Y para añadir ese *LayoutAnchorable* al *LayoutAnchorablePane* hay que añadirlo a la colección de hijos. En la Figura 6.1 se puede ver un ejemplo de como se añade un *UC_DataVisualizer* a AvalonDock.

```
1       public void addToAnchorablePane(UserControl
           objectToAdd, string Title)
2       {
3           UC_DataVisualizer datav = (UC_DataVisualizer)
               objectToAdd;
4           LayoutAnchorable doc = new LayoutAnchorable();
5           doc.Title = Title;
6           doc.Content = datav;
7           mainPanelChartContainer.Children.Add(doc);
8
9       }
```

Figura 6.1: Adición de elemento a AvalonDock

Una vez conocemos el funcionamiento de AvalonDock, podemos pasar a hablar de la interfaz diseñada. La estructura que se ha diseñado es un "workbench", con un panel lateral en el que se pueden visualizar las observaciones y propiedades disponibles después de haber cargado un fichero XML. Cuando se hace doble click sobre un elemento de ese árbol de propiedades y observaciones, el software sabe si se ha pinchado en una propiedad o en una observación. Si se quiere saber que sucede cuando se pincha en cada uno de ellos, todo esto está mejor explicado en anexo B de Captura de requisitos, en los casos de uso extendidos.

Cuando se cargan los datos que se desean, es decir, que se visualizan, como puede verse en la Figura 6.2 se muestran organizados por observaciones. Y dentro de cada observación las propiedades que han sido cargadas. Las

ventanas pueden organizarse como se quiera, ponerlas lado a lado, reordenar las pestañas, colapsarlas a un lateral, dejarlas como ventanas flotantes etc. La única limitación intencionada es que una propiedad no puede acoplarse a una observación a la que no pertenece, para evitar confusiones.

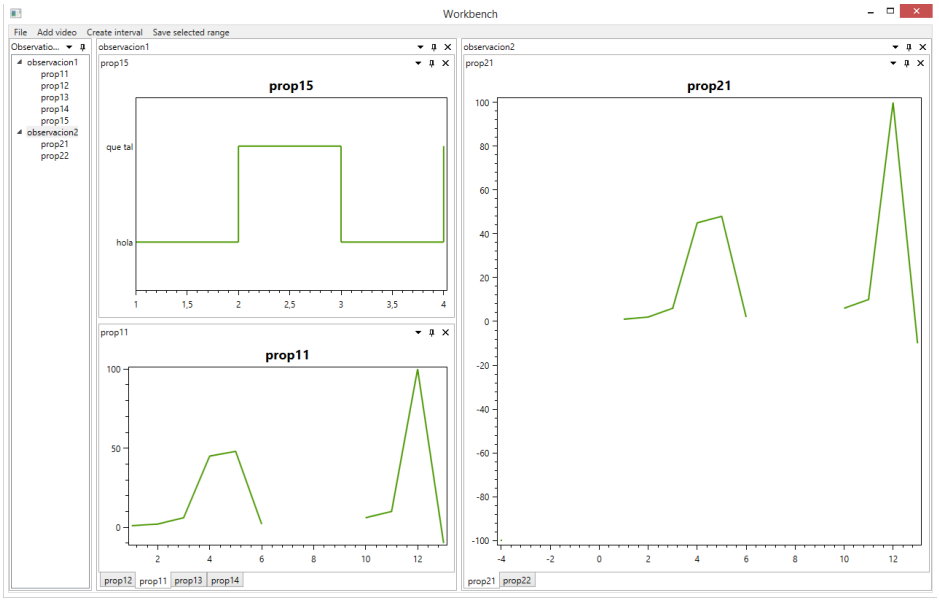


Figura 6.2: Ejemplo de visualización de propiedades

Después de escoger el tipo de organización de ventanas, era necesario elegir como iban a mostrarse los gráficos en pantalla. Para el sistema de gráficos no hubo problemas a la hora de encontrar distintas librerías, pero sí que fue un problema encontrar una que cumpliera las siguientes características.

- Permitir mostrar gráficos contínuos y discretos, y estos últimos tanto numéricos como por categorías.
- Permitir la selección de rangos con el ratón.
- Permitir mostrar el progreso del vídeo asociado. Bien sin programar la función o al menos que fuese fácilmente programable.

Se descartaron todas las bibliotecas de pago, como pueden ser Telerik, SciChart, Visiblox, etc. por ser demasiado caras y porque se decidió utilizar en la medida de lo posible tecnologías libres y gratuitas. Entre las bibliotecas gratuitas, destacaron tres por encima de todas. Las capacidades nativas de WPF, WPF toolkit, desarrollada por Microsoft, y OxyPlot.

Los gráficos 2D integrados en WPF son muy simples de utilizar, con una estética muy minimalista, de colores planos. Lamentablemente, la selección

de rangos, y mostrar leyendas en los ejes X e Y no eran tareas triviales, por lo que se descartó al poco de iniciar el desarrollo.

WPF toolkit fue la siguiente librería en pasar la prueba. Se integra a la perfección en Visual Studio y muestra unos gráficos muy profesionales de una forma muy simple, pero desafortunadamente, no existe una manera sencilla de seleccionar un rango.

OxyPlot, por su parte, es un proyecto comunitario en constante desarrollo. Con decenas de ejemplos online en su web oficial fue muy sencillo crear un prototipo funcional con todos los gráficos requeridos, con las leyendas en los ejes y los títulos. Conseguir la selección de un rango conllevó algo más de tiempo de desarrollo.

Para crear un gráfico es tan fácil como crear en el XAML un elemento *<oxy:PlotView />* que va a ser el elemento que va a contener el gráfico. Después, únicamente hay que añadir un *PlotModel*, que es la clase que contiene tanto los puntos, el titulo del gráfico, los ejes etc. Lo mejor en estos casos es un ejemplo. En la Figura 6.3, se ve el método de creación de un *PlotModel* para un visualizador de datos discreto.

```
1       protected override PlotModel createModel(List<
           DataPoint> points)
2       {
3           var plotModel = new PlotModel();
4           var functionSeries = new StairStepSeries();
5           var categoryAxis = new CategoryAxis();
6
7           categoryAxis.Position = AxisPosition.Left;
8           categoryAxis.AxislineStyle = LineStyle.Solid;
9           categoryAxis.MinorStep = 1;
10          categoryAxis.TickStyle = TickStyle.None;
11          categoryAxis.Labels.AddRange(Labels);
12          plotModel.Axes.Add(categoryAxis);
13          functionSeries.Points.AddRange(points);
14          plotModel.Series.Add(functionSeries);
15          plotModel.Title = Title;
16          Points = functionSeries.Points;
17          return plotModel;
18      }
```

Figura 6.3: Creación PlotModel

La selección de un rango, se hace añadiendo al *PlotModel* un elemento *RectangleAnnotation*. Para actualizar el valor de inicio y fin de ese rango, hay que suscribirse a tres eventos, ya que una selección de rango es la combinación de tres eventos: hacer click, arrastrar, y dejar de hacer click. Cuando se pincha, se establece el lugar de inicio de selección, y mientras se va moviendo el ratón, si el botón izquierdo del ratón está presionado, entonces se va actualizando el rango seleccionado. Y cuando se deja de pinchar, termina el proceso.

En los primeros prototipos, en cada gráfico podía seleccionarse un rango diferente, pero era algo inconsistente, ya que cuando llegase la hora de guardar los rangos, ¿Cual se usaría? Por lo que fue necesario continuar un poco con la investigación para ver como podría un hijo notificar a su padre de que ha cambiado, para que ese padre notifique a todos sus hijos para sincronizarse. Recuérdese, que los elementos están dispuestos en forma de árbol, de la pantalla principal cuelgan las observaciones y el contenedor de vídeos, y de cada observación sus hijos son los gráficos que representan las propiedades, y del contenedor de vídeos, sus hijos son los propios vídeos. Para mayor detalle ver el capítulo de Análisis y diseño.

La primera aproximación, y posiblemente la peor de todas, era realizar un *polling*, es decir, cada, por ejemplo, 60 ms, preguntar a cada elemento si ha cambiado, y si alguno lo ha hecho, utilizar ese valor para poner ese rango en los demás gráficos. Este enfoque, tiene diversos problemas, pese a que los gráficos están guardados en árboles rojo-negro [3] [4], que son unos árboles binarios autobalanceables, no tiene sentido recorrer el árbol constantemente, y si se encuentra un elemento, habría que recorrer cada observación cargada, y para cada observación todas las propiedades cargadas actualmente. Eso, conllevaría un coste de O(2*n*m) siendo "n" el número de observaciones cargadas y "m" el número de propiedades.

La otra solución, es implementar la interfaz *INotifyPropertyChanged*. Utilizando esta interfaz, se expone un nuevo evento, llamado *PropertyChanged* al que otros objetos se pueden suscribir para estar a la escucha.

De esta manera, cada vez que se añada un nuevo gráfico, el contenedor de gráficos se suscribe a ese evento para todos sus hijos. De esta manera, cuando el rango seleccionado cambie, será ese hijo el que notifique al padre, que iniciará el proceso de sincronizado, tanto con sus hijos como con sus hermanos (el resto de observaciones cargadas).

[3] `https://www.cs.auckland.ac.nz/~jmor159/PLDS210/red_black.html`

[4] `https://www.cs.princeton.edu/courses/archive/fall08/cos226/lectures/10BalancedTrees-2x2.pdf`

Para la visualización de vídeos no hubo mucho problema, y se utilizaron los propios componentes de WPF, que cumplían todas las necesidades.

Por su parte, cargar los datos no fue demasiado problema una vez se decidió cual iba a ser el formato de los ficheros XML.

Se barajaron dos posibilidades. La primera, que por cada instante de simulación se guardaran las observaciones que están teniendo lugar, y los valores de las propiedades que tienen en ese instante, tal y como se puede ver en la Figura 6.4.

Este formato, a la vista parece bueno, ya que se pordría construir los gráficos instante a instante, todos a la vez. Pero, ¿Y si se quiere obtener el valor de una propiedad en concreto para mostrar todo el gráfico de golpe? No se obtiene de manera directa, ya que habría que ciclar a través de todos los nodos instante, y a través de todos los nodos observación, buscando la propiedad deseada.

También hay que tener en cuenta, que en distintas observaciones puede haber propiedades que tengan el mismo nombre, aumentando la complejidad de la consulta LINQ (más adelante se hablará de este tema).

```
1  <data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2      EspObservation="ObservationModelExample.xml">
3    <instante numInstante = "1">
4      <observacion name="observacion1">
5        <propiedad name="prop11" value="1" />
6        <propiedad name="prop12" value="2" />
7        <propiedad name="prop13" value="3" />
8        <propiedad name="prop14" value="4" />
9      </observacion>
10   </instante>
11   <instante numInstante = "2">
12     <observacion name="observacion1">
13       <propiedad name="prop11" value="2" />
14       <propiedad name="prop12" value="3" />
15       <propiedad name="prop13" value="4" />
16       <propiedad name="prop14" value="5" />
17     </observacion>
18   </instante>
19 </data>
```

Figura 6.4: Estructura XML 1

La segunda opción barajada, toma un enfoque distinto., tal y como se puede ver en la Figura 6.5. Los datos se agrupan por observación, con las observaciones teniendo sus propiedades, y estas últimas sus instantes. De esta manera, de un simple vistazo podemos determinar todos los valores que va a tomar una propiedad de una observación en concreto, sin ningún tipo de consulta compleja. En el nodo *<data>* se ha añadido un nuevo atributo *instantLength* que determina la longitud de un instante.

Por ejemplo, en la propiedad *prop14* se ve que los instantes no son consecutivos, por lo que el sistema sabe que el gráfico no debe ser contínuo en ese punto, es decir, desde el punto (1, 4) no debe unirse al (3, 5). Esto es útil para saber si una propiedad esta sucediendo o no.

```
1  <data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2      EspObservation="ObservationModelExample.xml"
3      instantLength="1">
4    <observation name="observacion1">
5      <property type="1" name="prop11">
6        <instant ins="1" value="1"/>
7        <instant ins="2" value="2"/>
8      </property>
9      <property type="1" name="prop12">
10       <instant ins="1" value="2"/>
11       <instant ins="2" value="3"/>
12     </property>
13     <property type="1" name="prop13">
14       <instant ins="1" value="3"/>
15       <instant ins="2" value="4"/>
16     </property>
17     <property type="1" name="prop14">
18       <instant ins="1" value="4"/>
19       <instant ins="3" value="5"/>
20     </property>
21   </observation>
22 </data>
```

Figura 6.5: Estructura XML 2

Otro de los problemas fundamentales que se tuvieron, fue como guardar los datos. Debido a que de momento MIPS va a ser completamente independiente, había que buscar una manera de compartir los datos que implicara el

menor número de cambios posibles en ULISES.

Se barajaron varias posibilidades, como por ejemplo, las bases de datos relaciones, pero este sistema requeriría de una estructura compleja para guardar unos datos bastante sencillos. Ya que del modelo de dominio, se infiere que mínimo, iba a haber 4 tablas para poder guardar los datos correctamente. Además, los datos no iban a ser fácilmente comprensibles por humanos, ya que serían un amasijo de números guardados en tablas que serían difíciles de relacionar entre ellas de manera sencilla.

Es por ello que se valoró la posibilidad de utilizar las bases de datos no relacionales, que están tan de moda últimamente. Concretamente, el producto sometido a pruebas fue MongoDB. Es una base de datos NO-SQL en la que los datos se guardan en formato de "documentos", no tablas. Internamente, se guardan en formato BSON, que no deja de ser un JSON binario [5]. Ofrece un gran rendimiento, además de unos datos mejor organizados para esta tarea, con elementos que tienen arrays, y cada elemento atributos etc.

Finalmente se optó por descartarlo también, ya que añadía mayor complejidad, "oscuridad" y requisitos a la implementación. Ya que habría que tener instalada una base de datos MongoDB, así como los conectores para C#.

Así que se decidió ir por el camino de en medio y elegir un estándar que no requiriera de software adicional, que fuese soportado de manera nativa por el lenguaje de programación y además fácilmente comprensible por humanos. El elegido fue XML más la biblioteca LINQ To XML incluida por defecto en .NET Framework.

XML es un estándar recomendado por la W3C (Bray, Paoli, Sperberg-McQueen, Maler, y Yergeau, Noviembre 2008), y la versión actual es la 1.0 quinta edición. Por si mismo, XML no aporta ningún tipo de ventaja adicional respecto a las otras opciones, excepto el que no se requiere de software adicional para compartir los datos de una aplicación a otra, y que se pueden crear esquemas XSD para validar esos datos de manera muy sencilla. Lo que realmente marca la diferencia es LINQ To XML.

LINQ To XML es una extensión de la biblioteca LINQ (Language Integrated Query), que permite hacer consultas similares a SQL a estructuras de datos .NET.

En el ejemplo de la Figura 6.6, extraído de mi propio código, se obtienen las propiedades de una determinada observación. Concretamente, este

[5]Especificación JSON: `http://json.org/`

código se ha utilizado a la hora de cargar el árbol lateral de observaciones y propiedades.

```
1       internal List<string> getPropertiesOf(string
           observation)
2           {
3               List<string> listProperties = null;
4
5               IEnumerable<XElement> properties =
6                   from prop in xml.Descendants("property
                       ")
7                   where (string)prop.Parent.Attribute("
                       name") == observation
8                   select prop;
9
10              listProperties = addElements(properties);
11              return listProperties;
12          }
```

Figura 6.6: Consulta LINQ

Otras de las cosas que se implementaron a última hora fue la posibilidad de guardar múltiples intervalos en un mismo Paso o Situación, ya que en palabras del director del TFG: "No todo lo que está relacionado sucede únicamente durante un intervalo de tiempo". Para hacer frente a esta nueva funcionalidad hubo dos enfoques principales. El primero de ellos, consistía en que cada vez que se le diera a guardar intervalo, el software preguntara si se querían añadir intervalos mas tarde. Cuando se pinchara que no se mostraría el diálogo de guardar y se guardaría a disco.

