Sistema de copia de seguridad automática para un programa de diseño técnico

Memoria del proyecto final de carrera correspondiente a los estudios de Ingeniería Superior en Informática presentado por Adrián Serrano Reina y dirigido por Diego Javier Mostaccio Mancini.

Bellaterra, 6 de Septiembre de 2009

El firmante, Diego Javier Mostaccio Mancini, profesor del Departamento de Arquitectura de Computadores y Sistemas Operativos

CERTIFICA:

Que la presente memoria ha sido realizada bajo su dirección por Adrián Serrano Reina

Bellaterra, 6 de Septiembre de 2009

Firmado:

El firmante, Pascual Castellón Gadea, vicepresidente de la empresa CADT Software and Drafting, S.L.

CERTIFICA:

Que el trabajo al que corresponde esta memoria ha sido realizado bajo su supervisión mediante convenio 12/3/2009 firmado con la Universidad Autónoma de Barcelona.

Así mismo, la empresa tiene conocimiento y da su visto bueno al contenido que se detalla en esta memoria.

Bellaterra, 6 de Septiembre de 2009

Firmado:

A María José,
por su apoyo incondicional
y su paciencia infinita
ante mis contínuas distracciones.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo se presenta una descripción del entorno del proyecto, los problemas y funcionalidades que se desean corregir, así como una introducción a la solución adoptada.

1.1. Proyecto en empresa

El proyecto se desarrolla en el ámbito de la empresa CADT Software and Drafting, S. L. , dedicada al desarrollo de software para la industria textil. Gracias a una experiencia de 5 años en la empresa y a los conocimientos adquiridos durante la carrera se puede afrontar un proyecto de estas características con confianza e ilusión. Más allá de las mejoras en el software que se persiguen con este proyecto, se pretende iniciar con él la transición hacia el uso de prácticas estandarizadas de diseño, desarrollo y test; así como el cambio gradual de paradigma de programación, pasando de una programación puramente procedural en lenguaje C a la orientación a objetos mediante C++.

1.1.1. Historia

La empresa CADT Software and Drafting, S.L. nació en 1987 con el objetivo de desarrollar software para la entonces creciente industria del encaje. Desde sus inicios, CADT se ha centrado en el desarrollo de software para máquinas de tipo *Raschel*, sector dominado por la empresa germana Karl Mayer Textilmaschinenfabrik, GmbH., fabricante de dichas máquinas. Debido a esto, CADT se ha visto obligada a competir duramente con el software desarrollado por esta empresa, el cual partía con la ventaja de ser el único software recomendado por el fabricante.

Esta situación causó que el software de CADT fuese percibido por el público como una alternativa arriesgada al software oficial. Dado el alto coste de la maquinaria (cercano al millón de euros) el cliente era reacio a utilizar desarrollos procedentes de un software alternativo y no recomendado. No fue hasta la década siguiente, cuando una conocida

empresa norteamericana confió en el producto de origen español, principalmente debido a su mayor flexibilidad y facilidad de uso respecto al software original.

Así nacía la *suite* SAPO (*Sistema Automático de Puesta en carta por Ordenador*) dispuesta en sus inicios a hacerse un hueco el mercado del encaje, hito que alcanzaba años más tarde.

1.1.2. La suite de diseño

Con el paso de los años, el sistema SAPO se ha afianzado en el mercado del encaje, siendo percibido como la más avanzada de todas las *suites* disponibles, y la preferida por los estudios de diseño con mayor reputación. Dada su gran flexibilidad y multitud de opciones, es común que los contratos de compra del software incluyan un curso de aprendizaje para los diseñadores que deberán utilizar el sistema. En ocasiones dicho curso excede las 3 semanas de duración, siendo necesarios varios meses de atención telefónica para reafirmar los conocimientos adquiridos por el aprendiz.

Esta suite de diseño incluye numerosas herramientas para asistir a los estudios de diseño en todas las tareas que necesiten realizar en su trabajo diario. Escanear los bocetos hechos a mano por un diseñador artístico, trabajar estos bocetos hasta obtener el diseño técnico, y escribir *diskettes* en formatos aceptados por el telar son los pasos comunes del desarrollo de un proyecto de diseño.

1.1.3. Presente

Actualmente, CADT provee software para la mayoría de fabricantes de encajes del planeta, siendo la única empresa con soporte para la maquinaria *Leavers*, utilizada principalmente en Francia y partes del Reino Unido. Además, cuenta con un contrato de colaboración con la empresa Karl Mayer, el cual le permite acceder en exclusividad a las especificaciones técnicas de su maquinaria.

1.2. La aplicación de diseño

La herramienta principal de la *suite* SAPO es el programa CARTA. Con este programa se realiza todo el proceso de "puesta en carta" de un diseño. Esto es, pasar del boceto o *sketch*, realizado a mano (diseño artístico), a los datos numéricos necesarios para implantar el diseño en un telar (diseño técnico). Este proceso se encuentra sujeto a múltiples limitaciones impuestas por la máquina y su configuración particular.

Anteriormente este proceso se realizaba de forma manual y podía tomar periodos de 2 a 4 meses, con múltiples pruebas en máquina, que en ocasiones podían acabar en una avería debido a algún error cometido por el diseñador. Actualmente, gracias a la aplicación

del software, se reduce este tiempo de unos pocos días a un máximo de 3 semanas. El programa CARTA es capaz de emular las limitaciones presentes en la máquina, resaltando los puntos problemáticos, y de realizar simulaciones fotorealísticas del tejido final. De esta manera se evita tener que realizar frecuentes pruebas en el telar, lo cual implica detener la producción actual y arriesgarse a averías.

1.2.1. Datos de trabajo

La información con la que trabaja el programa de diseño es muy variada. Se puede dividir en las siguientes categorías:

Configuración de máquina: Existen multitud de parámetros configurables según el tipo de máquina (más de 200 diferentes): La posición y tipo de cada pasador de hilos (entre 1 y 3000 por barra), actuadores activos, tensiones máximas, etc. Algunas máquinas trabajan con cartones perforados y es necesario describir el significado de cada agujero presente en el cartón.

Información de barras: Se trata de los hilos que tejen el diseño. Cada uno está definido por un vector de enteros unidimensional que indica la posición del hilo en cada punto. Existen entre 40 y 120 barras según el modelo de telar.

Información de jacquard: Formada por varios mapas de bits (capas), que cubren todo el diseño. Configura el comportamiento de unos hilos especiales, el *jacquard*, sobre los cuales trabajan las barras.

Capas extra: Algunas máquinas disponen de información extra conceptualmente similar al jacquard: Capa EFS, para activar un dispositivo mecánico de corte de hilos (*clipping*); capa de frontura, para tejer en dos superficies simultáneamente; etc.

La superficie cubierta por esta información está representada por dos coordenadas:

Agujas: Coordenada Y. El límite típico de la maquinaria es de 3168 agujas, pese a que un diseño no necesariamente las utiliza todas. Es posible unir dos diseños siempre que no superen el límite del telar, de manera que se confeccionen los dos en paralelo.

Pasadas: Coordenada X, solo limitada por memoria disponible en la máquina. La máquina trabaja todas las agujas al mismo tiempo, de pasada en pasada. Los diseños suelen variar entre las 100 y 5000 pasadas.

El tamaño de un proyecto pueden variar considerablemente, según sus dimensiones y las capas presentes (jacquard, EFS, etc.). A continuación se expone un pequeño resumen de los tamaños típicos:

Tipo	Descripción	Pasadas	Agujas	Tamaño
Pequeño	Pequeños trabajos de encaje	200	240	150 KB
Mediano	Trabajos con jacquard simple	500	800	1 MB
Grande	Grandes diseños con jacquard doble	1500	2000	10 MB
Enorme	Mantillas de grandes dimensiones	5000	3000	30 MB

Cuadro 1.1: Ejemplo de dimensiones de un proyecto

En la práctica el tipo de proyecto depende del producto de encaje que desarrolla el cliente (lencería, cortinas, mantillas, etc.). Realmente es difícil ofrecer un porcentaje de proyectos para cada tipo enunciado, pues dependiendo de la maquinaria de cada cliente y el tipo de producto que desarrolla, puede que la mayoría de sus proyectos sean de tamaño pequeño y mediano, o puede que todos ellos sean de gran tamaño.

1.2.2. Problemas

Existen ciertos problemas en el software que llevan a los usuarios del programa a la pérdida de datos. Algunos de estos problemas se pueden mitigar, otros están fuera de nuestro control. Es necesario asumir que los programadores, como seres humanos, siempre pueden cometer errores. Por tanto, hay que contar siempre con la posibilidad de que aparezcan fallos del software y encontrar la manera de reducir su impacto.

Fallos del software

A medida que un software crece, adquiere más y más funcionalidades, sufriendo también múltiples reestructuraciones internas. Además, se trabaja sobre código en algunos casos pobremente documentado y realizado por programadores que abandonaron la empresa hace años. A esto hay que añadir ciertos problemas específicos de este software:

- **Gran tamaño**, estando compuesto de más de 500 mil líneas de código C, librerías compartidas incluidas. Carta creció en el último año en 40 mil líneas y se modificaron más de 80 mil.
- **Dificultad de test**: Debido a su gran flexibilidad, es imposible comprobar los cambios sobre todas las combinaciones posibles, ni prever los usos que va a dar el cliente a las nuevas características.

- **Secreto industrial**: En muchos casos el cliente tiene sus propios trucos y técnicas que forman parte de su secreto industrial y que no desea revelar. Es posible que cambios en apariencia inocuos causen conflictos con estas técnicas y lleven al programa a un fallo general.

Fallos de alimentación eléctrica

No es ni mucho menos despreciable la incidencia de los fallos eléctricos en la pérdida de datos. A los problemas causados por la cada vez más antigua red eléctrica de los países industrializados, hay que sumar los continuos problemas de subidas y bajadas de tensión en la precaria red eléctrica de los países en desarrollo, donde la deslocalización está llevando cada vez a más empresas.

Pobres hábitos de trabajo

En muchos casos el principal enemigo de los usuarios son ellos mismos. La elevada concentración mientras realizan su trabajo, junto con el miedo a guardar para darse cuenta más tarde de que han cometido un error y no pueden volver atrás, acaba resultando en unos periodos muy largos sin salvaguardar el trabajo realizado.

Para ilustrar estas afirmaciones, se ha medido la frecuencia con la que usuarios habituales del programa guardan su trabajo. El resultado de estas mediciones se encuentra en las figuras 1.1 y 1.2. Se muestra una barra vertical cada vez que el usuario ha guardado su trabajo. La longitud de la barra indica el número de operaciones sin guardar en ese momento[1]. Salta a la vista como algunos usuarios trabajan por periodos superiores a una hora sin guardar su trabajo, llegando a realizar un centenar de operaciones durante este tiempo.

Está claro que no se puede obligar a los usuarios a guardar su trabajo, pero sí sería necesario encontrar algún método para salvaguardar los datos del usuario frente a fallos, evitables o no, de manera que el usuario pueda seguir trabajando sin necesidad de sobrescribir el fichero original ni de mantener numerosas versiones de un mismo proyecto.

Operación deshacer limitada

Otro de los problemas que lleva al usuario a la pérdida de datos es la rudimentaria implementación de la operación DESHACER. Esta operación es la que permite anular las últimas ediciones realizadas en caso de error o disconformidad con el resultado. Se puede encontrar una descripción detallada de esta implementación y sus desventajas en el capítulo ESTADO DEL ARTE, sección 2.2.

[1] El número de cambios calculado por el programa Carta es en algunos casos una sobreestimación, ya que interpreta algunas operaciones individuales como un conjunto de operaciones.

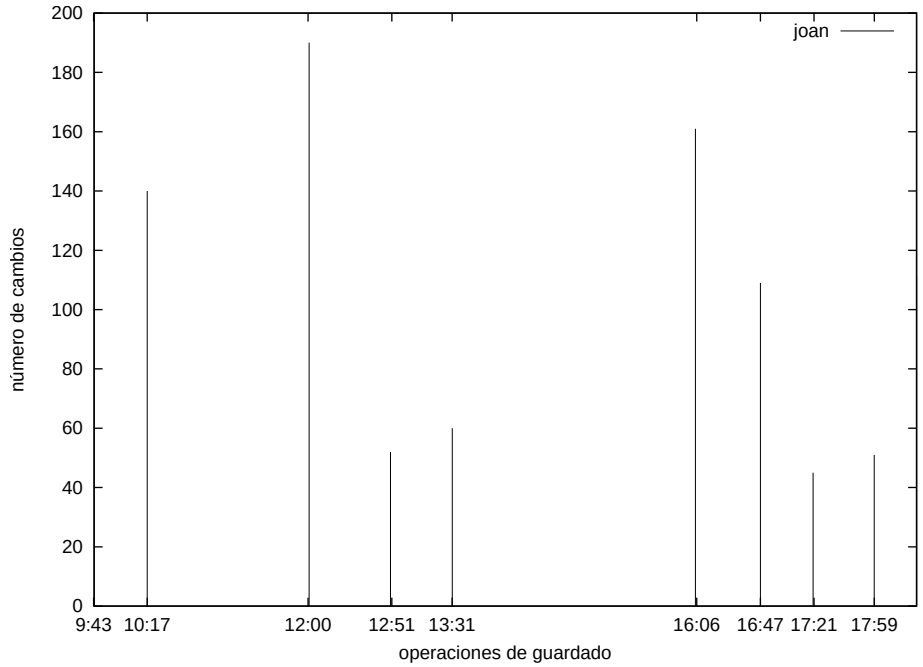

Figura 1.1: Frecuencia de guardado (usuario 1)

1.3. Propuesta de proyecto

Una vez visto el problema y sus causas, se propone como solución el siguiente proyecto: Un *sistema incremental de copia de seguridad automática*. Cada componente de este largo nombre tiene su razón de ser: «Copia de seguridad automática» porque se desea salvaguardar los datos sin necesidad de ningún tipo de intervención por parte del usuario. «Incremental» porque no solo se desea mantener almacenados los datos en el último punto de guardado, sino que se desea poder recuperar el estado anterior a cada cambio realizado por el usuario, hasta un máximo de cambios que se estudiará.