La otra opción, era separar las dos acciones, por un lado, crear los intervalos, y por otro lado guardar a disco. Finalmente se utilizó la segunda manera porque se cree de que es más intuitivo tener dos acciones separadas que se sabe bien que realiza cada una, con un flujo de trabajo claro, a tener una única acción, que dependiendo de la acción del usuario haga una cosa u otra.

Una vez elegido el método de guardado de datos y teniendo en mente que deben poder guardarse múltiples intervalos en un mismo paso o situación, se llego a la tesitura de elegir una estructura de guardado para el XML.

Como no se quería añadir complejidad, se utilizó la misma estructura que para la de cargado, pero añadiendo un nodo *<interval>*. Lo mejor, es un ejemplo, tal y como puede verse en la figura 6.7.

```
1  <step name="guardado.xml">
2      <interval start="1" end="3">
3          <observation name="observacion2">
4              <property type="1" name="prop21">
5                  <instant ins="1" value="1"/>
6                  <instant ins="2" value="2"/>
7                  <instant ins="3" value="6"/>
8              </property>
9              <property type="0" name="prop22">
10                 <instant ins="1" value="hola"/>
11                 <instant ins="2" value="que tal"/>
12                 <instant ins="3" value="hola"/>
13             </property>
14         </observation>
15     </interval>
16   ...
17 </data>
```

Figura 6.7: Estructura de guardado

Para poder determinar si los ficheros XML son válidos se han creado dos esquemas XSD. La ubicación de los ficheros XSD se especifican en el fichero de configuración del programa. Por defecto la ubicación de los esquemas es en la carpeta raíz del software, dentro de una carpeta "schemas".

Una cosa interesante y que además no se pedía desde el principio, era si tenía sentido implementar procesamiento paralelo o procesos en segundo plano. El procesamiento paralelo finalmente no tiene sentido implementarlo ya que no hay tareas que puedan separarse en subtareas para que cada "trabajador" pueda procesar por su cuenta. En cambio, al no conocerse el tamaño máximo de entrada, se ha decidido implementar un *BackgroundWorker* a la hora de crear un intervalo, y a la hora de guardar en disco el paso o la situación.

Se ha tomado esta decisión porque si el guardar los datos se hace en el hilo principal de ejecución, dependiendo de la cantidad de datos, podría bloquear la interfaz y de esta manera el software es mucho más escalable. Bien es verdad que un procesador moderno es capaz de procesar unos 50 millones de operaciones por segundo, y no se cree que la entrada, ni la salida vaya a ser tan grande.

A la hora de cargar datos, no se ha implementando el *BackgroundWorker* porque en la aplicación no se puede realizar ninguna acción hasta que los

datos están listos. Pese a todo, al cargar el XML no se cargan todos los datos, sino que se obtienen únicamente los nombres de las propiedades y las observaciones. Los datos en si se cargan bajo demanda, para optimizar la memoria utilizada del propio software haciéndolo mas ágil al mismo tiempo, ya que de esta manera el software no tiene que preocuparse de que datos están cargados y cuales no, ya que todo aquello que está cargado debe procesarse.

En cuanto a las cuestiones de diseño, tal y como se ha contado previamente en el capítulo de Análisis y diseño se ha realizado de manera iterativa, mejorando el diseño sprint tras sprint. Se tomó esta decisión siguiendo los principios de Scrum, concretamente el principio empírico (ScrumStudy, 2013), es decir, que se basa en la experimentación y no en la planificación. Por tanto se acepta que un problema no puede ser completamente entendido o definido desde un primer momento, ya que en fases más avanzadas del proyecto, cuando se tenga más experiencia, se verá que muchas de las cosas diseñadas previamente están mal o se pueden mejorar en gran medida.

Por ejemplo, al principio, la clase *GraphicsActions* contenía una lista con todos los *UC_DataVisualizer*. Este acercamiento se eligió al principio ya que era el más simple para conseguir el objetivo del mes: mostrar datos.

Posteriormente, se hizo necesario guardar también los *UC_ChartContainer*, ya que las propiedades estaban agrupadas por observación. Por lo tanto *GraphicsActions* pasó a tener un *HashMap* teniendo como clave el nombre de la observación y como contenido una lista de *UC_DataVisualizer*. Pero ese diseño entra en conflicto con uno de los principios SOLID que ya fueron explicados. Por lo que *GraphicsActions* ahora es un repositorio de *UC_ChartContainer*, y cada *UC_ChartContainer* contiene una lista de los *UC_DataVisualizer* que pertenecen a esa observación y un *HashSet* con los nombres de las propiedades, para poder obtener en tiempo constante si ya estaban cargados, ya que solo se pueden cargar una única vez.

En la última iteración, se decidió sustituir el *HashSet* y la lista por un árbol rojo-negro. Ya que permite inserción, búsqueda y eliminación en tiempo O(ln n) frente al O(n) de las listas. Además un árbol rojo negro, sólo permite que existe una vez el mismo objeto en el árbol.

6.2. Gestión del código fuente

Hasta ahora no se ha hablado de algo que bien no es parte del desarrollo, pero que igualmente es importante.

Hoy en día, todo proyecto debe tener algún tipo de gestión del código fuente, para evitar pérdida de datos, poder volver a versiones anteriores y tener una mejor noción de que se ha cambiado si es un proyecto colaborativo.

En este caso, al ser un proyecto realizado por una única persona, parecía razonable utilizar algún otro tipo de sistema. Ya que Git, CVS o Subversion son piezas de software muy potentes y usadas a gran escala, pero tienen una curva de aprendizaje bastante pronunciada.

Por tanto se valoró la posibilidad de utilizar Dropbox como sistema de gestión y sincronizado del código fuente. Pero los problemas no tardaron en aflorar con cuestiones del tipo:

¿Qué pasaría si se quiere volver a una versión de hace 20 días? ¿Cómo se crea un trabajo derivado a partir de una versión concreta? Por lo que finalmente se decidió usar Git, ya que incluso con la curva de aprendizaje, es una herramienta muy potente y útil, en la que se puede ver fácilmente qué se hizo cierto día, o volver sin ningún problema a lo que se hizo ayer, o a lo que se hizo el primer día de proyecto.

Crear una nueva rama de desarrollo a partir de cualquier versión es muy útil y práctico, por ejemplo, para realizar pruebas experimentales sin romper el trabajo estable que se ha realizado hasta el momento, o para arreglar algún bug de una versión que ya está en producción. También ha sido fruto de diversos problemas, pero nada que el manual oficial o StackOverflow no pudieran resolver en unos minutos.

6.3. Creación de la documentación

Aunque esta sección pueda parecer una elección obvia, trajo bastantes quebraderos de cabeza. Por un lado se quería algo que no requiriera de un estudio previo de 3 meses, que fuese lo suficientemente potente, que su rendimiento no decreciera según aumentaban las páginas, y desde luego, que fuese multiplataforma.

Cómo el proyecto lo estaba realizando en Windows y para Windows, Microsoft Office parecía la elección obvia. Un producto potente muy bien integrado en Windows, que permite crear documentos muy vistosos de manera muy sencilla. Pese a sólo tener versiones oficiales de Windows y OS X en GNU/Linux funciona razonablemente bien utilizando WINE, una capa de compatibilidad de Windows para sistemas GNU/Linux. Pero cuanto más largo es un documento, el programa cada vez reacciona más despacio, siendo complicado trabajar con documentos largos o con un diseño complejo.

También se valoró la posibilidad de usar LibreOffice u OpenOffice, por ser multiplataforma y de código abierto, pero adolecen de las mismas limitaciones que Microsoft Office. En ambas suites la colocación de imágenes y tablas puede ser como poco, intrincado, pudiendo destrozar por completo el diseño del documento por querer mover medio centímetro una tabla a la derecha.

La última opción valorada, y en la que este documento está escrito ha sido LaTeX. Software de composición de textos creado por Donald E. Knuth [6] muy popular entre los científicos. Ofrece una manera similar de crear documentos a HTML aunque debería decirse al revés, ya que LaTeXes bastante más antiguo.

Para sacarle todo el partido se requiere un buen editor, con resaltado de sintaxis y autocompletado, la opción elegida ha sido TexStudio, un software de código abierto que ofrece todo tipo de facilidades, multiplataforma.

Allá donde el resto de software de edición de documentos fracasa, LaTeX resalta. Los ficheros, al ser de texto plano, pueden integrarse en el software de control de versiones. El escritor puede abstraerse completamente del formato y centrarse en escribir.

También es necesario decir, que pese a que la documentación ha sido escrita principalmente en LaTeX, algunas figuras y gráficos han sido creados con otras herramientas, entre ellas LibreOffice y Microsoft Word 2010.

[6] `http://www-cs-faculty.stanford.edu/~uno/`

Capítulo 7

Verificación y evaluación

En este capítulo se detallarán las pruebas que se han llevado a cabo para verificar que el software cumple todos los requisitos correctamente.

Las pruebas representan los comportamientos que debe tener la aplicación, es decir, lo que se prueban son los casos de uso y subcasos de uso, no si cada método realiza de forma correcta su tarea.

7.1. Cargar XML

7.1.1. Preparación de las pruebas

Para poder probar este comportamiento no se requiere de ninguna preparación especial.

7.1.2. Cargar XML válido nada más abrir la aplicación

Se prueba el comportamiento que debe tener la aplicación al cargar un XML para empezar a trabajar, lo que se conoce como un arranque en frío.

Resultado esperado Todos los datos cargados en la aplicación eliminados, tanto intervalos creados, como visualizadores de datos, contenedores... todo debe haberse borrado de memoria volviendo a un estado inicial de la aplicación.

Resultado obtenido Todos los datos se han borrado excepto los elementos del árbol de observaciones y propiedades.

Acciones realizadas Modificado el método de cargar XML para limpiar también los elementos del árbol.

7.1.3. Cargar XML válido cuando ya haya un XML cargado

Esta prueba contempla dos situaciones, ya que este comportamiento se da cuando hay hay un XML cargado pero no se ha hecho nada, como cuando hay un XML cargado justo después de guardar un paso o situación a disco.

Resultado esperado Todos los datos cargados en la aplicación eliminados, tanto intervalos creados, como visualizadores de datos, contenedores... todo debe haberse borrado de memoria volviendo a un estado inicial de la aplicación.

Resultado obtenido Todos los datos se han borrado.

Acciones realizadas Ninguna.

7.1.4. Cargar XML válido habiendo un XML cargado y con uno o varios intervalos guardados

Esta prueba contempla el hecho de cargar un XML cuando ya hemos creado uno o varios intervalos listos para ser guardados

Resultado esperado Todos los datos cargados en la aplicación eliminados, sobre todo dando mucha importancia a que se eliminen los intervalos.

Resultado obtenido Todos los datos se han eliminado correctamente.

Acciones realizadas Ninguna por obtener el resultado esperado.

7.1.5. Cargar un XML no válido

Se utiliza para comprobar si la función de validación funciona correctamente.

Resultado esperado Se mostrará un error informando de que el XML que se ha intentado cargar no valida contra el XSD.

Resultado obtenido El archivo se valida e intenta cargarse produciendo una excepción.

Acciones realizadas Se revisa tanto el código de validación como el XSD para descubrir dónde estaba el fallo. Finalmente se descubrió que el fallo estaba en el XSD por lo que se arregla para que valide correctamente.

7.2. Crear intervalo

7.2.1. Preparación de las pruebas

Para poder crear un intervalo es necesario tener cargado un XML, y además haber cargado, al menos, una propiedad en el sistema y tener un rango seleccionado.

7.2.2. Guardar el rango no habiendo ninguno guardado previamente

Se probará si guarda correctamente un rango cuando no hay ninguno guardado de antes. Aquí se pone a prueba el comportamiento de cuando se van a guardar rangos que no entran en conflicto con ninguno.

Resultado esperado El rango se guarda en la colección de intervalos a guardar en disco, y se muestra un mensaje informando de que se ha añadido correctamente.

Resultado obtenido El resultado obtenido es el esperado, se ha añadido correctamente y se muestra el mensaje de información

Acciones realizadas Ninguna al obtener el resultado esperado

7.2.3. Guardar un rango que solape a otro previamente guardado

Esta prueba contiene a su vez cuatro pruebas que han sido agrupadas, ya que un intervalo se puede superponer a otro de cuatro maneras, tal y como se ve en la Figura 7.1.

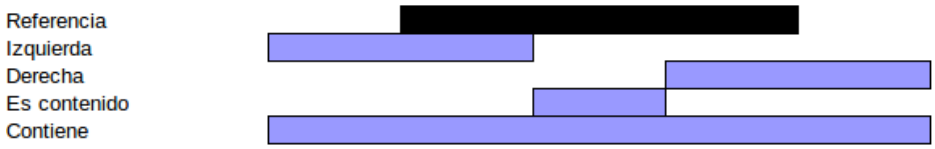


Figura 7.1: Modos solapamiento

El cuadrado negro representa el intervalo que está guardado en el sistema, y los azules-lilas representan el que va a intentarse guardar.

Resultado esperado Las cuatro veces que intenten guardarse esos intervalos debe salir un mensaje informando de que el intervalo a guardar solapa con alguno previamente guardado, y no debe guardarse.

Resultado obtenido Para esos cuatro casos se obtiene el resultado esperado, pero en el caso extremo en el que el intervalo guardado y el intervalo a guardar son iguales se guarda, dando lugar a incongruencias. Hay que tener en cuenta que en el caso de empezar y acabar en el mismo punto es la situación de "contiene".

Acciones realizadas Se ajustan las comprobaciones, se cambian los "$<$" y "$>$" por "$\leq$" y "$\geq$".

7.2.4. Guardar un intervalo vacío

Lo que se prueba aquí es que no se pueda guardar un intervalo vacío, ya que sería llenar el XML resultante con etiquetas vacías. Para probar se seleccionará un rango vacío y se pinchará en crear intervalo.

Resultado esperado Un mensaje de advertencia indicando que es necesario seleccionar un rango, y el intervalo no se guardará.

Resultado obtenido No se muestra nada y se guarda el intervalo

Acciones realizadas Se crea la condición en la que se muestre el diálogo y que no se guarde nada, si el inicio del intervalo es igual al final del intervalo.

7.3. Guardar paso o situación

7.3.1. Preparación de las pruebas

Para guardar un paso o situación, no se necesita ninguna preparación especial. Para guardar un paso o una situación que tenga sentido, se requiere haber creado al menos un intervalo.

7.3.2. Guardar paso o situación

Al pinchar en guardar paso, se mostrará el diálogo de guardado. Dependiendo de si el usuario quiere guardar o no, se realizará una acción u otra.

Resultado esperado Si el usuario decide guardar, se guardarán todos los intervalos creados previamente y se descargarán de memoria. Si no se decide guardar, no se hará nada y se podrá seguir trabajando.

Resultado obtenido Los resultados son los esperados.

Acciones realizadas Ninguna.

7.4. Añadir vídeo

7.4.1. Preparación de las pruebas

Para añadir un vídeo no se requiere de acciones previas. Se pueden añadir en cualquier momento.

7.4.2. Añadir un vídeo que no existe

Al añadir un vídeo que no existe, se añade a la lista de vídeos cargados.

Resultado esperado El vídeo se añade.

Resultado obtenido El vídeo es añadido

Acciones realizadas Ninguna.

7.4.3. Añadir un vídeo que ya existe

Al añadir un vídeo que ya existe, no se añade a la lista de vídeos cargados.

Resultado esperado El vídeo no se añade a la vista de vídeos cargados y además no se añade a la lista de vídeos cargados.

Resultado obtenido El vídeo se añade tanto a la vista como a la lista de vídeos cargados.

Acciones realizadas Implementadas las interfaces *IEquatable<T>* e *IComparable<T>* utilizando como método de comparación el nombre del archivo cargado. Se requieren esas dos interfaces para que el árbol rojo-negro funcione correctamente. En el caso de *IComparable<T>* se ordena alfabéticamente por nombre del archivo cargado.

7.5. Cargar propiedad u observación

Todas las pruebas tienen en común que al ser cargadas deben mostrar el rango seleccionado por otras propiedades ya en memoria.

7.5.1. Preparación de las pruebas

Para poder cargar una propiedad u observación es requisito indispensable que haya sido cargado un XML válido

7.5.2. Cargar una propiedad no cargada

Al cargar una propiedad no cargada previamente se prueba el comportamiento por defecto de la aplicación.