De esta manera se mantiene almacenado el proyecto en su último estado, por si un fallo de software, hardware o de alimentación evita que el usuario guarde su trabajo. Además, se ofrece la posibilidad al usuario de revertir voluntariamente el estado, mejorando la rudimentaria implementación de DESHACER vista anteriormente.

Como funcionalidad adicional, se desea estudiar, y si es posible implementar, el almacenamiento de un historial completo del proyecto desde su inicio hasta el momento actual. De esta manera se podría revertir el estado a un punto anterior en el desarrollo, con el objetivo de iniciar un nuevo proyecto a partir de ese punto o de estudiar la evolución del proyecto por motivos de aprendizaje. Se entiende que dicho historial no podrá mantenerse con una precisión de cada cambio realizado. Se deberá estudiar un mecanismo para reducir la granularidad de los cambios con el paso del tiempo, con el objetivo de reducir el uso de recursos. Esta funcionalidad está inspirada en el software Apple Time Machine, descrito en el apartado 2.3.2.

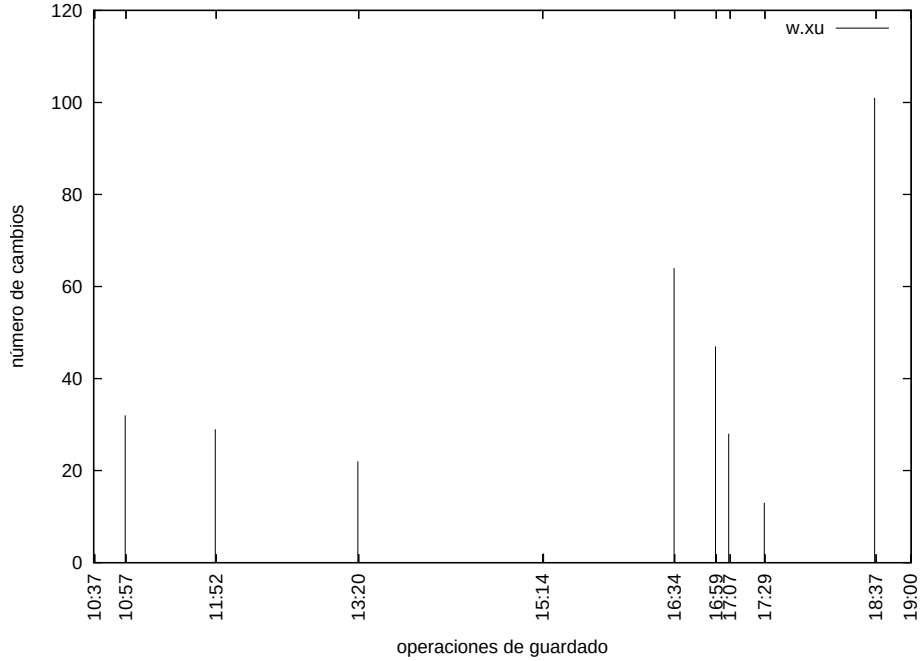

Figura 1.2: Frecuencia de guardado (usuario 2)

1.3.1. Objetivos

Una vez descrita la propuesta de proyecto, se pueden analizar los objetivos concretos que este debe cumplir. Son los siguientes:

- Almacenamiento incremental del estado del proyecto después de cada operación.
- Cálculo eficiente de las diferencias, en lugar de almacenar una copia completa de los datos después de cada cambio.
- Capacidad de recuperar los datos hasta el último estado almacenado.
- Posibilidad de revertir cambios, hasta un máximo configurable.

Como objetivos secundarios, se estudiarán los siguientes puntos. En caso de ser factibles se implementarán si la carga de trabajo lo permite, o se postergarán para desarrollar en el futuro:

- Almacenamiento de un historial completo de copias del proyecto desde su inicio.
- Posibilidad de detectar los cambios en la memoria de trabajo automáticamente.

1.3.2. Requisitos y limitaciones

Además de los objetivos globales del proyecto, este ha de cumplir con unos requisitos y limitaciones concretas para que el resultado sea aceptable y se pueda integrar con éxito en el programa de diseño:

1. **Librería reutilizable:** El grueso del proyecto se ha de llevar a cabo en forma de librería, de manera que sea fácilmente aplicable a otros proyectos de la compañía. Esto implica por un lado el uso de una interfaz de programación sencilla, y por otro lado se debe evitar incluir peculiaridades del programa de diseño.
2. **Cambios en el programa:** Se intentará minimizar al máximo posible las modificaciones en el código del programa con tal de adaptarlo a la librería. Es especialmente importante que no sea necesario realizar cambios en la librería cada vez que se modifique la estructura interna del programa Carta.
3. **Uso de recursos:** El proyecto no debe incrementar los requisitos de recursos del programa de diseño. Por suerte estos son bastante generosos. Se verá una descripción detallada en el siguiente apartado.
4. **Aceptación del usuario:** No se debe introducir ningún artefacto en el funcionamiento fluido del programa. El usuario no aceptará que existan retrasos sensibles después de realizar una operación.
5. **Licencias de código:** En caso de utilizarse librerías externas para desarrollar funcionalidades del proyecto, éstas han de tener una licencia que permita su inclusión en aplicaciones de código cerrado. Por tanto solo se admitirán librerías licenciadas bajo BSD, LGPL o similares.

1.3.3. Entorno de ejecución

El entorno de ejecución para el proyecto es el mismo que el del programa de diseño Carta. La mayoría de clientes del software han actualizado sus sistemas en los últimos años, por lo que la configuración mínima es similar a la siguiente:

- CPU Pentium IV a una velocidad superior a 1.5 GHz
- Mínimo de 1GB de memoria RAM
- Tarjeta gráfica NVIDIA de altas prestaciones (*Quadro FX*)
- Decenas o cientos de GB de almacenamiento secundario disponibles. El trabajo finalizado siempre se traslada a un servidor de almacenamiento o a cintas DAT de copia de seguridad, por lo que la mayoría del espacio está disponible.
- Sistema Operativo *Red Hat Linux* o compatible (*Fedora*, *CentOS*, etc.) con Kernel 2.4(70 %) o 2.6 en los sistemas más recientes (30 %).

- Entorno gráfico *CDE* (*Common Desktop Environment*).

Solo unos pocos clientes siguen utilizando workstations HP adquiridas entre 1994 y 1999. Un buen ejemplo es la HP Visualize Workstation 9000 Series 700 (712/100), de Junio de 1995:

- CPU HP-PA a 100 MHz
- 80 MB de RAM
- Disco duro de 4 GB
- Sistema Operativo HP-UX 10.20

Pese a que se sigue ofreciendo soporte y nuevas versiones para estos sistemas, está claro que no se puede garantizar el mismo rendimiento que en sistemas más modernos. Aun así, se pretende que todas las nuevas funcionalidades del software continúen funcionando en estos sistemas. Por tanto se necesita que la solución implementada por este proyecto funcione en estos sistemas, pero no es necesario garantizar el mismo rendimiento.

1.3.4. Planificación temporal

Se ha planificado el proyecto para ser desarrollado en 4 meses, dedicando 4 horas al día de Lunes a Viernes. Así se obtiene un margen de 2 meses para poder amortizar posibles retrasos en el desarrollo, o para dedicar tiempo a otros proyectos de la empresa. Se ha dividido el proyecto en cuatro partes principales: Análisis y diseño, implementación, test y documentación. Durante el análisis, se estudian las posibles alternativas para cada funcionalidad. Esta fase se encuentra detallada en el capítulo 3. La fase de implementación se ha dividido en tres hitos, donde cada uno ofrece una funcionalidad concreta del software:

1. Se desarrolla la estructura principal de la librería, junto con la capacidad de almacenar el estado de la aplicación periódicamente y la funcionalidad de restaurar al último estado en caso de fallo en la aplicación.
2. Se almacenan todas las operaciones en forma de diferencias en disco y se desarrolla el interfaz para que el usuario pueda deshacer operaciones.
3. Se almacena un histórico completo desde el inicio del proyecto. permite obtener el proyecto en un punto anterior del desarrollo.

Finalmente tienen lugar los apartados de test y documentación. El primero implica instalar una versión *beta* a un grupo de usuarios y corregir los posibles errores que se detecten. En el apartado de Documentación, se desarrolla la documentación de usuario y la presente memoria.

La planificación inicial se encuentra en la figura 1.3.

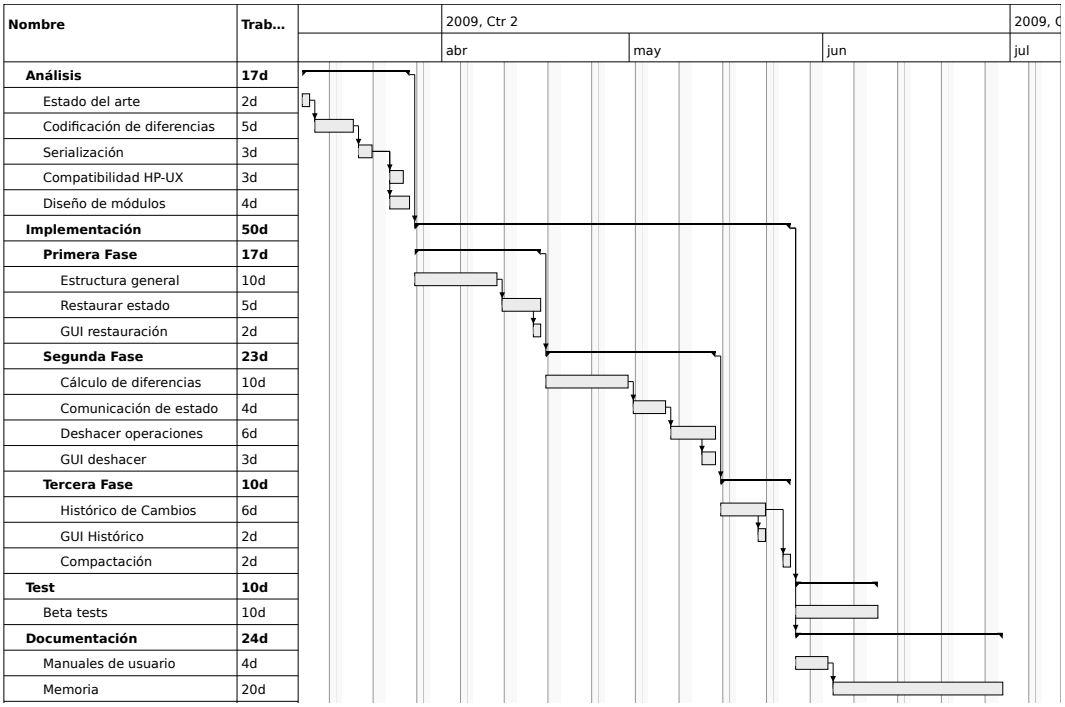

Figura 1.3: Diagrama de Gantt

Capítulo 2

Estado del arte

En este capítulo se van a estudiar las técnicas y herramientas de uso común en la industria para solucionar los problemas planteados, así como una descripción de las soluciones implementadas actualmente en el programa Carta y que se pretenden mejorar.

2.1. Recuperación de datos

Existen multitud de técnicas mediante las cuales una aplicación puede salvar su estado en almacenamiento secundario con el objetivo de resumir su ejecución en caso de fallo. A continuación se estudiarán las ventajas y desventajas de cada una.

2.1.1. Guardado automático

La técnica más sencilla, y también la más común en aplicaciones de escritorio. Aprovecha la propia funcionalidad de la aplicación para guardar el trabajo del usuario a un fichero, creando un fichero temporal (generalmente oculto) con los datos actuales y eliminándolo al finalizar la aplicación correctamente. Si al iniciar la aplicación se detecta la presencia de uno de estos ficheros, se asume que no finalizó correctamente y se ofrece al usuario la posibilidad de recuperar sus datos y continuar trabajando con ellos.

La principal diferencia entre implementaciones yace en la lógica que decide cuándo se debe almacenar la información: Cada cierto número de operaciones realizadas por el usuario; antes de realizar operaciones complejas, pues son más propensas a fallos; o bien pasado un intervalo de tiempo concreto. Cabe destacar que esta no es una clasificación exhaustiva. En muchos casos se utiliza una solución híbrida o bien se permite al usuario configurar esta funcionalidad según sus preferencias.

Como ejemplo de programas de uso común utilizando esta funcionalidad, está la *suite* de ofimática Microsoft Office, que almacena los cambios cada 10 minutos por defecto (Microsoft, 2007). El editor de texto VIM, de uso mayoritario en sistemas UNIX, mantiene

en todo momento una copia de los ficheros en edición, que se actualiza con cada cambio por pequeño que sea. Generalmente el usuario puede recuperar el estado completo de la edición después de un fallo (Moolenaar, 2003). En los últimos años esta funcionalidad está siendo aplicada cada vez más a transacciones *online*, utilizando la exitosa tecnología AJAX, de manera que los datos quedan almacenados temporalmente en el servidor remoto (Cioroinau, 2007). Esta técnica es aplicada en el conocido gestor de contenidos web WordPress.

Esta solución cuenta con dos desventajas importantes. La primera es que, salvo honrosas excepciones, no protege al usuario frente a la pérdida de sus datos más recientes, pues es posible que no almacene al instante cada nuevo cambio realizado, llegando incluso a almacenar tras diez minutos de trabajo. La segunda desventaja es que en ocasiones esta técnica resulta molesta e intrusiva para el usuario, pues debe esperar a que se realice la operación de guardado para continuar trabajando. Si se activa cuando el usuario desea continuar editando, o el retraso introducido es excesivo, causará frustración en el usuario y pérdida de productividad. Esta problemática ha de ser tenida en cuenta desde el diseño de la funcionalidad, en caso contrario se corre el riesgo de que el usuario no tolere las desventajas y desactive la funcionalidad.

2.1.2. *Checkpoint and restart*

Pese a que en esencia puede parecer un concepto similar al anterior y hereda todas sus desventajas, las técnicas de *checkpoint and restart*, también conocidas como de checkpoint periódico, ofrecen una funcionalidad muy superior. Trabajando típicamente a nivel de *kernel* del sistema operativo, permiten almacenar el estado completo de la aplicación (*checkpoint*) y no sólo los datos que el programador de la aplicación ha previsto mediante su rutina de guardado. Dependiendo de la implementación, el estado que se almacena en memoria secundaria puede ir desde todas las páginas de memoria reservadas por la aplicación, hasta gran parte de los recursos utilizados por ésta: Manejadores de ficheros, hilos de ejecución, recursos bloqueados (*locks)*, etc. Esto permite al usuario continuar la ejecución del programa (*restart*) en el punto exacto en que se encontraba al momento de realizar el checkpoint.

Se puede encontrar un buen ejemplo de implementación del mecanismo *Checkpoint and Restart* en la librería Berkeley Lab Checkpoint/Restart (BLCR), disponible para el sistema operativo Linux. La tabla 2.1 resume los diferentes recursos de la aplicación que BLCR es capaz de almacenar.

Esta solución sin embargo cuenta con cierto número de desventajas. En este aspecto su principal ventaja introduce a su vez un gran inconveniente frente al método anteriormente estudiado. Guardar todos los datos en memoria causa un retraso temporal considerable, mientras que en el caso anterior, al tratarse de un almacenamiento selectivo, este retraso es menor. Cabe destacar que algunas implementaciones de la solución *Checkpoint-Restart* utilizan una optimización para reducir este retraso: duplican el proceso (*fork*) y almacenan la copia duplicada mientras el original continua su ejecución. El uso de esta optimización

Característica	Implementada	Planeada	No se soportará
Procesos con un único hilo	X		
Procesos multi-hilo	X		
Grupos de procesos		X	
Sesiones		X	
User-checkpoint hooks	X		
Manejadores de señal	X		
Reiniciar llamadas al sistema	X		
Procesos monitorizados (ptrace)			?
Identificador de proceso	X		
Identificador de usuario y grupo		X	
Limites de recursos		X	
Información de uso de recursos		X	
Ficheros regulares		X	
stdin/stdout/stderr	X		
Tuberías (con y sin nombre)		X	
Sockets TCP y UDP			X
Ficheros mapeados en memoria	X		
Regiones de memoria compartidas		X	
Dispositivos de bloque		Parcial	
Ficheros en /proc		Parcial	
Ficheros bloqueados		X	
E/S asíncrona			?
Memoria fija (*pinned*)		X	
Memoria protegida	X		
System-V IPC			?
Mutexes/condiciones pthread	X		

Cuadro 2.1: Características soportadas por BLCR (Duell *et al.*, 2002)

no es tan sencillo como pudiera parecer, causando conflictos con las aplicaciones multihilo y la estrategia de *copy-on-write* del sistema operativo (Roman, 2002).

Aunque fuese posible mitigar este problema con optimizaciones o mejorando el ancho de banda del sistema, *Checkpoint-Restart* sigue siendo una solución que no puede aplicarse todavía en aplicaciones de escritorio, como la aplicación en la que se enmarca este proyecto: Los recursos relacionados con el entorno gráfico y otros recursos de uso común en aplicaciones de uso general, tales como conexiones de red e intercomunicación entre procesos (IPC), no están soportados y es posible que no lo estén nunca. No hay que olvidar que estas librerías están destinadas a plataformas de computación de altas prestaciones y entornos científicos, donde la interacción con el usuario es mínima, y la comunicación entre procesos se realiza mediante librerías especializadas, como MPI, la cual sí está soportada en BLCR.

2.1.3. Solución actual

El programa Carta cuenta con la funcionalidad de guardado automático, descrita en 2.1.1, guardando el fichero temporal cada 10 cambios. Desafortunadamente, pocos diseñadores utilizan esta opción pues causa un retraso que resulta muy molesto, especialmente al trabajar con proyectos de gran tamaño.

2.2. Deshacer

La funcionalidad de deshacer, también conocida por su nombre en inglés, *undo*, permite anular las últimas operaciones realizadas sobre los datos de trabajo. Dependiendo de la implementación, se limita el número máximo de pasos a deshacer. Esta limitación suele variar entre unos pocos cambios, 3 en editor gráfico Microsoft Paint, hasta varias decenas, como los 20 cambios de Adobe Photoshop.

Se puede clasificar esta funcionalidad separando las implementaciones en dos modelos diferentes, según el nivel de funcionalidad que se desee ofrecer:

2.2.1. Deshacer lineal

Se considera que las operaciones están encadenadas, esto es, una operación depende de todas las anteriores. Por tanto, para anular una operación se han de anular también todas las operaciones posteriores. Un ejemplo de implementación de este modelo es mediante el patrón *memento* de diseño de software. Con esta técnica se representa el estado interno de la aplicación (memoria de trabajo) como un objeto (el *memento* o recuerdo). Por cada operación que realiza el usuario, se almacena una copia de este estado interno. Así, cuando el usuario desea deshacer una operación (y por tanto, todas las operaciones posteriores), se reemplaza el estado interno de la aplicación con el estado almacenado anteriormente a la operación a deshacer.

2.2.2. Deshacer no-lineal

Esta solución, más flexible y también más complicada de implementar, permite anular libremente cualquier operación manteniendo las operaciones posteriores. Es posible encontrar un ejemplo de esta técnica en (Vratislav, 2008). La representación más común es mediante el patrón de diseño de software *command*, el cual permite encapsular la información necesaria para invocar un método de la aplicación. De esta manera, se puede almacenar una lista de objetos *command* a medida que se realizan operaciones, pudiendo aplicarlos o no selectivamente para anular las operaciones no deseadas por el usuario.

2.2.3. Solución actual

El programa Carta cuenta con una implementación del tipo lineal. El funcionamiento general de esta operación en el programa Carta es el siguiente:

1. Se identifican, con mayor o menor acierto, los datos susceptibles de ser modificados por la operación.

2. Dichos datos son copiados a un buffer de memoria (prealojado para información de barras, dinámico para información de jacquard).

3. Se lleva a cabo la operación

4. El usuario puede acceder al menú Barras → Deshacer o Jacquard → Deshacer. Esto reemplazará la información modificada con la copia almacenada antes de realizar la operación.

Los problemas de esta solución son múltiples:

- No todas las operaciones se pueden deshacer. Específicamente, solo las operaciones que afectan a información de barras o mapas de bits (jacquard, capas EFS, etc.) se pueden deshacer. No están protegidas por esta funcionalidad las modificaciones sobre la configuración de la máquina o las operaciones complejas que pueden afectar a todo el diseño (ajuste de calidad[1], importación de ficheros sobre el proyecto).

- Solo es posible deshacer la última operación, no está contemplado el almacenamiento de múltiples operaciones. En algunos casos (información de barras) el programa puede acumular diversas operaciones con la misma herramienta en unos pocos segundos. En este caso, se desharán todas las operaciones como si se tratase de una única operación.

Cabe destacar que esta operación fue implementada en Carta cuando se ejecutaba en máquinas con unos recursos muy limitados: Hewlett-Packard Series 700, bajo sistemas operativos HP-UX 9.0 y 10.0, los cuales podían no exceder los 16 MB de RAM física y disponer de un espacio de paginación (*swap*) muy limitado. Por aquel entonces, era difícil encontrar una implementación más avanzada en editores gráficos, estando estas limitadas a editores de texto, pues tenían unos requisitos de memoria mucho más reducidos.

[1] el ajuste de calidad es un resampleado de la información, de manera que se modifica la resolución (en agujas y/o pasadas) del proyecto pero se mantiene un diseño similar. Por tanto se modifican todos los datos del proyecto.

2.3. Histórico de cambios

La funcionalidad de histórico de cambios se puede considerar una ampliación del método anterior. Permite mantener un historial completo de los cambios realizados sobre los datos a controlar, generalmente uno o varios archivos organizados en carpetas, sin estar limitado a los cambios realizados durante la invocación de una aplicación concreta.

Actualmente el programa Carta no dispone de una funcionalidad equivalente. Con este proyecto se busca subsanar esta carencia, pues podría aportar muchos beneficios al usuario.

2.3.1. Control de versiones

El control de versiones, también conocido como control de revisiones, permite controlar los cambios sufridos a lo largo del tiempo por un árbol de ficheros y directorios, y se utiliza típicamente en proyectos de desarrollo de software. Existen multitud de implementaciones de este concepto, siendo las más populares *SubVersion*, *GIT* y *SourceSafe*.

Estos sistemas mantienen versionado cada elemento a controlar, pudiéndose revertir su estado a uno anterior, crear diferentes ramas, que permiten evolucionar el elemento en dos o más independientes, y combinar los cambios realizados por varias personas sobre el mismo fichero.

Pese a que existen muchas más funcionalidades, interesan en este caso la posibilidad de obtener un historial de cambios y de revertir cada fichero controlado a un estado anterior. Existen aplicaciones que explotan únicamente esta funcionalidad para ofrecer un histórico detallado de los ficheros del usuario. Por ejemplo en (Hess, 2005), se detalla la implementación de esta técnica en sistemas UNIX para mantener un historial de todas las modificaciones sufridas por los documentos del usuario.

2.3.2. Apple Time Machine

Time Machine, un software de la compañía *Apple*, ofrece una interesante implementación de histórico de cambios. Se trata de un sistema de copia de seguridad incremental, donde todos los archivos y carpetas del usuario se mantienen almacenados en un dispositivo de disco externo. Cada hora, *Time Machine* crea un nuevo árbol de directorios en el directorio raíz del disco, donde guarda una copia de los ficheros que han cambiado en la última hora, y un enlace (*hard-link*) a la copia anterior para los ficheros que no se han modificado. Para esto, *Time Machine* se aprovecha de la información de "fecha de última modificación" que el sistema operativo almacena para cada fichero y directorio, evitando tener que comparar el contenido de ficheros con la copia almacenada para detectar posibles cambios.

En cualquier momento, el usuario puede "ir atrás en el tiempo", utilizando una línea temporal, y navegar por sus ficheros antiguos para recuperar un documento eliminado por error, revisar el contenido anterior de un archivo, o restaurar completamente su sistema.

Capítulo 3

Análisis y diseño

3.1. Casos de uso

Una vez vistos los requisitos y funcionalidades del proyecto en el apartado 1.3, es posible detallar la funcionalidad que se ha de implementar. Se ha recogido esta funcionalidad en forma de casos de uso. La figura 3.1 muestra cómo se relacionan dichos casos de uso para componer la funcionalidad completa del proyecto. En la tabla 3.1 se describe cada caso de forma individual.