Resultado esperado La propiedad se añade junto a su observación si no estaba añadida. Si no, simplemente se añade la propiedad a la observación que corresponde. Además en memoria también se habrán cargado correctamente.

Resultado obtenido El resultado es el esperado.

Acciones realizadas Ninguna.

7.5.3. Cargar una propiedad ya cargada

En esta situación se prueba cuando la propiedad ya está cargada y el usuario intenta volver a cargarla

Resultado esperado La propiedad no debe cargarse ni visualmente ni en memoria.

Resultado obtenido La propiedad no se carga visualmente pero sí en memoria, y el software al no ser consciente de que ese dato ha sido cargado, causa una fuga de memoria pudiendo dejar al pc sin memoria disponible.

Acciones realizadas Se sigue paso a paso la ejecución hasta descubrir que la comprobación de si ya estaba cargada se realizaba contra el objeto equivocado, causando que siempre podía ser añadido. Se sustituyó por un árbol rojo-negro porque una de sus propiedades es que sólo puede haber un objeto de cada. También se implementaron las mismas interfaces que a los vídeos, utilizando como comparación el nombre de la propiedad, al igual que para insertarlo en el árbol.

7.5.4. Cargar una observación no cargada

Igual que con la propiedad, solo que al cargar una observación se añaden tanto la observación como todas las propiedades. También se puede realizar esta acción cuando algunas propiedades de la observación ya han sido cargadas.

Resultado esperado Si nada de esa observación había sido cargado previamente, se añadirán todas las propiedades de esa observación tanto en memoria como visualmente.

Si ya había alguna propiedad de esa observación cargada, se añadirán las restantes.

Resultado obtenido El resultado es el esperado.

Acciones realizadas Ninguna.

7.5.5. Cargar una observación ya cargada

Podría considerarse un subcaso de la prueba anterior. Pero se va a considerar que una observación cargada es aquella que tiene todas sus propiedades añadidas al sistema.

Resultado esperado No debe suceder nada, las propiedades al ya estar cargadas no deben añadirse ni en memoria ni visualmente.

Resultado obtenido El esperado.

Acciones realizadas Ninguna.

7.6. Cerrar observación

7.6.1. Preparación de las pruebas

Para poder cerrar una observación, se requiere cargar un XML válido y además haber añadido por lo menos una propiedad.

7.6.2. Cerrar observación

Al pinchar sobre la "X" de la esquina superior derecha de la observación, se quitará de vista y de memoria.

Resultados esperados La observación se quita de la vista y todas las propiedades asociadas se eliminarán de memoria y de la vista también.

Resultados obtenidos Se borran de la vista pero no se borran de memoria causando otra fuga de memoria.

Acciones realizadas Revisar donde puede fallar y añadir un evento que cuando se este ocultando en la vista, elimine la observación, y con ello, todas las propiedades de memoria.

7.7. Cerrar propiedad

7.7.1. Preparación de las pruebas

Para poder cerrar una observación, se requiere cargar un XML válido y además haber añadido por lo menos una propiedad.

7.7.2. Cerrar propiedad

Al pinchar sobre la "X" de la esquina superior derecha de la propiedad, se quitará de vista y de memoria.

Resultados esperados La propiedad se quita de la vista y de memoria.

Resultados obtenidos El resultado es el esperado.

Acciones realizadas Ninguna.

7.8. Seleccionar rango

7.8.1. Preparación de las pruebas

Se requiere cargar de al menos cuatro propiedades para probar el funcionamiento completo. Dos de una observación, y dos de otra.

7.8.2. Seleccionar un rango

En cualquiera de las propiedades abiertas pinchar y arrastrar para seleccionar un rango.

Resultados esperados El resto de propiedades deben sincronizar con el rango automáticamente.

Resultados obtenidos En lógica el valor ha cambiado pero el gráfico no se refresca.

Acciones realizadas Investigar la manera de refrescar manualmente los gráficos. Finalmente se descubre que la función *InvalidatePlot(true)* de la clase *PlotView* realiza esa tarea.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Conclusiones

Para finalizar, en una memoria no puede faltar una sección en la cual el desarrollador hace una reflexión sesuda sobre el trabajo realizado, qué es lo que se planificó y lo que ha sido realmente...

8.1.1. Reflexiones

Pues allá vamos. Empecemos por la reflexión sesuda. Con todo proyecto grande en el que me involucro me pasa siempre lo mismo, a lo que yo llamo "el síndrome del programador". Básicamente consiste en que da igual lo que hagas, tu código nunca va a ser lo suficientemente bueno, tu código siempre va a necesitar de nuevas funcionalidades, funcionalidades que ni siquiera se pedían, pero como se le profesa un "cariño" especial al software, siempre se intenta mejorar, aunque sea con nimiedades.

Eso nos lleva a no centrarnos en las tareas principales, se tiende a no hacer el software que nos piden sino, el software que el propio desarrollador querría si fuese él quien ha encargado dicho software. Esto es algo muy peligroso, porque podemos meternos en un bucle de hacer/deshacer cosas épico, ya que lo que hoy parece una idea brillante, mañana se te ocurre una manera mejor de hacerlo, y siempre se piensa que la nueva manera es óptima. Pero nada mas lejos de la realidad. En muchas ocasiones el cambio está justificado, si se trata de refactorizar, mejorar diseño etc. Mi opinión es que únicamente hay que cambiar algo que ya funciona cuando la mejora es visible. Es decir, que

se haya cambiado un algoritmo O(n) por uno O(1) u O(ln n)... O si la nueva manera vuelve al código mucho más legible y entendible.

Este síndrome no afecta solo a la parte de diseño e implementación, sino también a la interfaz gráfica: "Es que si ponemos este botón aquí queda más claro...", "Si añadimos una nueva ventana de opciones es más amigable con el usuario...". A lo que yo digo, si no se piden esos cambios, son bobadas. Porque primero, esos cambios no te los van a pagar, y segundo, tal vez el cliente no los quiera y te toque deshacer el trabajo.

He de admitir que yo mismo he sufrido ese síndrome durante este proyecto, obsesionándome a ratos con tonterías que no afectaban al funcionamiento, y que además lo previo no podía considerarse que estuviera mal... Vamos, llegué a cambiar algoritmos enteros, sin mejorar su eficiencia simplemente porque pensé que el código sería mas comprensible. Eso si, sin olvidar que el código previo tampoco es que fuera chapucero.

En otra situación en la que sufrí dicho problema, es cuando se implementó la funcionalidad de crear un intervalo para después poder guardarlo a disco. Me pareció muy interesante crear otro panel lateral en el que mostrar los intervalos que habían sido creados y que estaban en la lista de intervalos a exportar. Puede ser una buena idea, que aún sigo pensando que merece la pena implementarlo, pero no era un requisito inicial, y además trae consigo otro tipo de implicaciones. Habría que mantener ese listado de alguna manera enlazado con la lista de los intervalos, si se muestra, tiene sentido que puedan ser tanto editados, como borrados. A simple vista parece un añadido fácil, pero no se sabe hasta que punto puede llegar a modificar el comportamiento actual, y cuanto tiempo me llevaría hacerlo, por lo que finalmente fue descartado.

Por otro lado, una de las razones personales de elegir este TFG fue porque me obligaba a salir de mi zona de confort. Habría sido muy fácil elegir un TFG que manejase materias que se me dieran bien y me gustasen especialmente, tecnológicamente hablando. A mi no me gusta crear interfaces gráficas, y este TFG giraba sobre todo a un software con una interfaz gráfica compleja. Jamás había trabajado con WPF ni con las versiones nuevas de .NET. Puede parecer que no es un cambio grande, pero hay un mundo de diferencia entre Java y .NET, las cosas se hacen muy diferente aunque sintácticamente sean muy similares.

Tampoco había usado nunca librerías externas de gráficos ni de distribución de ventanas. Ha sido un proyecto que ha presentado muchos retos, situaciones en las que nunca había trabajado previamente, pero que creo que han sido resueltas de una manera bastante elegante y eficiente para ser la

primera vez que se trabajan con ellas. Eso si, es muy mejorable todo. Aparte de no haber utilizado estas bibliotecas, está el problema de que ninguna de las usadas tenía una documentación completa, por lo que se ha tenido que utilizar mucho los foros, y realizar un poco prueba y error. Incluso se ha llegado a investigar el propio código fuente de estas herramientas para ver como realizaban las cosas.

8.1.2. Una historia de planificaciones y realidades

No voy a ser yo quien diga que una planificación temporal sea algo inútil. Pero sí voy a ser yo quien diga que trabajar en base a una planificación temporal es un esfuerzo fútil. Y más si se trata de un software como MIPS.

La gente que tiene un perfil tirando a gestor, y poco técnico, tiende a pensar que en el mundo del software es fácil que la planificación y la realidad vayan de la mano. Se equivocan, sobre todo cuando no estás haciendo un software propio, si no que estás creando un software por encargo, donde los requisitos pueden cambiar día si, y día también. Se tiende a cometer el error de utilizar la planificación temporal como una hoja de ruta inamovible, sin tener en cuenta la incertidumbre que rodea a todo proyecto de software.

En mi caso concreto, seguir la planificación no ha sido un gran problema, ya que al tratarse de sprints que contenían a su vez varias tareas los cambios no son tan visibles.

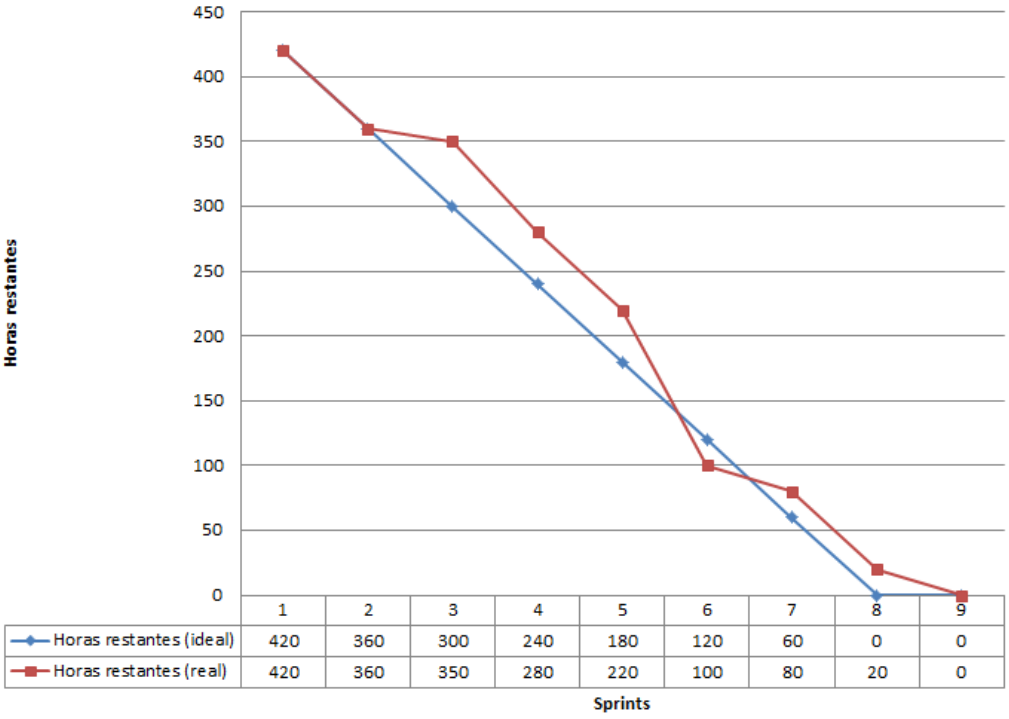
Aun así, puedo afirmar que las cosas han ido mejor de lo esperado, ya que este proyecto tenía un alto grado de incertidumbre al no conocer ni la plataforma de desarrollo, ni las bibliotecas utilizadas, la mayoría con documentación bastante deficiente.

En la Figura 8.1 se puede ver las tareas que me quedaban en cada iteración y cual era el progreso ideal para acabarlo a tiempo. Tal y como se puede observar, las dos líneas no van ni mucho menos cerca.

Finalmente, aunque con un mes de retraso, el proyecto finalizó satisfactoriamente.

Esto sirve para ver que aunque en algunos sprints se vaya con “retraso”, no implica que el proyecto en general vaya con retraso. Porque a veces se va con retraso, y a veces se va adelantado.

En la planificación inicial se hablaba de que se iban a dedicar 60h/mes al proyecto, pero finalmente no ha podido ser así. Bien es verdad que ha habido situaciones en las que se ha podido dedicar mas de 60h/mes, pero otras veces



	1	2	3	4	5	6	7	8	9
Horas restantes (ideal)	420	360	300	240	180	120	60	0	0
Horas restantes (real)	420	360	350	280	220	100	80	20	0

Figura 8.1: Burndown

apenas se ha podido dedicar unas escasas 20h/mes, por la carga de trabajo que había durante las clases.

Es por ello que al final el proyecto se ha retrasado casi un mes, con un desfase de 20 horas.

También me parece importante contar la experiencia que ha sido usar Scrum y Kanban. Seguir la metodología de Scrum no ha sido muy complicado. Es cierto que no se ha aplicado un Scrum de manual, ya que al ser un equipo de una persona, todas las tareas las realizaba la misma persona. Pese a todo, ha sido una manera de trabajar bajo mi punto de vista más agradable, ya que permite bastante flexibilidad. Y se ha hecho uso de esta flexibilidad en distintas ocasiones. Por ejemplo cuando se añadió la funcionalidad de poder guardar más de un intervalo en el mismo XML, o cuando se añadió el árbol lateral con las observaciones y las propiedades.

En cambio, Kanban ha sido un poco mas difícil de dominar. Normalmente en todos los sitios se insta a que el trabajador no se quede quieto si sufre un bloqueo, ya que de esa manera deja de ser "productivo". Por supuesto esa

manera de pensar, aunque yo personalmente no esté de acuerdo, ha pesado mucho y muchas veces, cuando no se sabía solventar un problema se tuvo la tentación de dejarlo de lado y empezar con otra cosa. En un par de ocasiones sí que se realizó eso mismo, pero estaba justificado ya que requería la interacción de una tercera persona para eliminar el bloqueo. También fue complicado mantener ordenado y actualizado el tablero Kanban. Este instrumento no sólo se utilizó para gestionar este TFG, también para gestionar las tareas de clase. Finalmente se cogió ritmo y me acostumbré a que antes de ponerse a trabajar había que actualizar tanto el diagrama burndown como el tablero Kanban. Ya que como se nos enseño en la asignatura de Sistemas de Gestión Integrados, aquello que no se puede medir no se puede mejorar.

Y pese a coger esa costumbre sistemática, fue imposible terminar el trabajo en el plazo planificado. Podría decirse que hay excusa a la hora de entregar con retraso, ya que una de las razones personales por las que elegí este TFG era porque me obligaba a salir de mi zona de confort como se ha comentado con anterioridad.

En cuanto a los riesgos, todos se habían previsto bien, pero finalmente se han cumplido dos de ellos, concretamente el riesgo de tener que deshechar una biblioteca que se hubiese elegido, y el de que se añadan al software nuevos requisitos.

Afortunadamente, con los nuevos requisitos al haber hecho un diseño aceptable, integrar los nuevos cambios no supuso un gran esfuerzo, por lo que tanto la prevención como el plan de contingencia funcionaron bien.

Por el contrario, con las librerías no hubo tanta suerte. Se realizó el plan de prevención, es decir, mirar con anterioridad si las bibliotecas cumplían con los requisitos para ser válidos para este proyecto, pero aun así fue imposible evitar el plan de contingencia. No se pudieron parchear las bibliotecas ya que son desarrollos muy complejos, por lo que hubo que desechar al menos en dos ocasiones la biblioteca de mostrar los gráficos, y concretamente esa es la causa del retraso sufrido en el proyecto.

8.2. Mejoras

Como todo software, siempre puede ser mejorado, y MIPS no es ninguna excepción. El software puede mejorarse sobre todo en tres niveles bien diferenciados: Apariencia, diseño y funcionalidades, eso sin contar, por supuesto, los bugs que no han sido descubiertos en la fase de testing y que casi seguro surgirán durante su vida útil en producción.