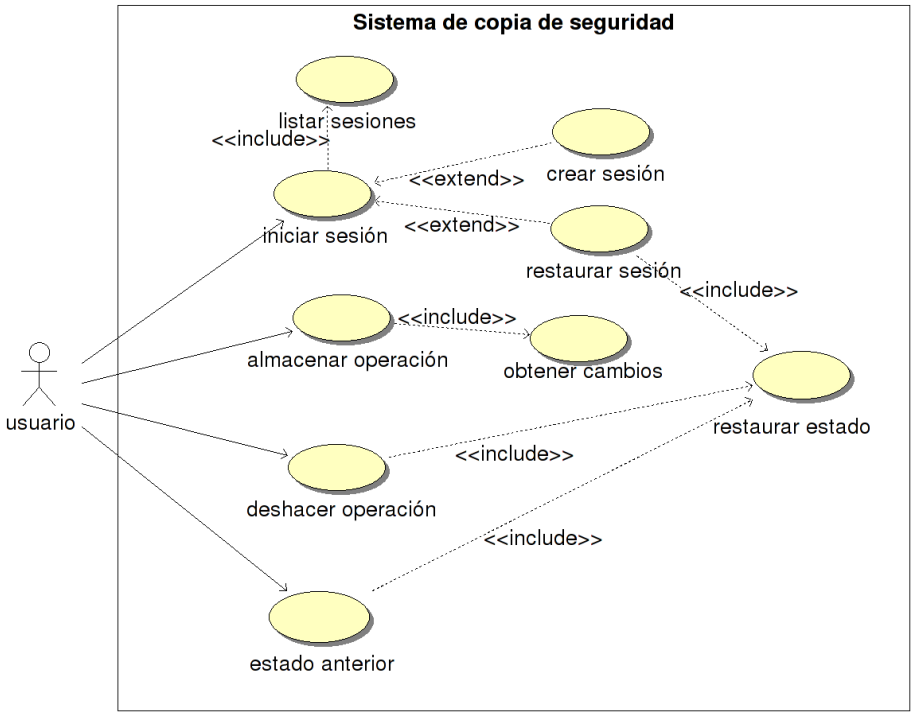

Figura 3.1: Diagrama de casos de uso

Caso de uso	Descripción
Iniciar sesión	Es el proceso que sigue el usuario para comenzar su trabajo. Implica LISTAR SESIONES y resulta en una NUEVA SESIÓN o la continuación de una sesión fallida (RESTAURAR SESIÓN).
Listar sesiones	Se comprueba si existen sesiones anteriores que se no se hayan cerrado correctamente y se muestran al usuario, para que decida si desea resumir su trabajo.
Crear sesión	Se carga un nuevo proyecto y se inicia una sesión asociada a este proyecto.
Restaurar sesión	Se carga el último estado almacenado del fichero de sesión, de manera que el usuario puede continuar su trabajo.
Almacenar operación	Después de cada operación que ejecuta el usuario sobre los datos de trabajo, se almacena en el fichero de sesión la información necesaria para devolver al proyecto al estado actual.
Obtener cambios	Se obtiene de la aplicación la información de la operación realizada y los datos modificados.
Deshacer operaciones	El usuario anula la última o últimas operaciones realizadas.
Estado anterior	El usuario decide consultar un punto anterior en el desarrollo del diseño con el que está trabajando.
Restaurar estado	Dado un fichero de sesión, se obtienen el estado o valor de los datos de trabajo en un punto determinado del proceso de edición

Cuadro 3.1: Descripción de casos de uso

Se puede observar cómo el conjunto de casos de uso cubre las funcionalidades que se desea ofrecer al usuario:

- Inicio de una sesión de copia de seguridad, creando una sesión nueva o cargando de disco una anterior.
- Almacenamiento incremental de operaciones, obteniendo los cambios realizados por cada operación.
- Anulación de una o más operaciones (deshacer).
- Restauración de un estado anterior (histórico de cambios).

3.2. Estructura general

Estudiados los casos de uso, se pasa a definir la estructura general a desarrollar. Para ello se han evaluado diferentes soluciones, que se detallarán a continuación, junto con los

problemas encontrados, los cuales han obligado a desechar soluciones que inicialmente parecían más adecuadas para la consecución de los objetivos.

La funcionalidad principal del proyecto será facilitada por un módulo que recibirá el nombre de *DataRecovery*. Este se encarga de mantener un fichero de *log* o historial donde almacena secuencialmente los datos de trabajo del programa (estado interno de la aplicación) después de cada operación realizada por por el usuario. Para la representación interna y mecanismos de obtención de estos datos de la aplicación se han estudiando diferentes soluciones, que serán mostradas en profundidad más adelante.

Tal como se ha visto en la introducción, el estado interno de la aplicación puede ser lo suficientemente grande como para no ser factible almacenar una copia completa después de cada cambio realizado por el usuario. Por esto, se ha implementado un mecanismo de diferenciación, de manera que se almacenan únicamente las diferencias entre cada par de estados consecutivos. También se estudiarán las diferentes técnicas analizadas para proveer la funcionalidad de diferenciación.

3.3. Soluciones analizadas

3.3.1. Almacenamiento de checkpoints

Al almacenarse la información en forma de diferencias, se abre la necesidad de almacenar una copia completa de los datos iniciales como base para las diferencias. Además de almacenar un checkpoint inicial, rápidamente se ve la necesidad de almacenar nuevos checkpoints cada cierto tiempo o número de operaciones. Así se evita tener que recorrer un fichero completo con miles de operaciones cuando el objetivo es restaurar el último estado. Con esto se consigue que la solución escale correctamente tras muchos días de trabajo, además de minimizar el riesgo de que daños en un punto del fichero de *log* resulten en la pérdida de todo el trabajo a partir de ese punto.

3.3.2. Fichero de log

El fichero de log debe acumular una a una las diferentes operaciones que se realizan. El formato de este fichero ha de permitir:

- Almacenar miles de operaciones, sin que se añada una sobrecarga excesiva. Por tanto es necesario un formato que haga un uso eficiente del espacio en disco.
- Almacenar otros tipos de información: También es necesario escribir checkpoints, información relativa al diseño, y puede que otros datos identificativos como el propietario, etc.

- Agregar nuevas operaciones de forma rápida. Idealmente, agregar una nueva operación debería ser tan fácil como añadirla al final del fichero, sin necesidad de actualizar otras partes del fichero.

- Ser robusto, de manera que sea posible recuperar automáticamente el fichero si por cualquier razón el sistema fallara durante el almacenamiento de una nueva operación. Esto implica que las actualizaciones de datos no deben dejar en ningún momento al fichero en un estado inconsistente.

- Ser portable entre computadores, sin necesidad de que la arquitectura bajo la cual ha creado el fichero sea compatible con la que se utilizó para crearlo.

Para cumplir con estos requisitos se ha decidido implementar un fichero con formato binario, muy compacto, compuesto por una colección de "bloques" de datos consecutivos e independientes entre ellos. De esta manera almacenar una nueva operación es tan fácil como escribir un nuevo bloque al final del fichero. Cada bloque está formado por una pequeña cabecera de bloque que contendrá al menos la información siguiente:

- El significado (tipo) del bloque (checkpoint, operación, etc.).

- El tamaño (en bytes) del bloque.

- Un checksum del bloque, para detectar bloques dañados o que han quedado a mitad de su escritura.

El formato binario estará especificado con tipos de tamaño fijo estándar del lenguaje C (y por tanto C++) como son *uint8_t*, *uint16_t*, *uint32_t* y se utilizará un orden de bytes (*endianness*) concreto para evitar conflictos entre diferentes arquitecturas de procesador.

El capítulo 4 incluye una descripción detallada del formato del fichero de *log* y de los diferentes tipos de bloques que puede contener.

3.3.3. Representación del estado interno

La funcionalidad más crítica y susceptible de ser un cuello de botella es la comunicación de los datos de trabajo entre la librería y la aplicación. Idealmente, la librería debería trabajar con bloques de memoria, fáciles de copiar de un lugar a otro y de aplicar algoritmos de diferenciación sobre ellos. Desafortunadamente, la forma en que una aplicación medianamente compleja almacena sus datos puede ser mucho más complicada: Árboles, listas enlazadas, etc. por lo que resulta necesario encontrar una forma de transformar este estado interno a un formato común.

Se han estudiado múltiples soluciones al respecto:

Acceso directo a memoria

Con esta técnica, la librería conoce la forma en que la aplicación estructura sus datos, y puede acceder a ellos directamente. La implementación típica sería mediante un *thread* o hilo de ejecución paralelo.

Pese a que sería la solución más veloz, obliga a que la librería conozca la forma en que se estructuran los datos internamente en el programa. Esta solución no es aceptable, tal como se ha visto en el apartado Requisitos y Limitaciones, pues sería necesario actualizar la librería cada vez que se modifica la estructura interna del programa.

Serialización

La solución más atractiva en un principio. Se basa en que la aplicación genera una descripción en forma de secuencia de bits de cada objeto interno que se desea preservar(serialización). Esto permite reconstruirlo (deserializar) en otro momento. Así puede comunicar a la librería el estado interno en un formato manejable (una secuencia de bits), sin necesidad de que la librería entienda su significado.

Ésta es una funcionalidad nativa de muchos lenguajes modernos, por ejemplo Java, donde prácticamente cualquier objeto puede ser serializado automáticamente por el lenguaje. Esto es gracias a las restricciones en los tipos de datos y las relaciones entre ellos que imponen los lenguajes de alto nivel. Por desgracia, lenguajes que ofrecen al programador mayor libertad para manejar y acceder a la memoria, como C o C++, complican mucho la implementación de esta técnica. Existen múltiples librerías que permiten realizar serialización de objetos en C++, por ejemplo *Boost* o *s11n*. Estas librerías se alejan del carácter automático de Java, exigiendo en la mayoría de los casos la implementación de un método para serializar y deserializar cada clase de la aplicación.

Como la aplicación que debe serializar sus datos, Carta, está programada en lenguaje C, se ha estudiado la única librería disponible para realizar esta tarea: *c11n*. Rápidamente se ha desechado esta solución, pues exige el desarrollo de métodos que describen la estructura de los datos a serializar. Esto obligaría a describir todas las estructuras internas del programa, lo cual vuelve a entrar en conflicto con las restricciones impuestas al proyecto.

Formato común

Una vez desechadas las dos ideas iniciales, se vio que la única solución factible era aprovechar el formato en el cual Carta exporta los proyectos a disco. De esta manera, no es necesario realizar grandes modificaciones en el programa, pudiendo reutilizar la funcionalidad de Guardar solo que redirigiendo la salida hacia la librería en lugar de hacia un fichero en disco.

Una vez analizado el formato en el que Carta almacena sus proyectos, se vio que este tenía ciertas propiedades que podíamos explotar, inspiradas en el formato TIFF, descrito en (Adobe, 1993). Se trata de un fichero formado por una colección de vectores de datos, llamados *tags*.

Un proyecto del programa Carta está compuesto por más de 300 *tags*, cada uno de ellos definido por:

- Identificador numérico, o etiqueta, único en el fichero. El programa utiliza un identificador diferente para cada dato o variable del proyecto.
- Tipo de datos (*byte, short, long, float*)
- Número de elementos del tipo dado.

Por tanto, se ha decidido utilizar como estado interno la colección de *tags* generada por el programa.

Esta aproximación tiene diversas ventajas respecto a trabajar con un único gran bloque de datos donde el programa guarda todo su el estado completo.

- A la hora de almacenar un nuevo estado en disco, justo después de realizar una operación, la aplicación puede comunicar a la librería solo aquellos *tags* que han sido modificados. Así se evita tener que generar todos los bloques, si se sabe que una operación concreta solo afectará a unos *tags* en particular.
- La diferenciación y compresión serán más rápidas y eficientes, pues la librería solo tendrá que diferenciar cada bloque de datos con su equivalente en el estado anterior. El caso contrario, diferenciar o comprimir un gran archivo de datos heterogéneos, exige utilizar un algoritmo de diferenciación más complicado. Más adelante se verá que esta mejora ha sido clave en el rendimiento final.

3.3.4. Cálculo de diferencias

Para almacenar eficientemente cada nuevo estado en disco, se almacenarán solamente las diferencias con el estado anterior, esto es, las diferencias *tag* a *tag*. El cálculo de estas diferencias se conoce como codificación de diferencias o codificación de deltas.

Existen numerosos algoritmos que permiten encontrar y describir las diferencias entre dos bloques de datos binarios. El resultado de estos algoritmos, que no es más que una descripción de las diferencias, no es único e incluso es posible obtener diferentes descripciones variando los parámetros del algoritmo. Esta variación acostumbra a ser un compromiso entre el tamaño de la descripción resultante y el tiempo dedicado al cálculo de ésta. Generalmente estos algoritmos están dedicados a obtener un conjunto de diferencias de tamaño muy reducido, a expensas de un tiempo de cálculo elevado. Esto no suele ser un problema,

pues se parte de la asunción de que siempre compensa encontrar una representación de tamaño más reducido en lugar de transmitir o almacenar un mayor volumen de datos. En nuestro caso, esto es un problema, pues el requisito fundamental de la aplicación es una latencia baja, tolerándose un mayor uso de disco.

Se han evaluado diversos algoritmos de diferenciación con el objetivo de encontrar uno que se adecue a las necesidades del proyecto.

3.3.4.1. RSync

RSync es una utilidad para sistemas UNIX que permite sincronizar ficheros y directorios a través de la red. Para ello utiliza un algoritmo propio de codificación de deltas, descrito en (Tridgell, 1999). A continuación se muestra una pequeña descripción del algoritmo de sincronización, donde un cliente A, que posee una copia antigua de un fichero, desea sincronizar el fichero con un servidor B, que posee una copia más nueva.

1. El cliente divide el fichero en bloques no superpuestos de tamaño S y calcula dos checksums por cada uno. Estos checksums son MD5 y una modificación de ADLER32 que lo convierte en checksum rodante[1] (*rolling checksum*).
   Una vez calculados los checksums, los transmite al emisor.

2. El servidor utiliza las propiedades del checksum rodante para encontrar rápidamente instancias de los bloques en la nueva copia del fichero. Cada acierto se verifica mediante el checksum MD5, más fiable pero más costoso. Entonces envía al cliente la información sobre como reorganizar los bloques existentes en ambas versiones del fichero así como las partes del nuevo fichero que contienen ningún bloque del fichero original.

3.3.4.2. Compresores basados en LZ77

Existen diversas implementaciones de compresores delta basados en el popular algoritmo de compresión LZ77. De entre ellas destacan XDELTA(MacDonald, 1998), ZDELTA(Trendafilov *et al.*, 2002) y VCDIFF(Korn *et al.*, 2002). Estos algoritmos explotan la capacidad de LZ77 para encontrar patrones repetidos en su entrada y reducir así la redundancia de los datos. Para ello, concatenan los datos originales (referencia) a la nueva versión de los datos (actual), y aplican el algoritmo de compresión LZ77 sin generar datos de salida hasta que no se alcanza el comienzo de la cadena de datos actual. Así, se aplica la compresión LZ77 usando los datos de referencia como ventana de contexto, y se obtiene como salida una serie de comandos de la forma:

[1] Sabiendo el checksum de un bloque de tamaño N, C_0=C(d_0..d_N), se puede calcular rápidamente el checksum del bloque desplazado una posición: C_1 = C(d_1..d_{N+1}) = F(C_0,d_0,d_{N+1}), donde la función F depende del algoritmo de checksum rodante en cuestión.

COPY: Se añaden a la salida datos presentes en el contexto. Sus argumentos son el índice dentro del contexto y el número de *bytes* a copiar.

INSERT: Se añaden nuevos datos a la salida, no presentes en el contexto. Su único argumento es la longitud de estos datos.

A continuación se presenta un pequeño ejemplo a modo ilustrativo. Se utilizarán las siguientes cadenas:

Referencia: A B C D E F G
Actual: A B C Y Z E F E F

Para generar las diferencias, se concatenan las cadenas y se LZ77 desde el comienzo de la cadena actual.

Concatenado: A B C D E F G | A B C Y Z E F E F E

Obteniendo como resultado:

Operación	Efecto
COPY 0,3	A B C D E F G | **A B C**
INSERT 2,Y Z	A B C D E F G | A B C **Y Z**
COPY 4,2	A B C D E F G | A B C Y Z **E F**
COPY 12,3	A B C D E F G | A B C Y Z E F **E F E**

Nótese que el resultado, especialmente el segundo argumento de la operación COPY, es susceptible de ser comprimido de nuevo. Por ejemplo, ZDELTA utiliza codificación Huffman en este punto para reducir aun más el tamaño.

Pese a que las tres herramientas anteriormente nombradas utilizan el mismo mecanismo para calcular los deltas, se pueden apreciar notables diferencias de prestaciones entre ellas, debido a las diferentes optimizaciones aplicadas en cada una. En (Suel and Memon, 2002) se presenta una comparativa entre XDELTA, ZDELTA y VCDIFF, usando como referencia las prestaciones de la utilidad de compresión GZIP. Tanto la comparativa de tamaño resultante como la de tiempo de ejecución están medidas respecto al parecido entre los ficheros de origen y destino. Ésta se representa como un valor entre 0 (ninguna similitud) y 1 (ficheros idénticos). Nótese que GZIP no tiene en cuenta el fichero de referencia, por tanto sus prestaciones son independientes de la similitud.

En la figura 3.2 en la página siguiente se puede apreciar como ZDELTA y VCDIFF ofrecen un resultado similar, siendo mejor el resultado de ZDELTA, cosa que el autor achaca al uso de codificación Huffman. XDELTA ofrece un resultado muy pobre en comparación, siendo prácticamente equivalente a la compresión GZIP hasta que no se supera el 80 % de parecido. La explicación que (Suel and Memon, 2002) da a esto es que XDIFF se encuentra más enfocado a la diferenciación y no a la compresión del resultado.

Respecto al tiempo de compresión, en la figura 3.3 en la página siguiente, se puede apreciar que existe poca diferencia entre las prestaciones de cualquiera de los tres métodos, siendo todos ellos más lentos que GZIP, debido a que deben procesar más información que este.

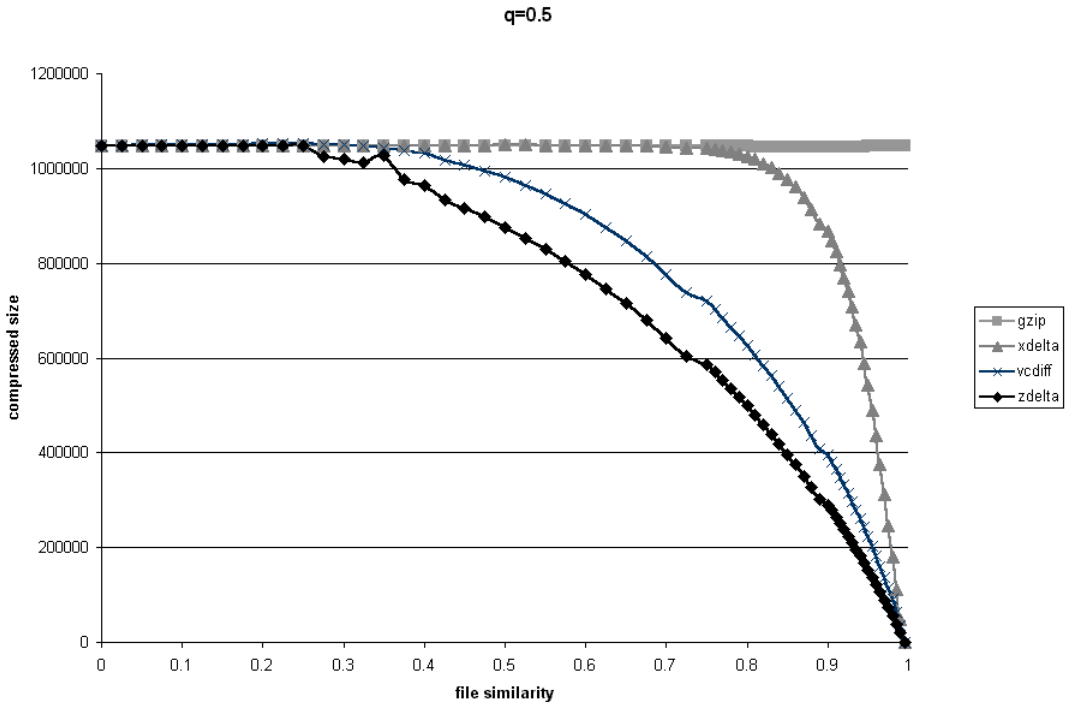

Figura 3.2: Compresión (KB) vs. similitud (Suel and Memon, 2002)

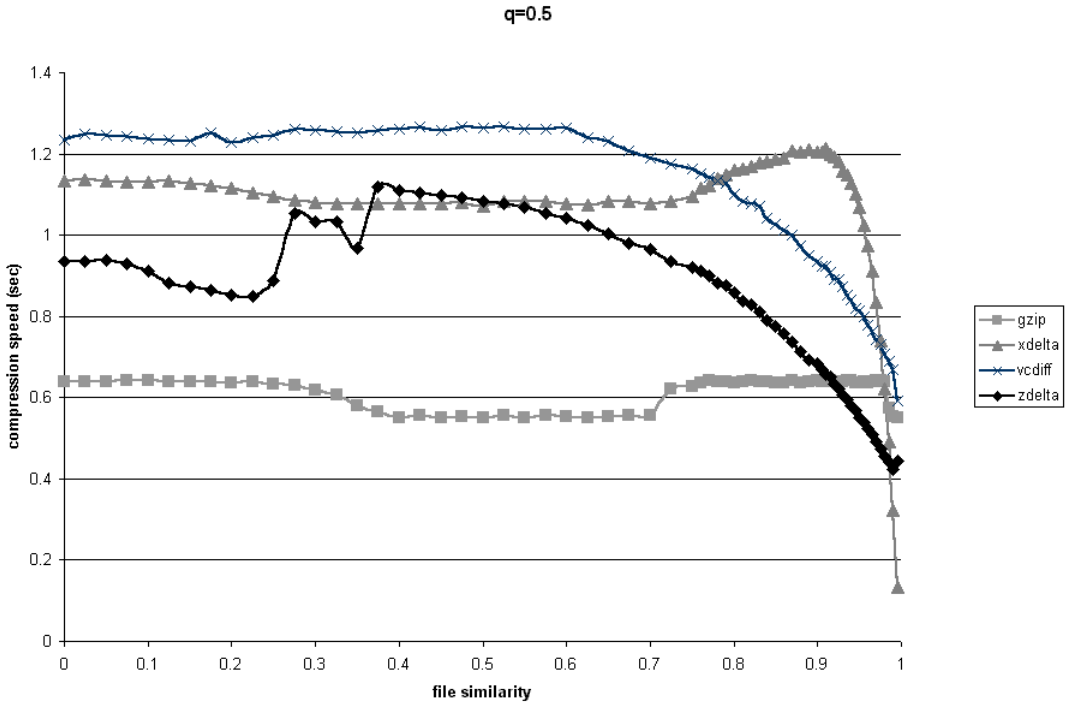

Figura 3.3: Tiempo de compresión vs. similitud (Suel and Memon, 2002)

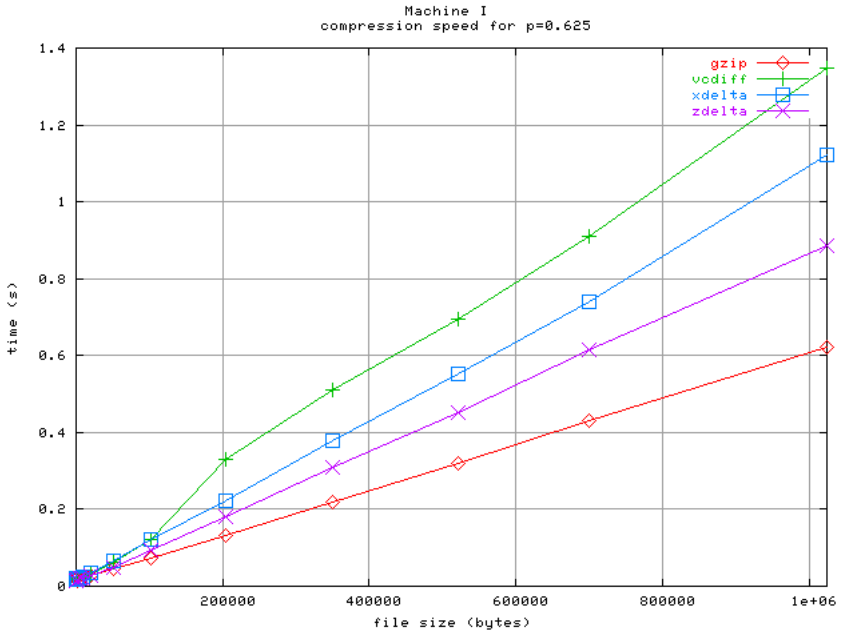

Figura 3.4: Tiempo de compresión vs. tamaño(bytes) (Suel and Memon, 2002)

La comparativa que será de mayor utilidad es la de tiempo de compresión respecto al tamaño de fichero (figura3.4), resalta que cualquiera de los algoritmos de diferenciación estudiados rinde a una velocidad cercana a 1MB/s, para dos ficheros con un 65 % de similitud.

3.3.4.3. Conclusiones

Después de estudiar estos algoritmos de diferenciación, se ha llegado a la conclusión de que no son adecuados para esta aplicación. Es necesario un algoritmo que pueda diferenciar un proyecto completo en pocas décimas de segundo. Esto implica al menos unos 50 MB/s en un computador actual. Las dos características de diseño principales de estos algoritmos son su punto débil en este caso:

- Se trata de algoritmos preparados para diferenciar datos que pueden haber sufrido muchas y muy variadas modificaciones entre la versión de referencia y la actual. En este proyecto, se parte de la ventaja de que solo se ha aplicado una operación entre ambas, y generalmente esta operación solo afecta a una pequeña parte de los datos. Esto es, la distancia de *Levelshtein* es muy reducida.

- Ambos algoritmos se esfuerzan en obtener una representación de las diferencias lo más reducida posible, a expensas de tiempo de cálculo. En esta aplicación, interesa realizar la diferenciación en un tiempo reducido, mientras que el tamaño del resultado no es tan importante, pues el espacio de disco está tan limitado.