Los puntos clave en los que creo que este software puede mejorarse son sobre todo referidos al diseño.

Por ejemplo, pese a que lo he hecho lo mejor que he podido y que he sabido, seguro que en muchos puntos podría utilizarse SOLID de manera más efectiva. Un ejemplo claro de esto que digo es el uso del patrón Singleton. Tal y como he detallado en el capítulo de Análisis y diseño, muchos desarrolladores sostienen que no es buena idea utilizarlo, ya que las dependencias están en el código, no en las relaciones entre las clases. Una clase que utilice el patrón Singleton tiene un ámbito global y utilizar objetos globales para no pasar la referencia de clase en clase está considerado un *code smell* [1] [2].

No sólo es un *code smell* sino que además incumple el principio de responsabilidad única de SOLID. Y lo incumple porque la clase que utiliza el patrón tiene dos responsabilidades: Controlar que sólo exista una única instancia y además su propia lógica de negocio.

Es por ello que una de las mejoras que propongo, es sustituir los Singleton por patrones Factory o patrones Builder para que sea esa clase externa la que limite la creación de instancias. De esta manera es más escalable, ya que es muy probable que lo que hoy es único, mañana sea múltiple.

Otra cosa que no se ha realizado, y que se considera una buena práctica de programación son las pruebas unitarias. Una vez implementadas podría utilizarse un enfoque más del estilo de extreme programming (XP) [3], esto es, que todo gire en torno a las pruebas unitarias. Un bug no es un error, sino una prueba que no está escrita. Y si esa prueba que no estaba escrita falla, entonces es que hay un requisito que no se está cumpliendo.

Muy relacionado con las pruebas unitarias, es el Test Driven Development (TDD) en el cual no se programa nada hasta que no se hayan escrito todas las pruebas. Y un software se considera terminado cuando todas las pruebas validan. Obviamente hay que probar todas las situaciones posibles. Yo propongo utilizar esta metodología ya que así, en todo momento se sabe qué es lo que funciona, qué es lo que no funciona, y hasta que punto ha sido probado el software. Igual que lo que he dicho en el capítulo de Verificación y evaluación, las pruebas no deben probar que los métodos hagan lo que se supone, sino el comportamiento de la aplicación.

Otro de los ámbitos que tiene mucho margen de mejora es en el procesamiento paralelo. Sí que es cierto que se ha utilizado, pero las posibilidades

[1] `https://en.wikipedia.org/wiki/Code_smell`

[2] Lista incompleta de smells: `http://blog.codinghorror.com/code-smells/`

[3] `http://www.extremeprogramming.org/`

que da son enormes, y no se ha utilizado más que en dos casos muy concretos. Con unas semanas de estudio y con algo de refactorizado del código estoy seguro que se le podría sacar mas partido a los múltiples núcleos de los ordenadores actuales.

Por otro lado, como MIPS en un futuro será integrado dentro de ULISES, requerirá de un poco de trabajo si no se quiere mantener como una aplicación independiente. Para ello, habría que convertir `MainWindow` en una subclase de `UserControl`, para que de esta manera pueda ser añadido como una ventana acoplable al otro sistema.

Por último en cuanto a temas de diseño se refiere, yo propongo cambiar los temporizadores de la aplicación y utilizar los *data bindings* que proporciona .NET. Para poder hacer esto habría que aprender bien a utilizar el patrón MVVM, que si bien se ha utilizado en este proyecto no se le ha exprimido todas las características.

Finalmente, estéticamente hablando, yo considero que es una aplicación intuitiva, pero se podría conseguir que todo fuera más claro si los gráficos entre observaciones no compartieran el mismo color, es decir, si la observación 1, tiene los gráficos de color verde, que la observación 2 los tenga de color azul. Ya que así, queda más bonito a la vista, y es más fácil no confundir a que observación pertenece cada propiedad. También hay otra mejora cosmética, y es que cuando el usuario haga doble click sobre un elemento del árbol de propiedades y observaciones, si ese ítem ya estaba cargado, que cogiera el foco, en vez de no hacer nada, que es lo que hace actualmente. Y también sería interesante crear un diálogo de opciones, ya que ahora las opciones están metidas dentro de un XML y hay que editarlo a mano.

Para finalizar, muchas de las propuestas requieren de mucho trabajo para apenas notar una mejoría, como puede ser el procesamiento paralelo. En casos muy extremos puede tener una justificación, pero con la capacidad de cálculo de los microprocesadores actuales y la cantidad de datos que va a manejar este software tal vez no merezca la pena implementarlo. En cuanto a las cuestiones de diseño y buenas prácticas como TDD yo, personalmente, sí que intentaría implementarlas, ya que si bien no van a conseguir que el software sea mas potente, sí que va a conseguir que sea mas robusto y fácil de expandir. En cuanto a tiempo de desarrollo, cambiar los Singleton por los patrones Factory o Builder no debería llevar demasiado tiempo. En cambio, preparar las pruebas para TDD sí que llevará bastante tiempo, ya que preparar todos los comportamientos posibles de un software es complicado y además lleva tiempo, pero es un tiempo bien invertido.

En el capítulo de Análisis y diseño se ha comentado que no se ha aplica-

do el principio de inversión de dependencias, aquel que dice que los objetos deben depender de abstracciones y no de concreciones. Ciertamente no habría que realizar cambios en el código fuente, solo habría que cambiar las cabeceras de los métodos para sustituir los objetos concretos por objetos del tipo *metodo1(Class<T>implements interface1, interface2 unParam)* y por supuesto implementar las interfaces. Igual que antes, no son cambios que obliguen a rehacer gran parte del software, sino que se puede trabajar sobre lo que ya está hecho.

En cambio, de todos los cambios, el que parece el mas tonto, que es el de sustituir los temporizadores por data bindings, es el único que creo que no merece la pena tocarlo. Para poder utilizarlo habría que quitar todos los timers, establecer los contextos de datos en la construcción y finalmente enlazar. Yo, particularmente no lo haría porque va a requerir bastante tiempo para no mejorar ni en diseño ni en eficiencia.

Referencias

Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., y Yergeau, F. (Noviembre 2008). *Extensible markup language (xml) 1.0 (fifth edition).* Descargado de http://www.w3.org/TR/2006/REC-xml11-20060816/

Button, B., y Densmore, S. (25/05/2004). *Why singletons are evil.* Descargado de http://blogs.msdn.com/b/scottdensmore/archive/2004/05/25/140827.aspx

CEIT. (2013). Manual de usuario intrasim [Manual de software informático].

Chapell, D. (Septiembre 2006a). *Addressing the problem: What windows presentation foundation provides.* Descargado de http://msdn.microsoft.com/en-us/library/aa663364.aspx#introducingwpf_topic3

Chapell, D. (Septiembre 2006b). *Introducing windows presentation foundation.* Descargado de http://msdn.microsoft.com/en-us/library/aa663364.aspx

Geek, y Poke. (2013). *Programming is an art.* Descargado de http://geek-and-poke.com/geekandpoke/2013/9/18/the-art-of-programming

Gossman, J. (04/03/2006). *Advantages and disadvantages of m-v-vm.* Descargado de http://blogs.msdn.com/b/johngossman/archive/2006/03/04/543695.aspx

Martin, R. C. (s.f.-a). *The interface segregation principle.* Descargado de http://www.objectmentor.com/resources/articles/isp.pdf

Martin, R. C. (s.f.-b). *Principles of ood.* Descargado de http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Martin, R. C. (s.f.-c). *The single responsibility principle.* Descargado de http://www.objectmentor.com/resources/articles/srp.pdf

Martin, R. C. (Enero 1996). *The open closed principle.* Descargado de http://www.objectmentor.com/resources/articles/ocp.pdf

Martin, R. C. (Marzo 1996a). *The dependency inversion principle.* Descargado de http://www.objectmentor.com/resources/articles/dip.pdf

Martin, R. C. (Marzo 1996b). *The liskov substitution principle.* Descargado de http://www.objectmentor.com/resources/articles/lsp.pdf

Scotland, K. (s.f.). *Aspects of kanban.* Descargado de http://www.methodsandtools.com/archive/archive.php?id=104

Scrum.org. (s.f.). *What is scrum?* Descargado de https://www.scrum.org/resources/what-is-scrum/

ScrumStudy. (2013). *Empirical process control.* Descargado de http://www.scrumstudy.com/scrum-empirical-process-control.asp

Apéndice A

Diagrama de clases completo

En este anexo se puede ver el diagrama de clases completo, con todos sus métodos, propiedades y atributos. Todos los miembros de las clases están ordenados por visibilidad.

Como el diagrama es demasiado grande, se puede visualizar online en: `https://github.com/RubenAgudo/TFG_Documentation/blob/master/Figures/ClassDiagramExpanded.png`

Si usted está visualizando este documento en pdf puede ampliar tanto como quiera para ver el diagrama en mayor detalle. Si por el contrario está impreso, se recomienda encarecidamente utilizar el link anterior.

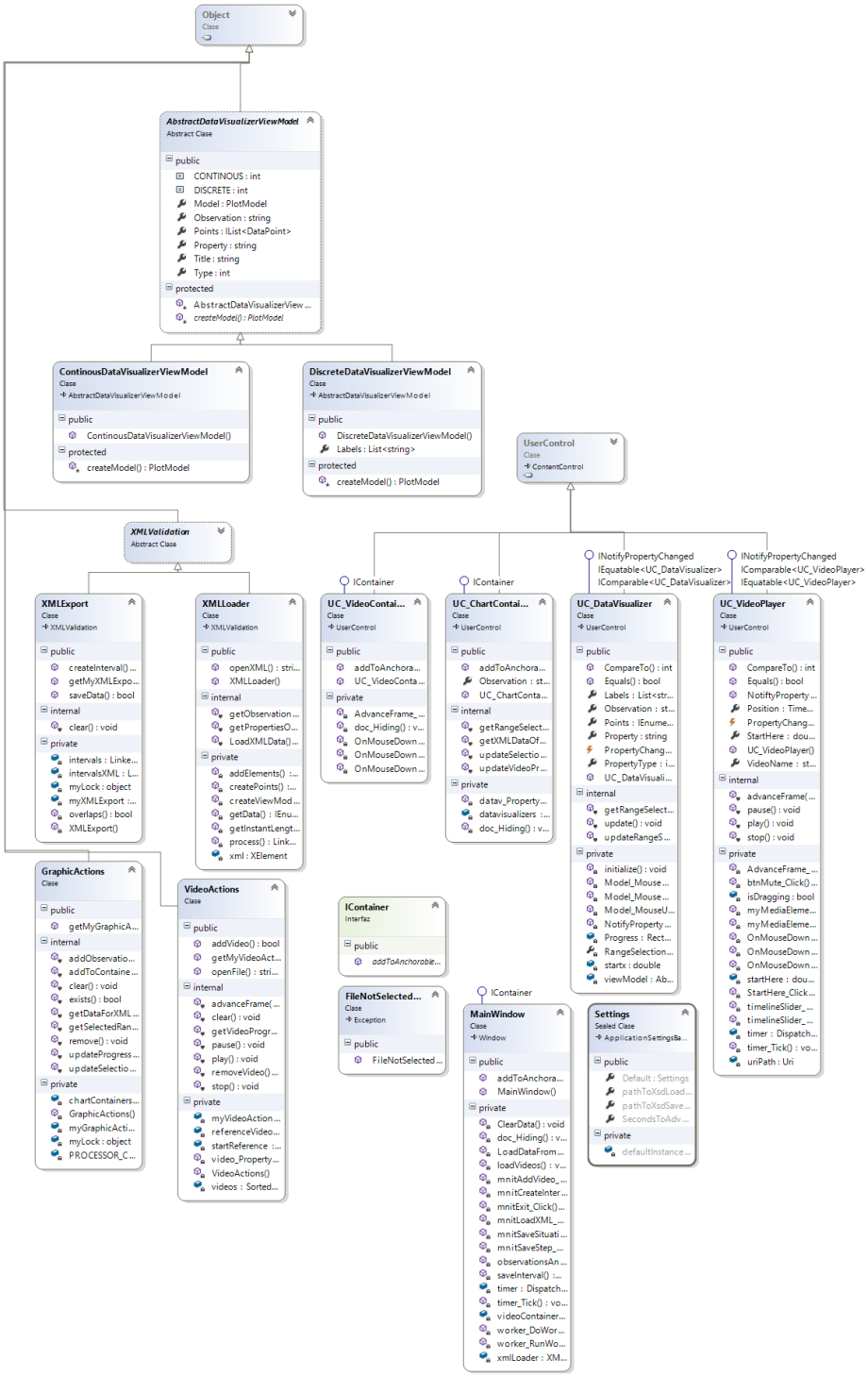


Figura A.1: Diagrama de clases completo

Apéndice B

Casos de uso extendidos

En esta apéndice se procederá a detallar cada caso de uso, dar una descripción de qué se hace, que actores toman parte, precondiciones, requisitos no funcionales, el flujo de eventos, así como sus poscondiciones.

De esta manera se podrá comprender más fácilmente el funcionamiento del software sin la necesidad de utilizar el software o mirar el código fuente e interfaz gráfica, que se encuentra al final de este anexo.

También se añade aquí el diagrama de casos de uso para que no haga falta volver al capítulo de Captura de requisitos.

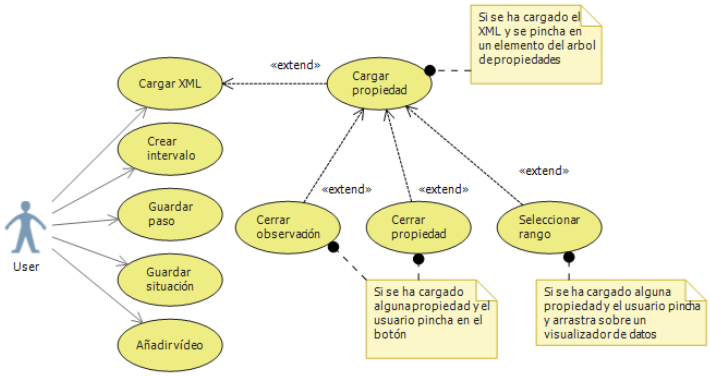
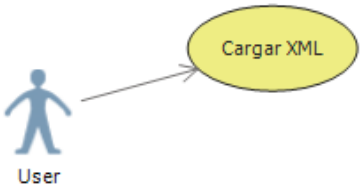


Figura B.1: Diagrama de casos de uso

B.1. Cargar XML



Nombre	Cargar XML
Descripción	Carga un XML que contiene las propiedades y las observaciones en el sistema. Únicamente se cargan si el archivo valida contra el XSD.
Actores	User
Precondiciones	Ninguna
Requisitos no funcionales	Ninguno
Flujo de eventos	1. El usuario pincha en File ->Load XML. 2. Se abre una ventana de selección de archivos. 3. **Si el usuario quiere cargar un fichero** *a*) Lo busca en el sistema de archivos [B.2] *b*) Lo selecciona y pincha en Abrir. *c*) **Si es un documento válido** 1) Las observaciones y sus propiedades aparecen en el árbol lateral. [B.3] *d*) **Si no** 1) Se muestra un error. [B.4] 4. **Si no desea cargarlo** *a*) Pincha en cancelar y se muestra un mensaje de que debe cargar un XML [B.5].
Poscondiciones	El árbol contendrá las observaciones y propiedades si era un documento válido, mostrará un error si era un documento inválido, o mostrará un aviso indicando de que se necesita cargar un XML para poder trabajar.
Interfaz gráfica	Figuras B.2, B.3, B.4 y B.5