Por estos motivos, se ha decidido implementar un algoritmo de diferenciación *ad-hoc* propio, que ha superado por mucho las expectativas. Este se encuentra descrito en profundidad en el apartado Implementación.

3.3.5. Notificación de operaciones

Una de las necesidades que queda abierta en este diseño es de qué manera se inicia todo el mecanismo que resulta en la operación escrita en disco. La solución fácil es que la propia aplicación dispare el proceso, después de realizar cada operación. Buscando minimizar al máximo los cambios en el programa, se ha estudiado la posibilidad de detectar automáticamente cuándo se había realizado una operación. Surgieron dos ideas principales al respecto:

- La librería, periódicamente, solicita el estado interno de la aplicación. Se compara este estado con la copia almacenada y en caso de encontrarse diferencias, se almacenan como un nuevo cambio.
- El programa ofrece a la librería un mecanismo para conocer si existe actividad del usuario. Esto se puede llevar a cabo monitorizando la actividad en los dispositivos de entrada del usuario[2], de tal manera que unos segundos después de detectar actividad se procede al igual que en el caso anterior.

La primera solución se descartó rápidamente poco fiable. Salta a la vista que la frecuencia con la que se realice la comparación se debía cuidar mucho, para no resultar muy intrusiva, robando constantemente ciclos de CPU al programa principal; o bien poco "fiable", encapsulando demasiadas operaciones en una sola o no detectando algunas de ellas. Si el usuario realiza una operación sobre un conjunto de datos, y seguidamente otra operación que deja estos datos en su estado original, esta solución podría no detectar ninguna modificación, impidiendo así que el usuario anule la última operación realizada.

La segunda solución tiene el inconveniente de no ser fácilmente implementable en otros programas. Carta maneja sus propios dispositivos de entrada mediante descriptores de fichero. De esta manera, puede compartir estos descriptores con la librería y que ésta se mantenga esperando hasta detectar actividad en ellos. Otras aplicaciones pueden trabajar con dispositivos de entrada diferentes, más difíciles de monitorizar, como por ejemplo los dispositivos del entorno gráfico, X11 en nuestro caso. Esta solución añadiría demasiada complejidad a la librería y sería necesario actualizarla para soportar nuevos dispositivos, lo cual es mejor evitar.

Por tanto, para no introducir complejidades innecesarias, se ha decidido delegar en el programa la responsabilidad de notificar una a una las operaciones que desea almacenar en el historial.

[2]El programa Carta lo hace mediante la llamada al sistema POSIX *select(2)*

3.4. Módulos de la librería

Una vez decidida la estructura general a seguir y solucionados los problemas encontrados, se está en disposición de detallar los módulos que formarán la librería y la forma en que se relacionan. En la figura 3.5 se puede ver el diagrama de módulos, seguido de la descripción de cada uno.

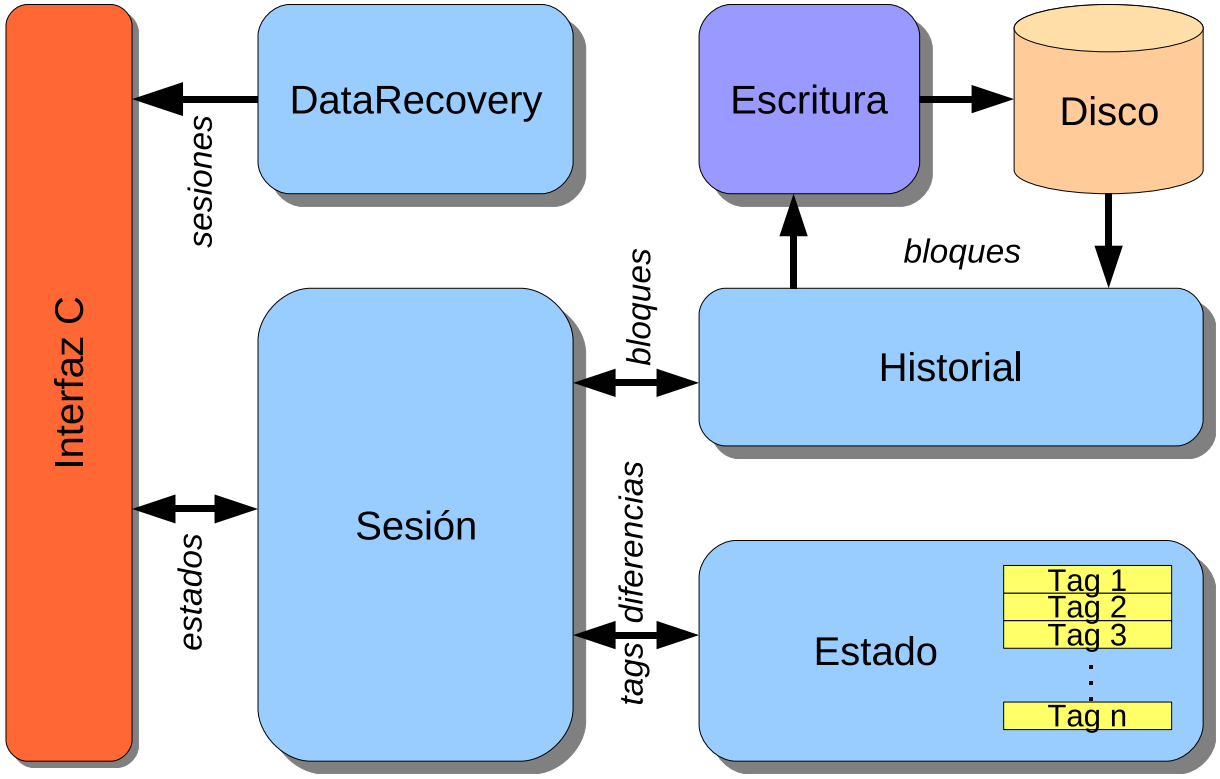

Figura 3.5: Diagrama de módulos

Interfaz C: Permite que el programa, escrito en C, se comunique con la librería, ocultando a sus objetos y métodos tras un manejador y ciertas llamadas a función que trabajan sobre él. También se encarga de gestionar las excepciones que se puedan lanzar desde los objetos de la librería y transformarlas en códigos de error.

DataRecovery: El módulo principal de la aplicación, permite crear nuevas sesiones, listar las ya existentes en disco y continuarlas.

Sesión: Gestiona todo lo relacionado con una sesión de trabajo. Recibe de la aplicación su estado completo cada vez que se realiza una operación, y también permite devolver a la aplicación un estado anterior, identificado por un instante de tiempo (historial del proyecto) o por la operación que lo ha causado (deshacer cambios).

Estado: Almacena el estado actual en forma de colección de bloques de memoria *(tags)*. Se encarga de generar el conjunto de diferencias, cuando se actualiza su estado a partir del nuevo estado de la aplicación; y de interpretar las diferencias, cuando se está cargando un estado de disco.

Historial: Hace de interfaz para el fichero de historial, trabajando con un conjunto de "entradas" de historial, que son los bloques por los cuales está compuesto dicho fichero.

Escritura: Mantiene una lista de entradas a escribir en el fichero de historial, realizando la escritura en un hilo de ejecución diferente, para que la aplicación pueda seguir trabajando.

3.4.1. Diagrama de clases

Sin necesidad de entrar en detalles, se detallan las principales clases que implementan la funcionalidad de estos módulos, la forma en que se relacionan entre ellas y sus métodos públicos más interesantes. En la figura 3.6 se pueden ver las clases que intervienen en la capa de lógica de negocio, esto es, las encargadas de gestionar todo el proceso de la información.

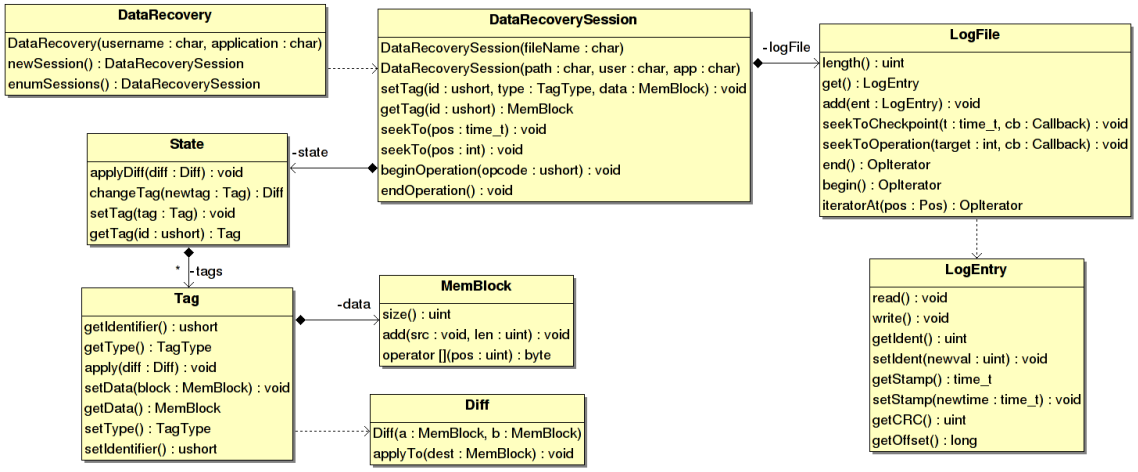

Figura 3.6: Diagrama de clases: Lógica de negocio

DataRecovery: Se trata de la clase de acceso a la librería. Identifica al usuario y aplicación que utilizan el sistema de recuperación de datos. Ofrece las funcionalidades de crear una nueva sesión y de listar las sesiones existentes para el usuario y aplicación dados.

DataRecoverySesion: Hace de intermediario entre la aplicación, el estado en memoria y el fichero de log. De cara a la aplicación, ofrece las funcionalidades de almacenar una nueva operación junto con su estado correspondiente y obtener un estado anterior, identificado por un instante de tiempo o por un número de operación.

State: Gestiona la lista de *tags* que componen el estado en memoria.

Tag: Almacena un bloque de datos, identificado por un valor numérico y su tipo de datos.

Diff: Aplica el algoritmo de diferenciación. Se crea a partir de las diferencias de dos bloques de memoria o se carga desde disco, y permite aplicar estas diferencias sobre un bloque de memoria para actualizarlo.

LogFile: Se encarga de leer y escribir entradas de log en disco, así como de desplazarse a estados concretos almacenados, devolviendo a DataRecoverySession cada entrada de log necesaria para reconstruir el estado solicitado. Además ofrece la funcionalidad de iterador para listar las operaciones almacenadas.

LogEntry: Esta clase genérica encapsula las entradas del fichero de log. Más adelante se verán todas sus subclases, que implementan las diferentes funcionalidades de cada tipo de entrada concreto.

Para la representación de las entradas de log, se ha utilizado una jerarquía de clases basada en herencia, donde las clases superiores implementan la funcionalidad más básica y a medida que bajamos en el árbol se va especializando la funcionalidad. Con esta solución, en apariencia más complicada, se evita duplicar código y permite añadir nuevas entradas de log con facilidad. La figura 3.7 muestra el diagrama de clases de esta jerarquía.

La escritura y lectura introduce ciertas complicaciones, pues es necesario calcular el *checksum* de los datos, algunas entradas se desearán comprimir, y también podría interesar en un futuro añadir diferentes tipos de compresión u otras opciones de escritura en general. Por tanto se trata de una funcionalidad que ha de crecer por separado a las propias entradas de log. Para esto se ha aplicado un patrón de diseño del software de tipo *bridge* o puente. Esto permite separar la funcionalidad de entrada/salida de la propia representación interna de las entradas, pudiendo variar ambas por separado sin problemas de escalabilidad.

LogEntry: Gestiona el tipo (identificador) de entrada de log y la fecha en que fue creada

ListEntry: Contiene una lista de variables con valor asociado, en forma de cadena de texto.

HeaderEntry: Almacena las variables que identifican el fichero de log: Nombre de usuario, programa, nombre y ruta del proyecto al que pertenece.

DataEntry: Entrada con un pequeño bloque de datos binarios.

CheckpointEntry: Identifica la cabecera de un checkpoint. Almacena el número de tags que incluye el checkpoint.

OperationEntry: Identifica una operación. Almacena el identificador de operación.

EnvelopeEntry: Entrada que está asociada a un objeto que cumple con el interfaz *Stream*. Este objeto representa un bloque de datos de gran tamaño, y permite leerlo o escribirlo directamente, sin tener que duplicar temporalmente sus datos, como sucedería con un DataEntry.

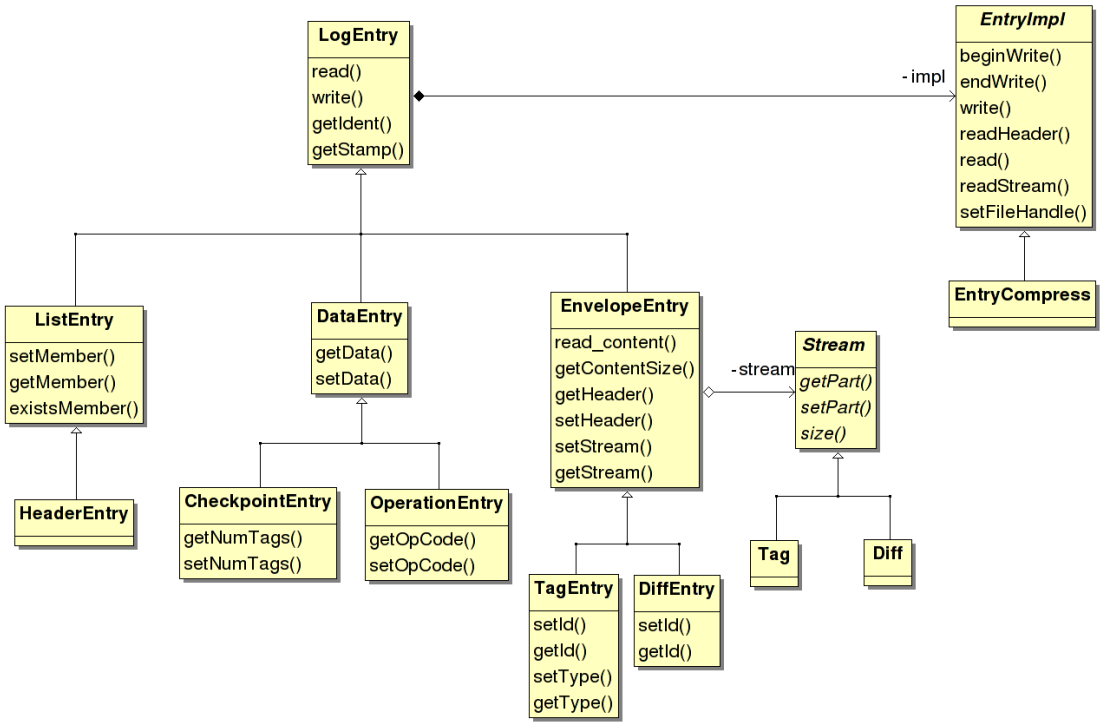

Figura 3.7: Diagrama de clases: Almacenamiento

TagEntry y DiffEntry: Entradas que representan un Tag y Diff en disco, respectivamente.

EntryImpl: Clase que ofrece la funcionalidad de leer y escribir las entradas en disco.

CompressImpl: Permite comprimir los datos antes de escribirlos a disco.

Capítulo 4

Implementación

En este capítulo se presentan los detalles concretos de la implementación del proyecto, como pueden ser los diversos algoritmos utilizados y los formatos concretos empleados para almacenar la información.

4.1. Cálculo de diferencias *ad-hoc*

Después de estudiar diversos algoritmos para la diferenciación de los datos, en el apartado 3.3.4, se ha encontrado que ninguno de estos algoritmos trabaja a una velocidad suficiente para esta aplicación. Los datos con los que ha de trabajar la librería cumplen ciertas peculiaridades que permiten obtener buenos resultados con un algoritmo mucho más sencillo:

- Los cambios entre un estado y el siguiente suelen ser mínimos.
- La operación más común es modificar una pequeña parte de los datos.

Haciendo el símil con un programa de dibujo, se ve cómo pocas veces el usuario hará uso de la operación de redimensionar el lienzo, y la mayoría de operaciones serán modificaciones locales.

Una vez vistas estas limitaciones, se ha ideado un simple algoritmo que permite representarlas de forma extremadamente rápida, a costa de ser muy inferior a los algoritmos generalistas en el resto de casos.

En la práctica, se han considerado tres casos:

1. El nuevo estado (B) tiene el mismo tamaño que el anterior(A). Se buscan las diferencias punto a punto ($A_i <> B_i$):

   En este caso se compara cada byte con el correspondiente en el bloque más nuevo,

obteniendo trazas de diferencias, que se representan mediante una lista de tripletas (*desplazamiento, longitud, nuevos datos*). Por ejemplo, dados los datos:

Estado actual (A):	1	2	3	4	5	6	7	8	9	10	11	12
Nuevo estado (B):	1	**5**	3	4	**7**	**2**	**6**	8	9	**11**	**10**	12

el resultado será

desplazamiento	longitud	datos
1	1	5
2	3	7, 2, 6
2	2	11, 10

2. El nuevo estado es más grande que el actual:

   En este caso, el algoritmo asume que el nuevo estado contiene al anterior, sin modificaciones, más nuevos datos al inicio y/o al final de este. Así, se busca una copia del bloque menor dentro del de mayor tamaño, y se almacena una copia de los nuevos datos añadidos al inicio y/o final del mismo. Ejemplo:

Estado actual:		01	02	03	04	05	06	07	08		
Nuevo estado:	**00**	01	02	03	04	05	06	07	08	**09**	**10**

   Resultado:

Longitud	Contenido
1	00
2	09,10

3. El nuevo estado es más pequeño que el actual:

   Se utiliza el mismo mecanismo que en el caso anterior, esta vez asumiendo que el estado más grande (el anterior) contiene una copia exacta del más pequeño (actual). En este caso la salida es más simple, pues no es necesario indicar el *contenido*, pues lo que interesa realmente es descartar estos valores:

Estado actual:	**00**	**01**	**02**	03	04	05	06	07	08	09	10
Nuevo estado:				03	04	05	06	07	08	09	10

   Resultado:

Longitud
3
0

4. Casos más complejos (*fallback*)

   Existen situaciones en las que el algoritmo no puede emitir un conjunto de diferencias válido. En estos casos se incluyen los nuevos datos completos, en lugar de sus diferencias. Esto sucede esporádicamente en casos concretos, por lo que el resultado global sigue siendo muy bueno en relación tamaño / tiempo empleado.

   Los casos en los que se utiliza el *fallback* son los siguientes:

   - Las diferencias entre los dos bloques de datos no pueden ser representadas mediante ninguno de los modelos anteriores. Concretamente cuando se varía el tamaño, pero el bloque más pequeño no se encuentra contenido exactamente en el más grande. Esto sucede por ejemplo cuando desde el programa se aplica una operación de "cambio de calidad", lo cual implica una reinterpretación de la información de todo el diseño.
   - Los dos bloques tienen el mismo tamaño, pero el conjunto de diferencias punto a punto es más grande que el tamaño del bloque completo. Esto sucede cuando se ha substituido el diseño por otro (Importar) o cuando se mantienen los mismos valores pero desplazados (Desplazar Barras). Estas operaciones solo se suelen dar en fases iniciales del proyecto.

4.1.1. Representación binaria

Para representar los valores *offset* y *longitud* del *script* de diferencias se han utilizado datos enteros de longitud variable (*Variable Length Integers*, o *VLQ*). Se trata de una codificación que permite almacenar números enteros en formato binario sin tener que fijar un tamaño (16 bits, 32 bits) utilizando en todo momento un número de bits reducido. Se destinan 7 bits por byte para representar el valor y el último bit (el más significativo) para indicar si se trata de los últimos 7 bits del valor. Un valor de 1 en este bit indica que el siguiente byte contiene 7 bits más, un valor 0 indica que es el último byte en la cadena. De esta manera es posible representar los valores con más o menos bytes según su tamaño:

Valor máximo	Binario	Longitud
127	01111111	1
16 383	11111111 01111111	2
2 097 151	11111111 11111111 01111111	3
536 870 911	11111111 11111111 11111111 11111111	4

Se ha limitado la longitud máxima de estos valores a 4 bytes, por tanto el cuarto byte puede utilizar todos sus bits para representar el valor. Así es posible almacenar enteros de hasta 28 bits de longitud, que permiten trabajar con tags de hasta 256 MB.

4.1.2. Prestaciones

Se ha comparado la velocidad de este método con la de los algoritmos RSYNC y XDELTA. Para ello, se han calculado diferencias sobre un fichero base de 8 MB y se ha medido el tiempo que se tarda en procesar los datos en cada uno de los escenarios descritos anteriormente.

- En la figura 4.1 se puede observar el resultado de ejecutar ambos algoritmos sobre dos bloques de datos del mismo tamaño. Las diferencias van del 0 % (bloques idénticos) hasta el 100 % (bloques completamente diferentes).

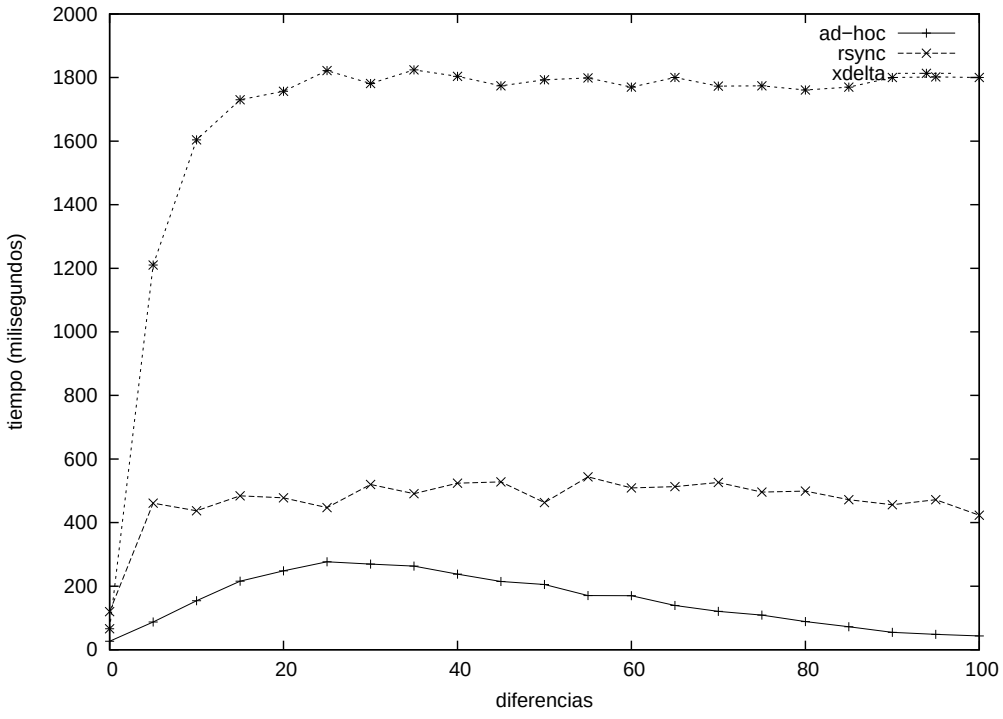

Figura 4.1: Comparativa de algoritmos delta. Caso 1

- En la figura 4.2, se ha comparado la velocidad de ambos cuando el segundo bloque de datos aumenta su tamaño, incorporando al primero en su interior. Unas diferencias del 0 % indican que ambos bloques son idénticos, mientras que el 100 % indica que el segundo bloque es 5 veces más grande que el primero.

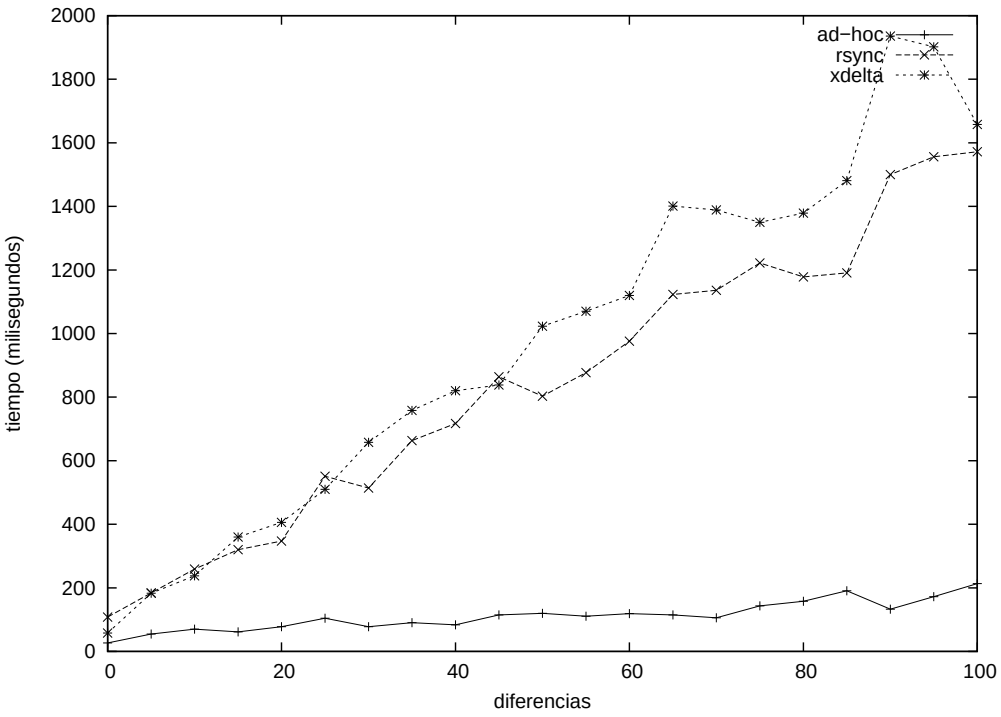

Figura 4.2: Comparativa de algoritmos delta. Caso 2

- Finalmente, en la figura 4.3, se observa el resultado cuando el segundo bloque contiene una porción del primero. Para un 50 % de diferencias, el segundo bloque contiene la mitad del bloque original. Para el 100 %, se trata únicamente de un byte.