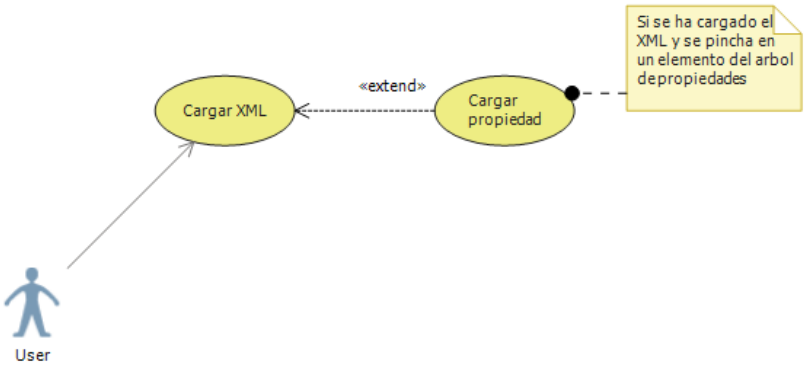
Cuadro B.1: Cargar XML

B.2. Añadir vídeo

Nombre	Añadir vídeo
Descripción	Abre uno o varios vídeos y los añade al contenedor de vídeos si ya había vídeos previamente cargados. Si no, añade también un contenedor de vídeos.
Actores	User
Precondiciones	Ninguna
Requisitos no funcionales	Ninguno
Flujo de eventos	1. El usuario pincha en Add Video. [B.6] 2. **Si el contenedor de vídeos no estaba cargado** *a*) Se crea y añade un contenedor de vídeos. 3. Se abre una ventana de selección de archivos. 4. **Si el usuario quiere cargar vídeos** *a*) Los busca en el sistema de archivos *b*) Los selecciona y pincha en Abrir. *c*) Se cargan si no estaban previamente cargados. 5. **Si no desea cargarlo** *a*) Pincha en cancelar.
Poscondiciones	El contenedor con los nuevos vídeos añadidos
Interfaz gráfica	Figuras B.6

Cuadro B.2: Añadir vídeo

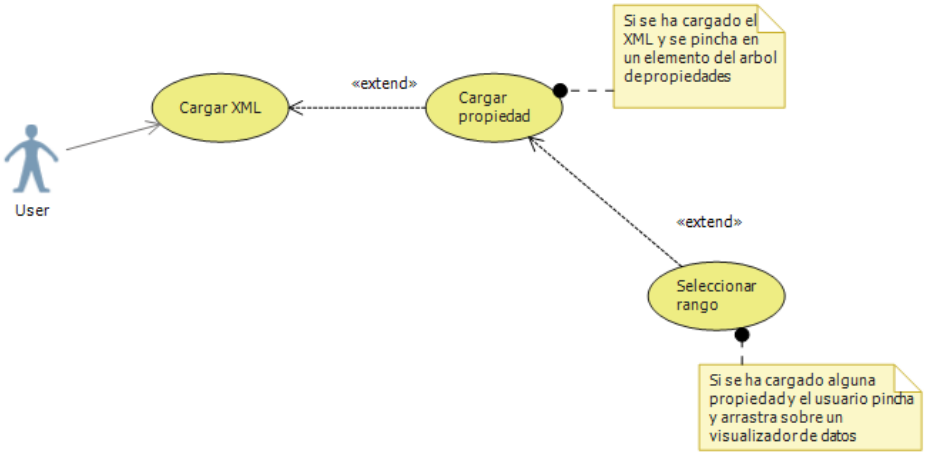
B.3. Cargar propiedad



Nombre	Cargar propiedad
Descripción	Carga una propiedad en el contenedor de gráficos correspondiente a la observación a la que pertenece. También puede añadir propiedades en grupo, haciendo doble click sobre el nombre de la observación. Si el contenedor no está creado, lo crea.
Actores	User.
Precondiciones	Ninguna.
Requisitos no funcionales	Ninguno.
Flujo de eventos	1. El usuario hace doble click en algún elemento del árbol de observaciones y propiedades. 2. Crea el contenedor si no estaba creado 3. **Si el elemento es una observación** *a*) Añade todos sus hijos no cargados al contenedor. 4. **Si el elemento es una propiedad** *a*) Añade la propiedad si no estaba cargada previamente.
Poscondiciones	El contenedor de la observación y todas las propiedades de la observación o la propiedad seleccionada dependiendo de dónde se haya hecho doble click y sincronizada con el rango existente, si es que lo hay.
Interfaz gráfica	Figuras B.7

Cuadro B.3: Cargar propiedad

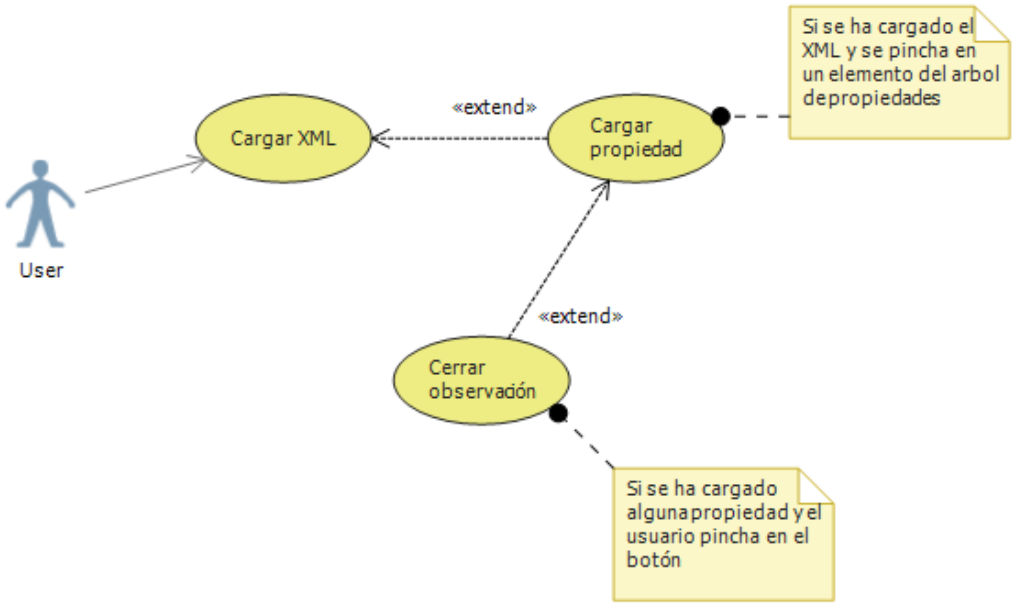
B.4. Seleccionar rango



Nombre	Seleccionar rango
Descripción	Al pinchar y mover el ratón sobre un visualizador de datos, se seleccionará un rango en todos los visualizadores de datos cargados en el sistema.
Actores	User
Precondiciones	Alguna propiedad tiene que estar cargada.
Requisitos no funcionales	Ninguno
Flujo de eventos	1. El usuario pincha en un visualizador de datos. 2. El usuario arrastra (manteniendo pinchado) el ratón. *a*) Cada vez que el rango seleccionado cambia, se notifica al contenedor padre. *b*) El contenedor, notifica al repositorio de contenedores, para que todos se sincronicen con el dato proporciado. *c*) Se recorren todos los visualizadores de datos para sincronizarse con el nuevo valor. 3. El usuario deja de pinchar, y se queda el rango seleccionado.
Poscondiciones	El rango quedará seleccionado en todos los visualizadores de datos cargados en el sistema.
Interfaz gráfica	Figuras B.8

Cuadro B.4: Seleccionar rango

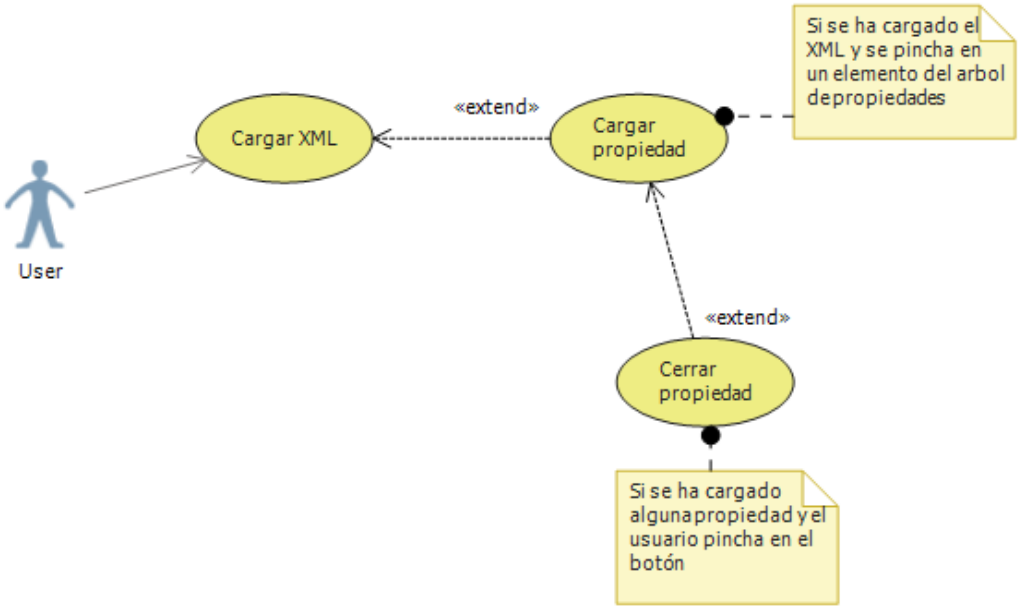
B.5. Cerrar observación



Nombre	Cerrar observación
Descripción	Elimina, tanto de manera visual, como de memoria, la observación seleccionada y todos sus visualizadores de datos (propiedades) asociadas.
Actores	User.
Precondiciones	La observación tenía que estar cargada.
Requisitos no funcionales	Ninguno.
Flujo de eventos	1. El usuario pincha sobre la "X" en la esquina superior derecha de la observación. 2. Se borra del repositorio, liberando sus recursos, y con ello, sus propiedades. 3. Finalmente se borra de la pantalla.
Poscondiciones	El espacio de trabajo contendrá todas las observaciones y propiedades menos la que se ha cerrado.
Interfaz gráfica	Ninguna relevante.

Cuadro B.5: Cerrar observación

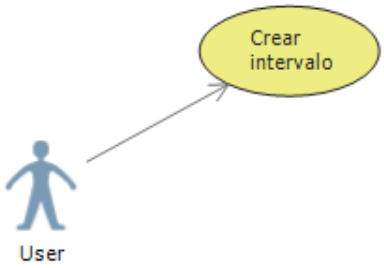
B.6. Cerrar propiedad



Nombre	Cerrar propiedad
Descripción	Elimina, tanto de manera visual, como de memoria, la propiedad seleccionada.
Actores	User.
Precondiciones	La propiedad tenía que estar cargada.
Requisitos no funcionales	Ninguno.
Flujo de eventos	1. El usuario pincha sobre la "X" en la esquina superior derecha de la propiedad. 2. Se borra del contenedor al que estaba asociada. 3. Finalmente se borra de la pantalla.
Poscondiciones	El espacio de trabajo contendrá todas las observaciones y propiedades menos la propiedad que se ha cerrado.
Interfaz gráfica	Ninguna relevante

Cuadro B.6: Cerrar propiedad

B.7. Crear intervalo



Nombre	Crear intervalo
Descripción	Añade el intervalo seleccionado a los intervalos a guardar en el xml si no se superpone con los que ya están guardados.
Actores	User
Precondiciones	Alguna propiedad debe estar seleccionada y un rango debe estar seleccionado.
Requisitos no funcionales	Ninguno
Flujo de eventos	1. El usuario pincha Create interval. 2. **Si el intervalo actual no se superpone con ninguno guardado previamente** *a*) Se guarda el intervalo en la lista de intervalos a exportar. *b*) Se muestra un dialogo notificando que la operación se ha realizado correctamente. [B.9] 3. **Si se superpone** *a*) Se muestra un dialogo notificando que el rango seleccionado se superpone con alguno guardado. [B.10]
Poscondiciones	Se guardará el nuevo intervalo en lista de intervalos a exportar.
Interfaz gráfica	Figuras B.9 y B.10

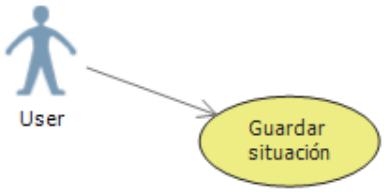
Cuadro B.7: Crear intervalo

B.8. Guardar paso

Nombre	Guardar Paso.
Descripción	Guarda todos los intervalos creados a disco con un elemento raíz “step”.
Actores	User
Precondiciones	Ninguna.
Requisitos no funcionales	Ninguno
Flujo de eventos	1. El usuario pincha en Save selected range ->Step. [B.11] 2. Se muestra un diálogo de guardado de datos. 3. **Si el usuario decide guardar el fichero.** *a*) Elige un nombre de fichero y pincha en guardar. [B.12] *b*) El fichero se guarda en la carpeta especificada. *c*) Se borran todos los intervalos guardados. 4. **Si pincha en cancelar** *a*) Se cierra la ventana y no se borran los intervalos, para poder seguir trabajando.
Poscondiciones	El fichero guardado en disco, o no, dependiendo de las acciones del usuario.
Interfaz gráfica	Figuras B.11 y B.12

Cuadro B.8: Guardar paso

B.9. Guardar situación



Nombre	Guardar situación
Descripción	Guarda todos los intervalos creados a disco con un elemento raíz “situation”.
Actores	User
Precondiciones	Ninguna.
Requisitos no funcionales	Ninguno
Flujo de eventos	1. El usuario pincha Save selected range ->Situation. [B.13] 2. Se muestra un diálogo de guardado de datos. 3. **Si el usuario decide guardar el fichero.** *a*) Elige un nombre de fichero y pincha en guardar. [B.14] *b*) El fichero se guarda en la carpeta especificada. *c*) Se borran todos los intervalos guardados. 4. **Si pincha en cancelar** *a*) Se cierra la ventana y no se borran los intervalos, para poder seguir trabajando.
Poscondiciones	El fichero guardado en disco, o no, dependiendo de las acciones del usuario.
Interfaz gráfica	Figuras B.13 y B.14

Cuadro B.9: Guardar situación

B.10. Figuras de los casos de uso

Todas las figuras asociadas a los casos de uso extendidos pueden encontrarse en esta sección

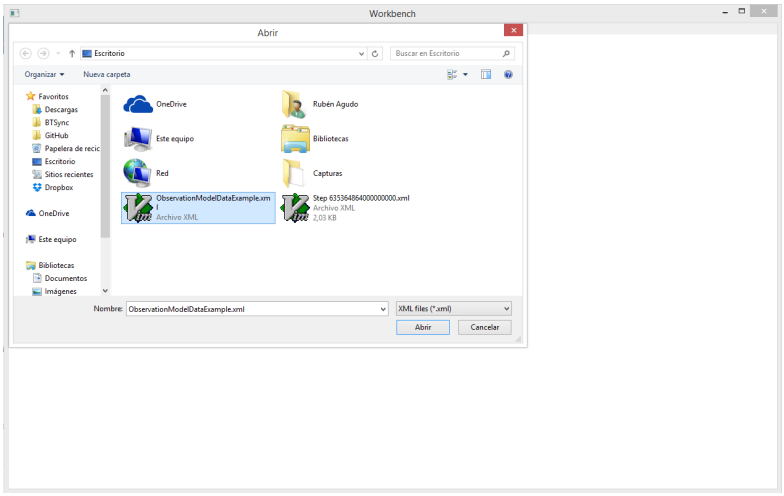


Figura B.2: Abrir XML 1

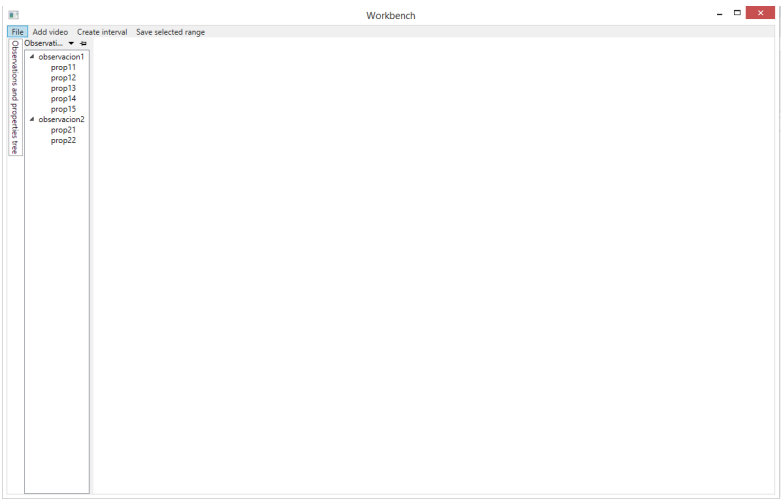


Figura B.3: Abrir XML 2

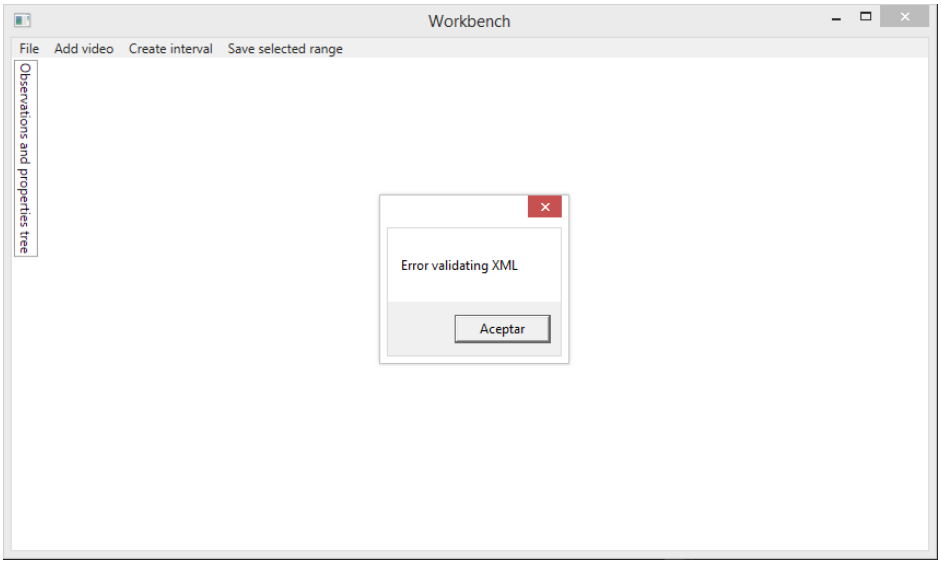


Figura B.4: Abrir XML 3

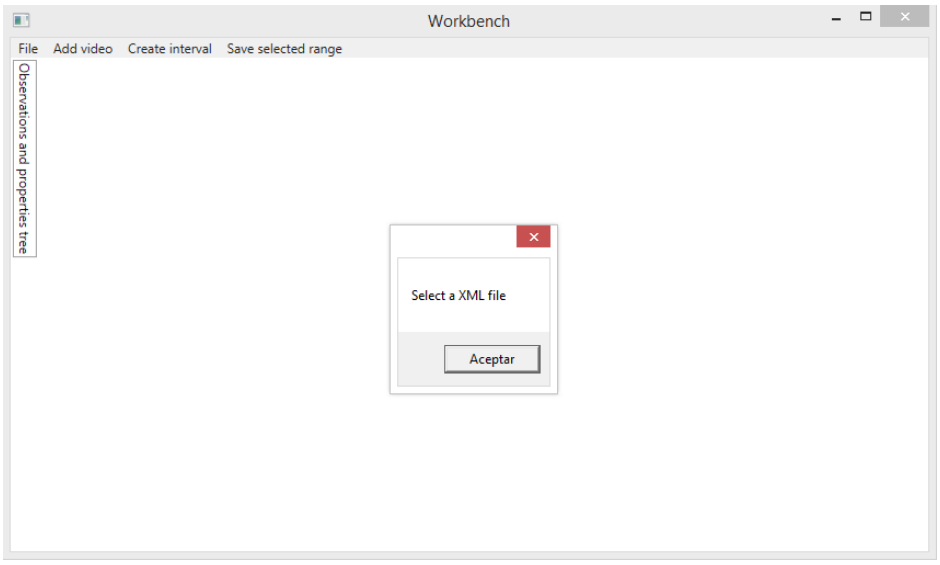


Figura B.5: Abrir XML 4

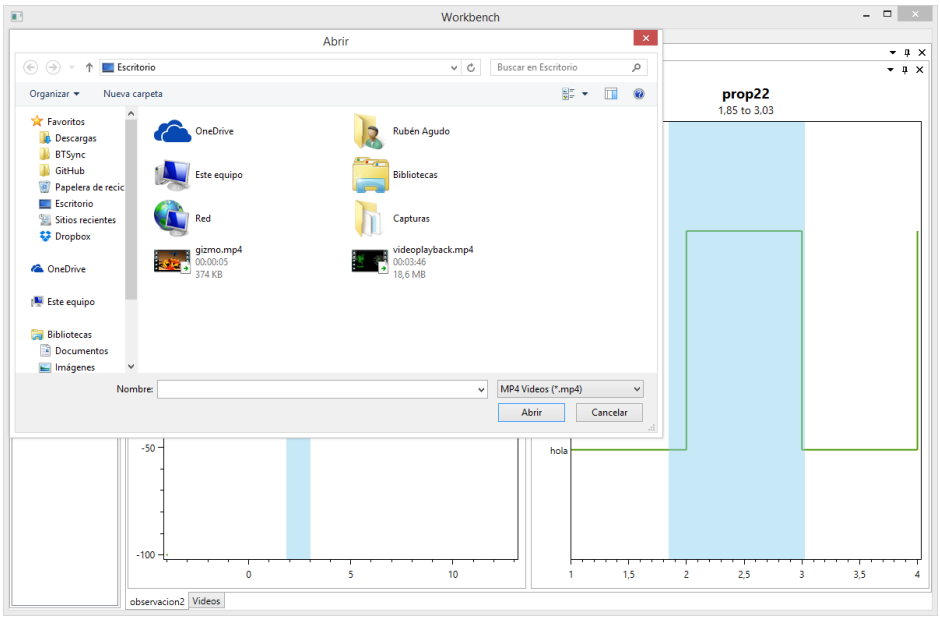


Figura B.6: Cargar Vídeo

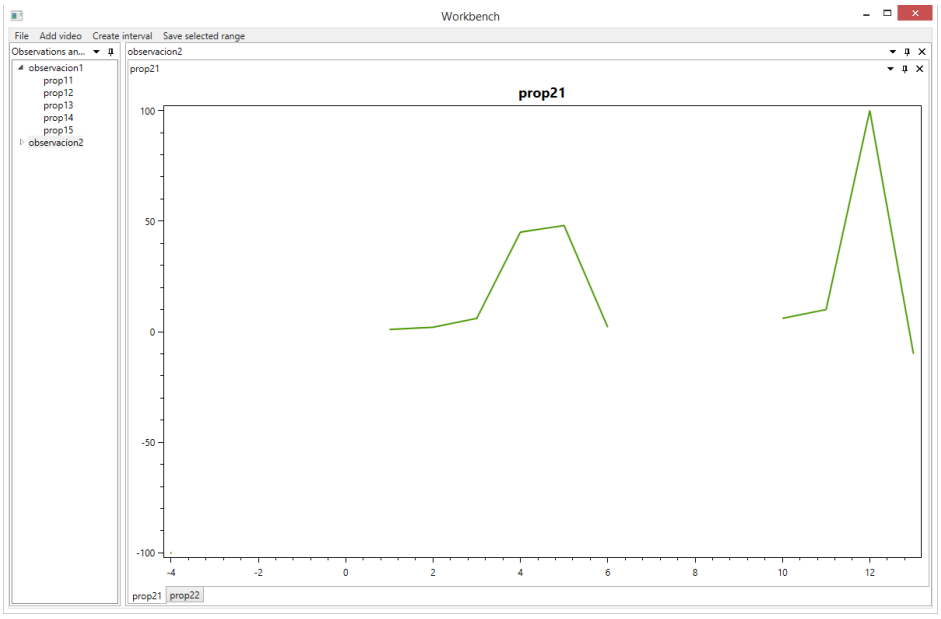


Figura B.7: Cargar Observación o Propiedad

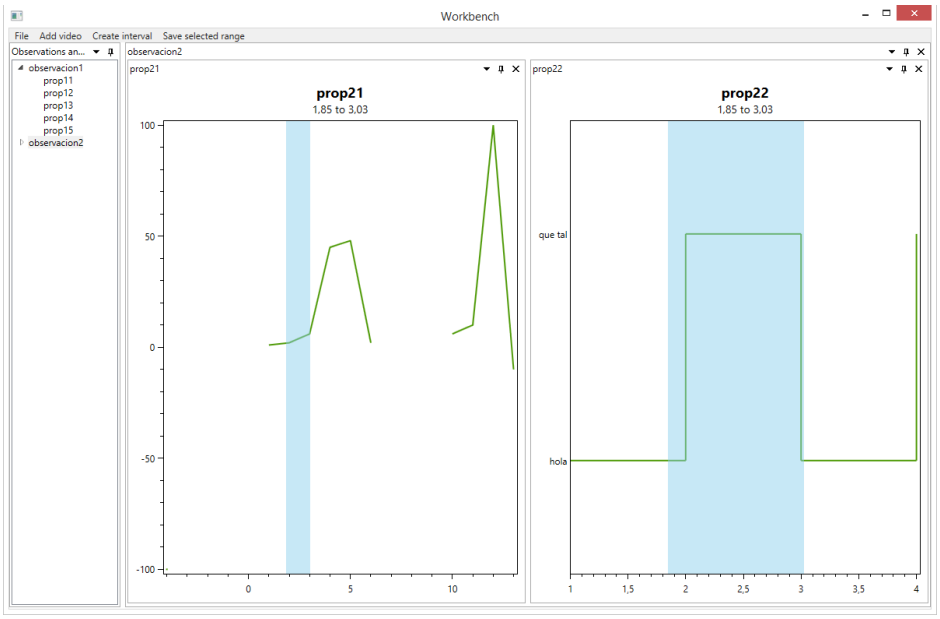


Figura B.8: Seleccionar rango

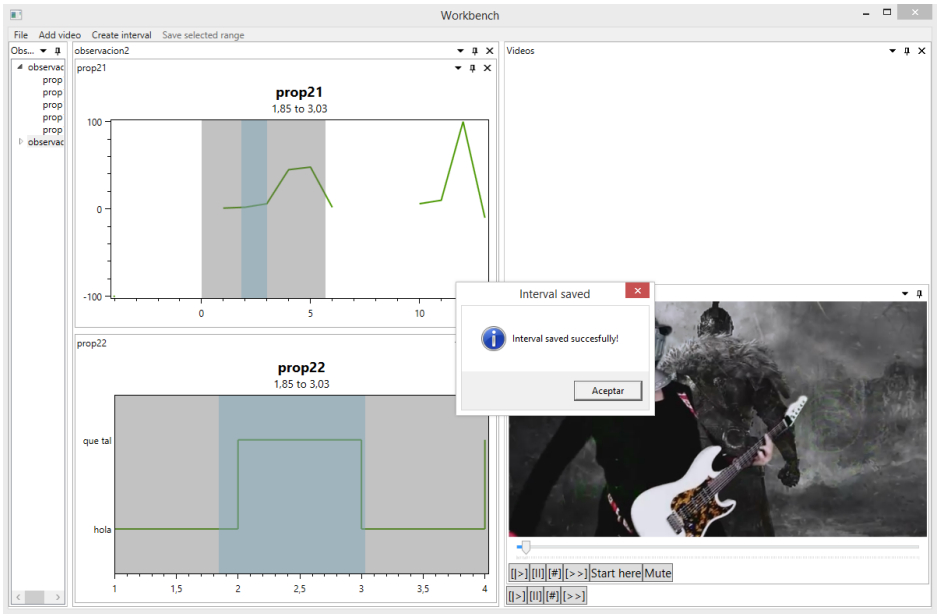


Figura B.9: Crear intervalo

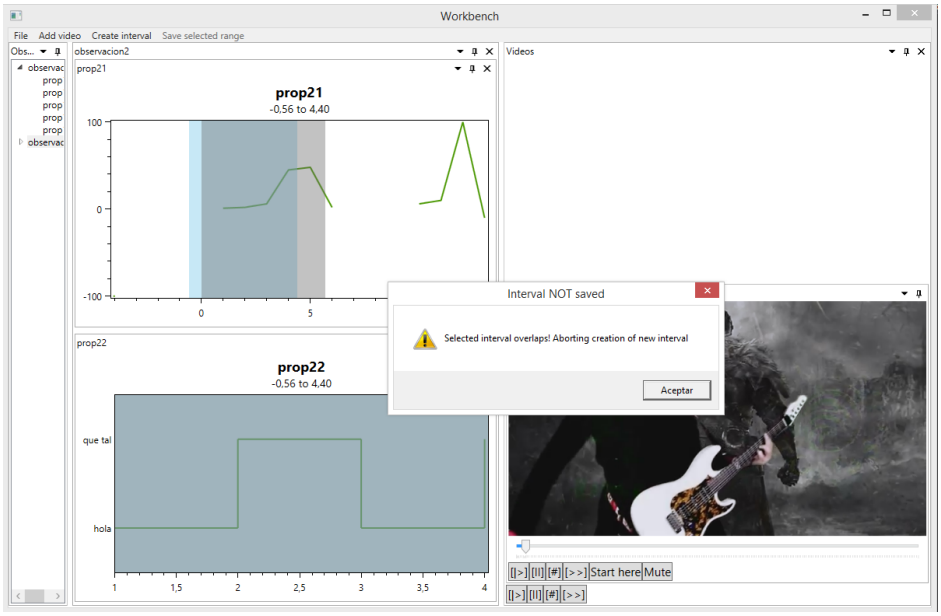


Figura B.10: Crear intervalo con error

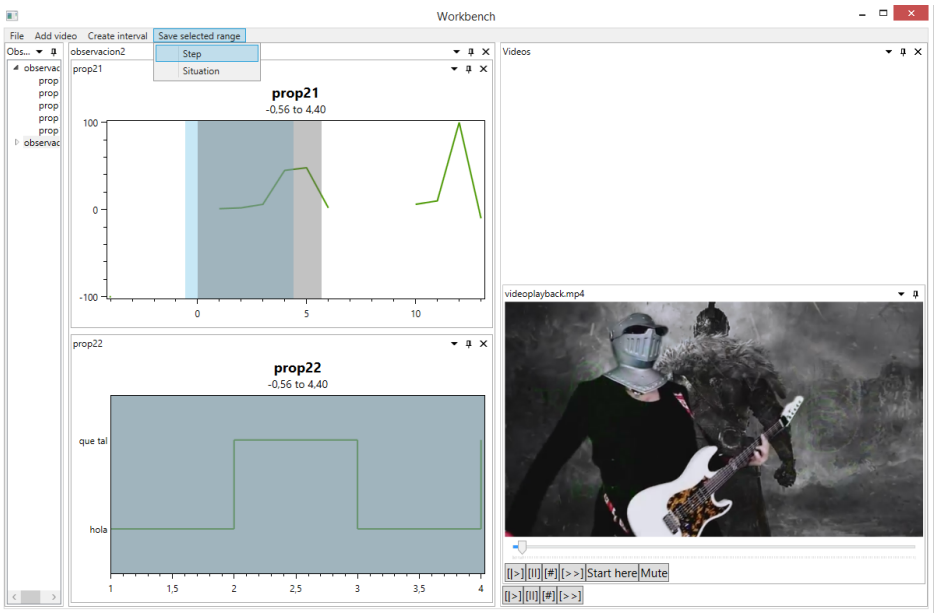


Figura B.11: Guardar paso 1

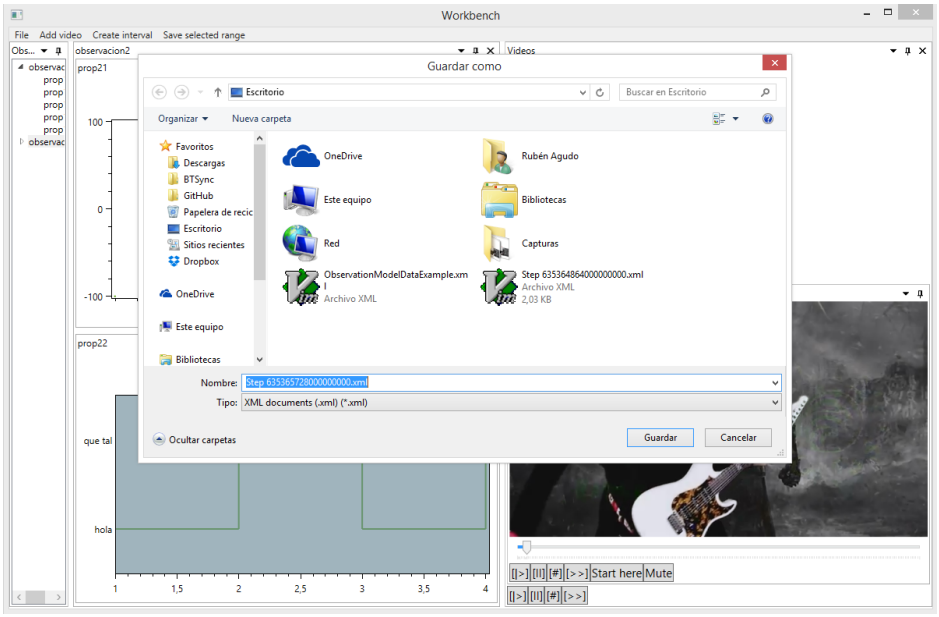


Figura B.12: Guardar paso 2

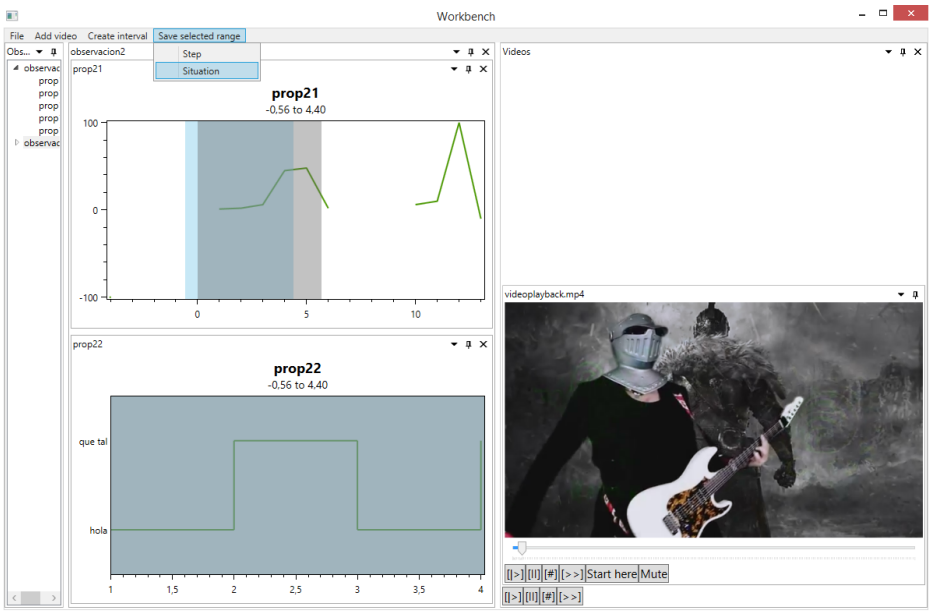


Figura B.13: Guardar situación 1

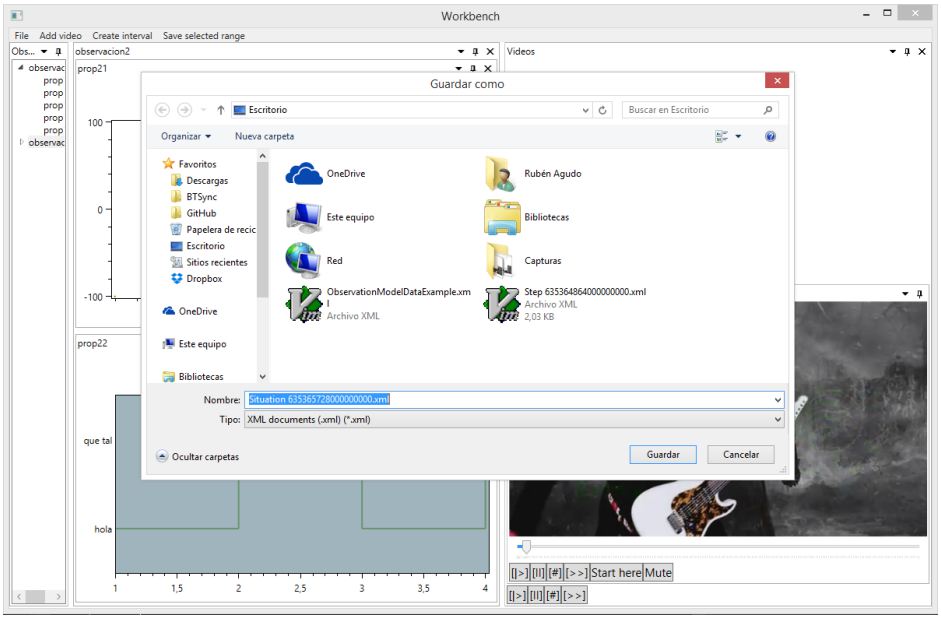


Figura B.14: Guardar situación

Apéndice C

Diagramas de secuencia

En este capítulo se listan todos los diagramas de secuencia relacionados con cada caso de uso.

Debido a que los diagramas de secuencia son demasiado grandes y no se ven adecuadamente, pueden consultarse aquí: `https://github.com/RubenAgudo/TFG_Documentation/tree/master/Figures/Secuencia`

Pese a todo, si está visualizando este documento en formato pdf, puede aumentar tanto como crea necesario para ver correctamente los diagramas de secuencia.

C.1. Cargar XML

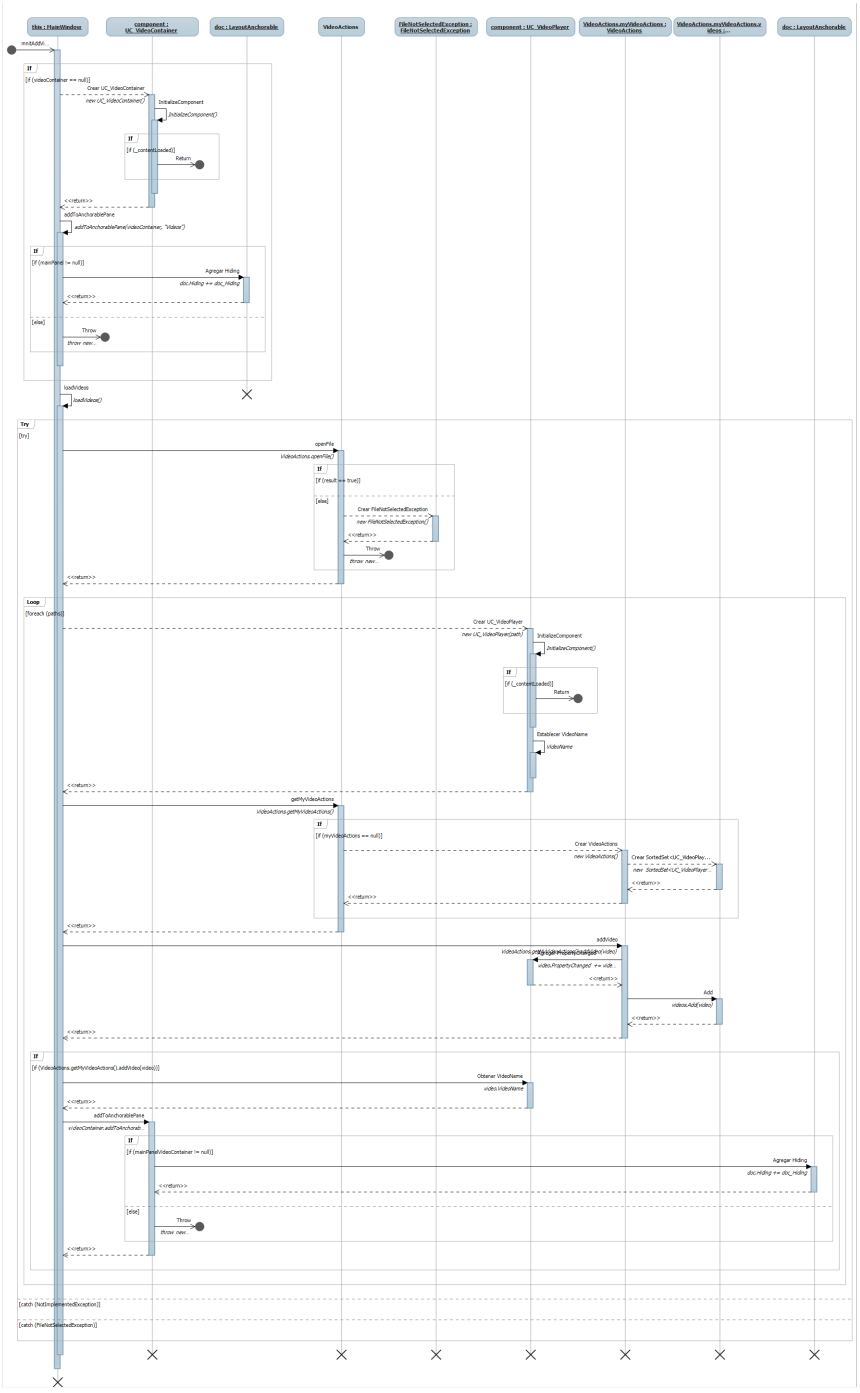


Figura C.1: Cargar XML

C.2. Cargar observación o propiedad

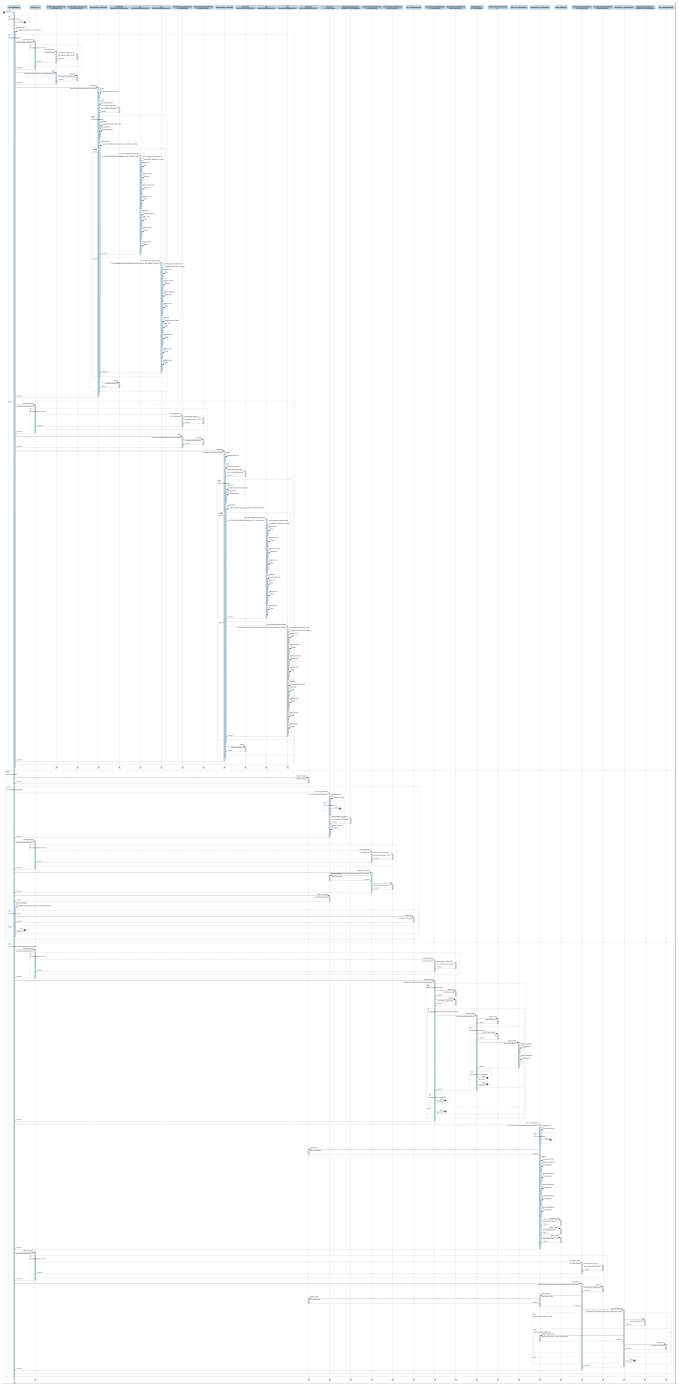


Figura C.2: Cargar observación o propiedad

C.3. Añadir vídeo

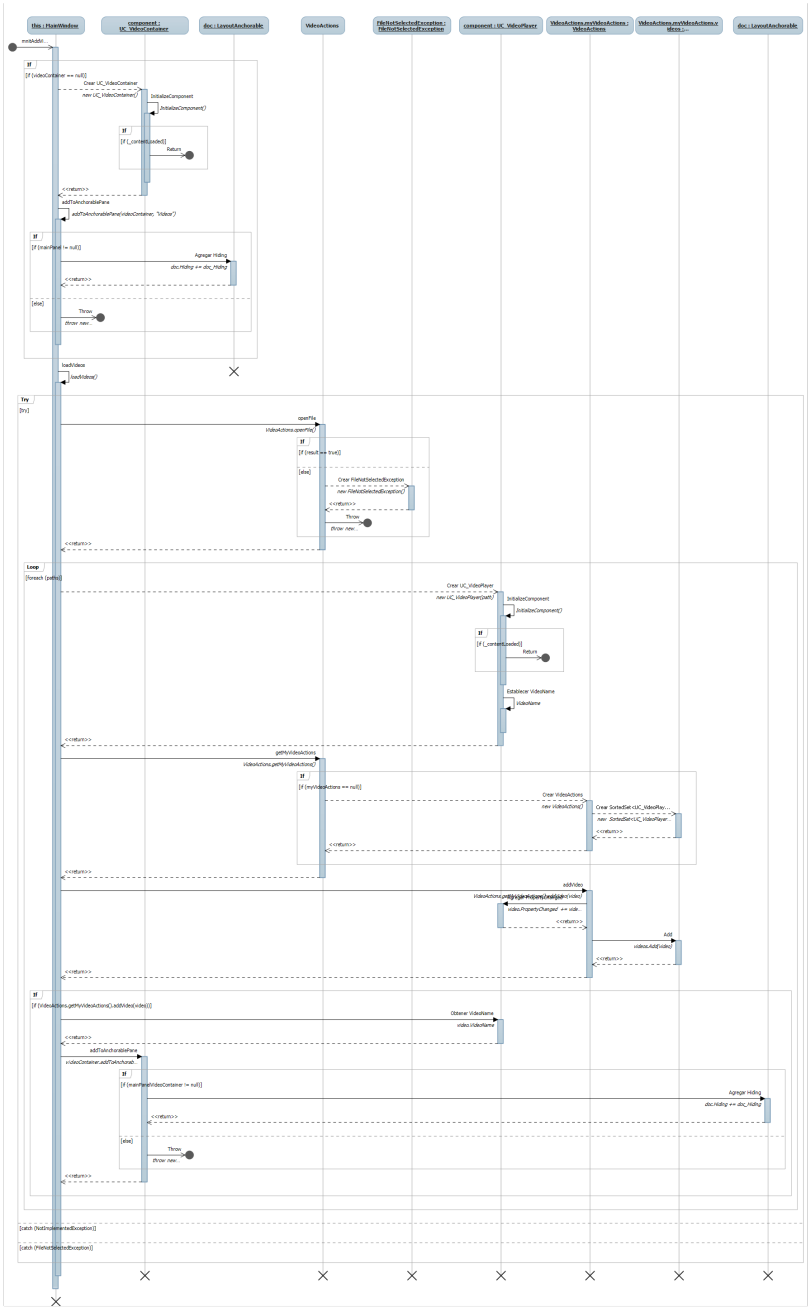


Figura C.3: Añadir vídeo

C.4. Crear intervalo

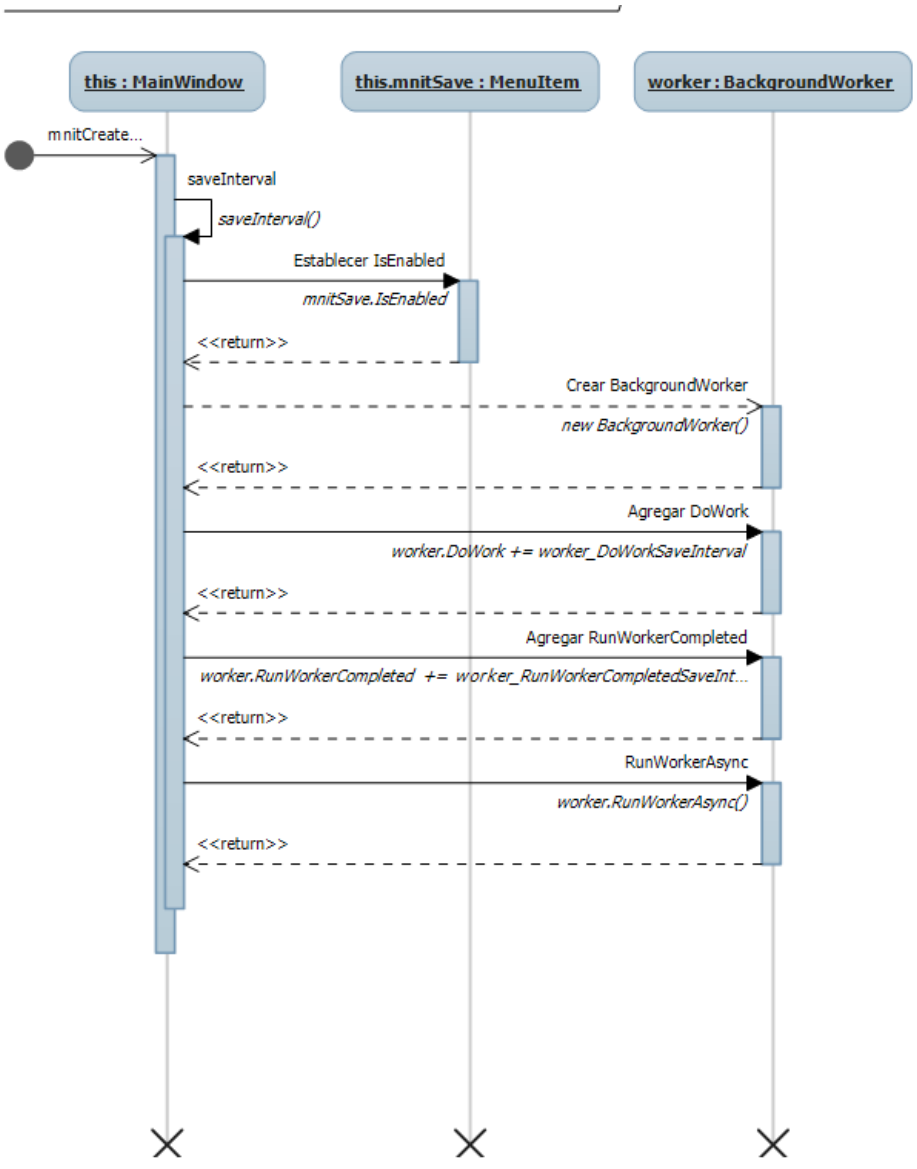


Figura C.4: Crear intervalo

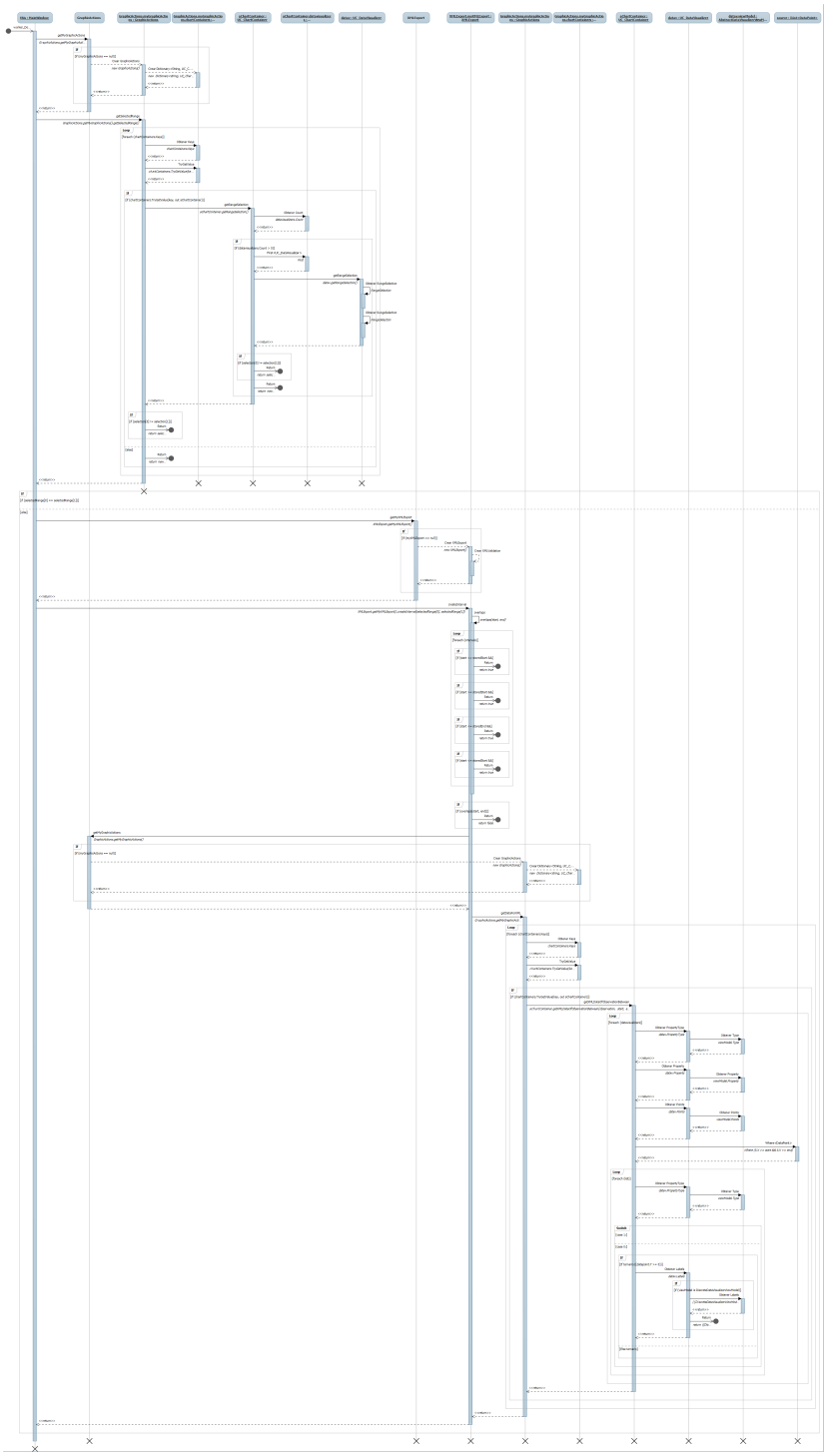


Figura C.5: Crear intervalo, parte del worker

C.5. Seleccionar rango

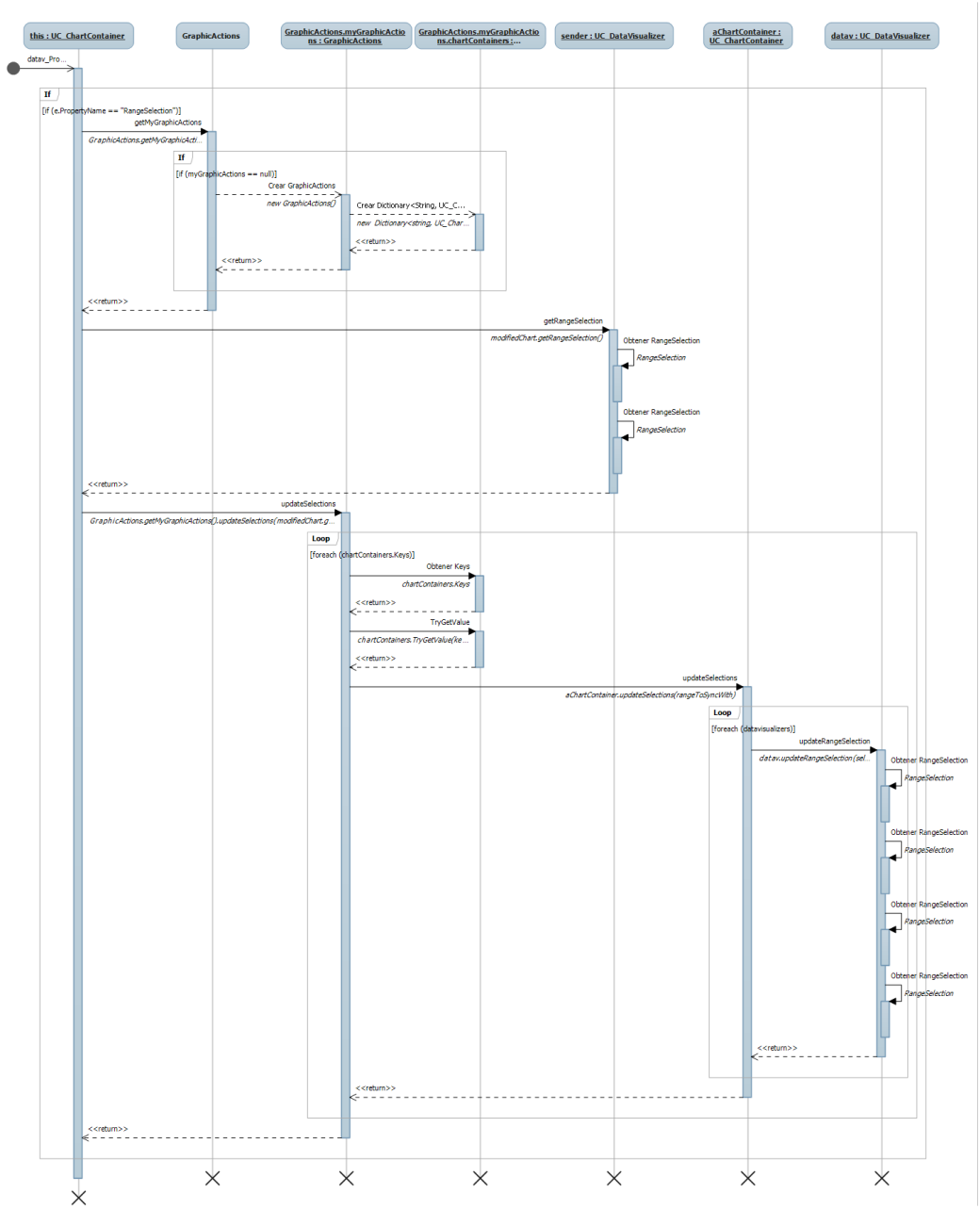


Figura C.6: Seleccionar rango

C.6. Guardar paso o situación

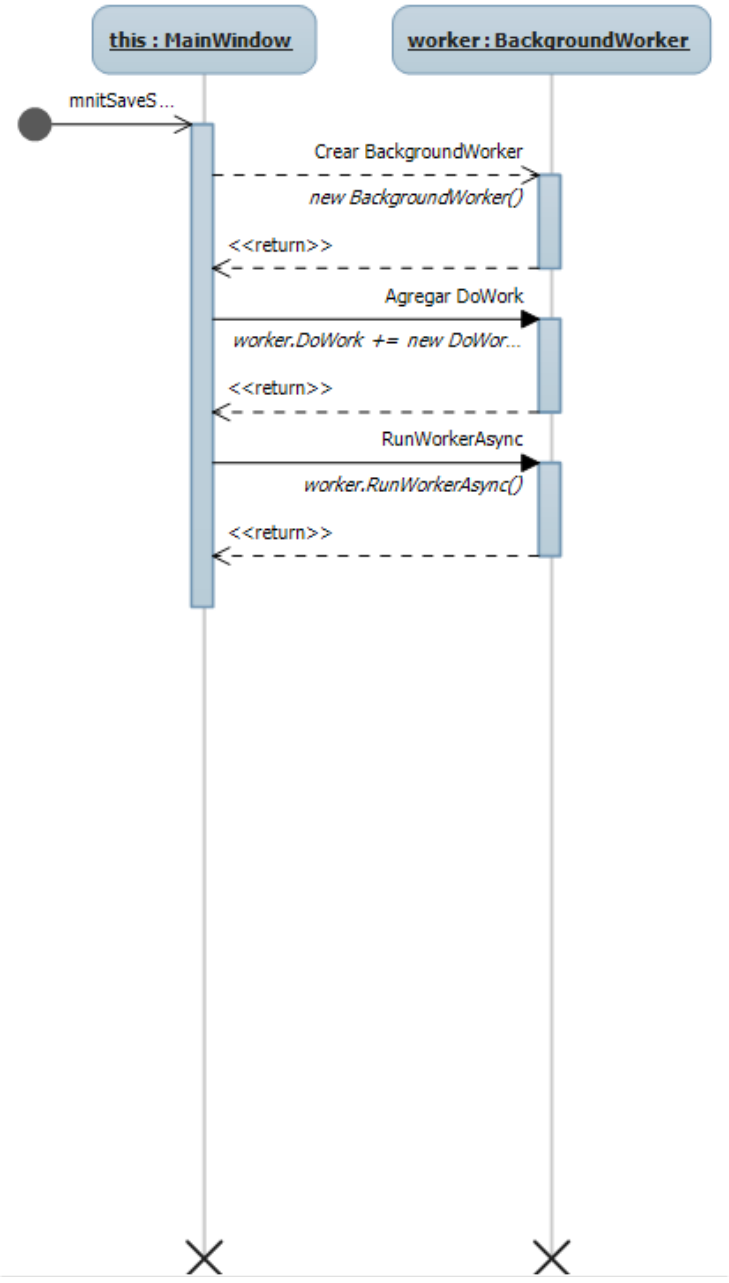


Figura C.7: Guardar paso o situación

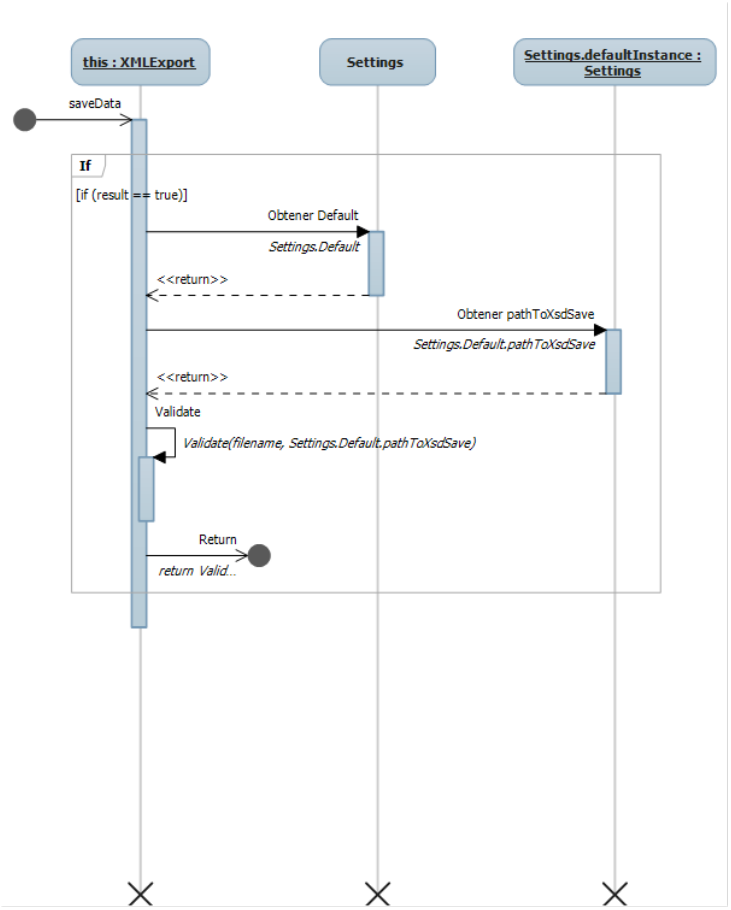


Figura C.8: Guardar paso o situación, parte worker