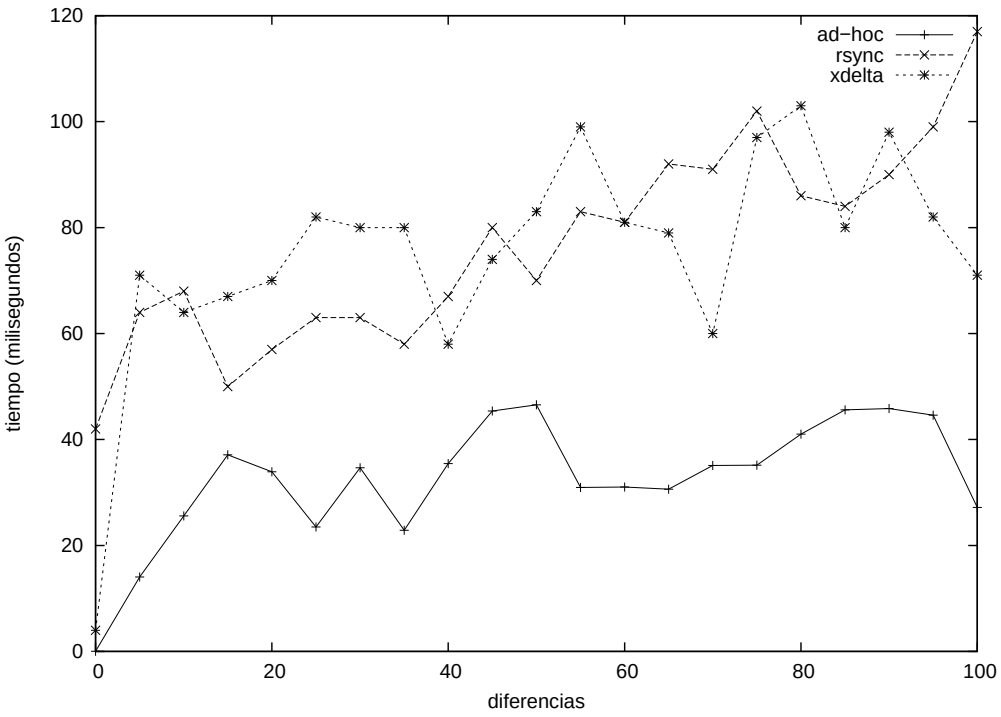

Figura 4.3: Comparativa de algoritmos delta. Caso 3

Se puede comprobar cómo este método permite procesar los datos a una velocidad hasta un orden de magnitud mayor. A costa, por supuesto, de no poder generar las diferencias en un caso diferente a los tres contemplados.

4.2. Fichero de historial

Tal como se ha descrito en el apartado 3.3.2, el fichero de *log* o historial tiene un formato binario basado en bloques. Cada bloque está formado por una cabecera con un tamaño fijo de 18 bytes:

Bytes	Tamaño	Contenido
0 a 3	32 bits	Tamaño en bytes
4	8 bits	*Flags*
5	8 bits	Tipo de bloque
6 a 9	32 bits	Fecha de escritura (UNIX *timestamp*)
10 a 13	32 bits	*checksum* (CRC32) del bloque
14 a 17	32 bits	*Offset* del bloque anterior

- El campo *flags* se utiliza para indicar diversas propiedades del bloque. Actualmente solo se utiliza un bit, que indica si el contenido del bloque se encuentra comprimido o no.
- Se almacena el *offset* del bloque anterior para permitir recorrer el fichero hacia atrás con facilidad, necesario a la hora de listar las últimas operaciones realizadas.

4.2.1. Tipos de bloque

La implementación inicial soporta 6 tipos de bloque diferentes, quedando abierta la posibilidad de añadir más bloques en un futuro si fuera necesario. Los diferentes tipos de bloque son:

Identificador	Descripción	Tamaño
1 (HEADER)	Es el primer bloque del fichero. Contiene una lista de cadenas que identifican el proyecto al que pertenece y al propietario.	Hasta 4 KB
2 (CHECKPOINT)	Marca el comienzo de un checkpoint. Indica el número de *tags* que forman el checkpoint.	2 bytes
3 (TAG)	Representa el contenido y tipo de un *tag.*	Variable
4 (OPERATION)	Indica el comienzo de una operación. Contiene el código de operación y el número de bloques de tipo DIFF que le siguen.	4 bytes
5 (DIFF)	Contiene las diferencias del *tag* anterior con el nuevo.	Variable

De esta manera, un fichero de historial está formado por:

1. Un bloque de tipo HEADER, con los datos del proyecto.
2. Un CHECKPOINT y posteriormente de las TAGs que componen el *checkpoint.*
3. Por cada operación almacenada, un bloque OPERATION seguido de tantos bloques DIFF como *tags* han sido modificadas por la operación.
4. Cada cierto número de operaciones, un nuevo CHECKPOINT.

4.2.2. Checksum de los datos

Todos los bloques del fichero de historial han de incluir un *checksum*, para poder detectar bloques dañados. Inicialmente se consideró el algoritmo Adler32 por su simplicidad y velocidad, pero se ha descartado porque no funciona bien con datos de longitud reducida (pocos centenares de bytes), de los cuales se hace un uso exhaustivo (identificadores de operación y pequeñas diferencias). Debido a esto, se ha optado por utilizar el algoritmo Crc32, el cual se recomienda en estos casos (Stone *et al.*, 2002).

4.2.3. Compresión de datos

Las entradas del fichero de historial que contienen datos de la aplicación (tipos TAG y DIFF) se almacenan comprimidas para minimizar el uso de disco. Para comprimir estos datos se ha optado por la librería Zlib. La elección de la librería ha sido en parte por la disponibilidad, ya que se está utilizando en otras módulos del programa, y por su flexibilidad, pues permite ajustar ciertos parámetros si necesitáramos mayor velocidad o mejor compresión de los datos.

El tiempo necesario para implementar y testear otras soluciones (Bzip2, Lzma, etc.) es demasiado elevado comparado con el escaso beneficio que prometen. Típicamente Zlib supera considerablemente al resto de librerías modernas de compresión en velocidad y uso de memoria, ofreciendo un resultado de compresión entre un 5 % y un 20 % inferior en la mayoría de casos (Collin, 2005). Por tanto, y dado que nuestro principal requisito prestacional es la velocidad, se decide implementar Zlib y en un futuro se estudiarán otras soluciones si la compresión ofrecida por este se mostrara insuficiente.

4.3. Pruebas

En este apartado se describen las diferentes pruebas que se han aplicado para asegurar la corrección y robustez del proyecto. Estas se dividen en dos grupos principales: Las pruebas que se realizan durante el desarrollo del proyecto y las pruebas de producción.

4.3.1. Pruebas durante el desarrollo

A medida que se desarrolla el proyecto, es necesario comprobar que el código añadido cumple con su funcionalidad, y que no se altera el comportamiento de otras partes ya desarrolladas. Para ello se realizan continuas pruebas a diferentes niveles.

Pruebas unitarias

Las pruebas a más bajo nivel que se han aplicado, comprueban cada clase de forma independiente, asegurándose de que cumplen con su contrato. Para esto se ha desarrollado un pequeño *framework* de pruebas unitarias, donde cada clase tiene una clase equivalente de test, que se ejecuta después de cada compilación del código.

En total se han realizado 21 clases de test de un total de 37 clases, pues hay clases que no se pueden probar de forma independiente. Para las pruebas se ha utilizado un mecanismo de aserciones.

Programación por contrato

Se ha seguido una metodología de programación por contrato para implementar las clases y métodos de la aplicación. Para facilitar a la detección de errores, se ha implementado un mecanismo de comprobación de *pre* y *post*-condiciones basado en macros de preprocesador, donde cada método se asegura de que se cumpla el contrato antes y después de su ejecución. Este mecanismo sólo se mantiene activo durante la fase de implementación.

Pruebas de cobertura

Para probar los módulos más importantes de la aplicación se han llevado a cabo pruebas de cobertura de código, que permiten verificar que se sigue el flujo de ejecución correcto. Para ello se ha utilizado la excelente *suite valgrind* y su herramienta *callgrind.*

Pruebas de integración

Una vez se ha comprobado que cada módulo funciona correctamente, es necesario comprobar que se integran correctamente entre ellos. Para ello se han realizado pruebas de integración, siguiendo una metodología *top-down*, donde se integran las diferentes clases que intervienen en cada funcionalidad concreta del proyecto (comunicación con la aplicación, almacenamiento, estado interno, etc.).

Siguiendo la misma metodología, se ha testeado la correcta interacción con el código que se ejecuta en un hilo paralelo, correspondiente a los módulos encuadrados en un rectángulo en la figura 4.4.

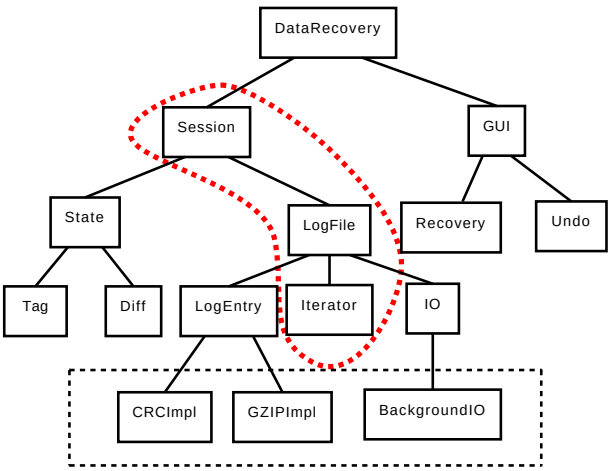

Figura 4.4: Ejemplo de pruebas de integración *top-down*

4.3.2. Pruebas de producción

Pruebas alfa

Al finalizar cada hito del desarrollo, se han llevado a cabo pruebas de producción de tipo *alfa*, donde un usuario potencial de la aplicación comprueba que la funcionalidad sea la esperada, probando cada caso de uso. Para ello se ha instalado una versión de prueba en el sistema de trabajo de un diseñador de la empresa que ha hecho las funciones de *tester* para luego comunicar los posibles problemas encontrados.

En total se han realizado dos sesiones de pruebas *alfa*, para los dos primeros hitos del proyecto:

1. Restauración ante un fallo

   Se ha utilizado una versión que almacena *checkpoints* con un periodo de 1 minuto a medida que el usuario trabaja. A los pocos minutos de trabajo, se ha forzado un fallo en la aplicación. Seguidamente se ha vuelto a ejecutar la aplicación, para comprobar que permite continuar trabajando desde el último punto almacenado.

   Con esto se ha verificado la funcionalidad de continuación ante un fallo y se ha comprobado que el almacenamiento de checkpoints en segundo plano no causa retrasos durante el trabajo del diseñador.

2. Deshacer operaciones

   Con una versión mucho más avanzada de la aplicación, se ha comprobado que el almacenamiento de cada operación individual se realiza de forma fluida e imperceptible para el usuario, y que la funcionalidad de deshacer funciona correctamente y se corresponde con las expectativas de este.

Pruebas beta

Al finalizar la fase de implementación, se ha pasado a realizar las pruebas tipo *beta*. En este caso se instala una versión completa de la librería a un grupo de diseñadores con el objetivo de que trabajen con normalidad durante unos días, utilizando cuando sea necesario las nuevas funcionalidades añadidas por la librería. De esta manera se comprueba, por un lado, que la librería actúa de la forma esperada, y por el otro, que es posible trabajar de forma fluida.

Capítulo 5

Resultados y conclusiones

Finalmente se analizan los resultados y conclusiones obtenidos a partir de las pruebas de llevadas a cabo una vez finalizado el proyecto. También se verán los apartados que quedan abiertos para implementar o mejorar en un futuro.

5.1. Rendimiento

Para poder valorar correctamente el proyecto se han realizado numerosas mediciones en un entorno de producción. Lo que interesa es conocer el rendimiento a la hora de procesar cada operación, pues es la parte más crítica y susceptible de ser un cuello de botella. En este aspecto se ha estudiado la latencia, que es el tiempo que se mantiene la aplicación bloqueada mientras se procesa una nueva operación; y el espacio de almacenamiento ocupado a medida que se acumulan operaciones.

Se han estudiado los siguientes proyectos:

	Nombre	Tamaño	Dimensiones	Operaciones	Duración
A	G09-1056	1 MB	216 x 400	131	3h 30min
B	G09-1145	2 MB	284 x 550	93	5h
C	12671-7	22 MB	672 x 5500	97	2h 10min

Las dimensiones están expresadas en agujas x pasadas.

Latencia

El tiempo de almacenamiento de cada operación viene determinada por el volumen total que ocupan los datos del diseño, constante una vez se ha dimensionado, y por la cantidad de cambios respecto al estado anterior, que dependen de la operación concreta realizada.

En las figuras 5.1, 5.2 y 5.3, se muestran las latencias de diversas operaciones consecutivas, para los tres proyectos analizados:

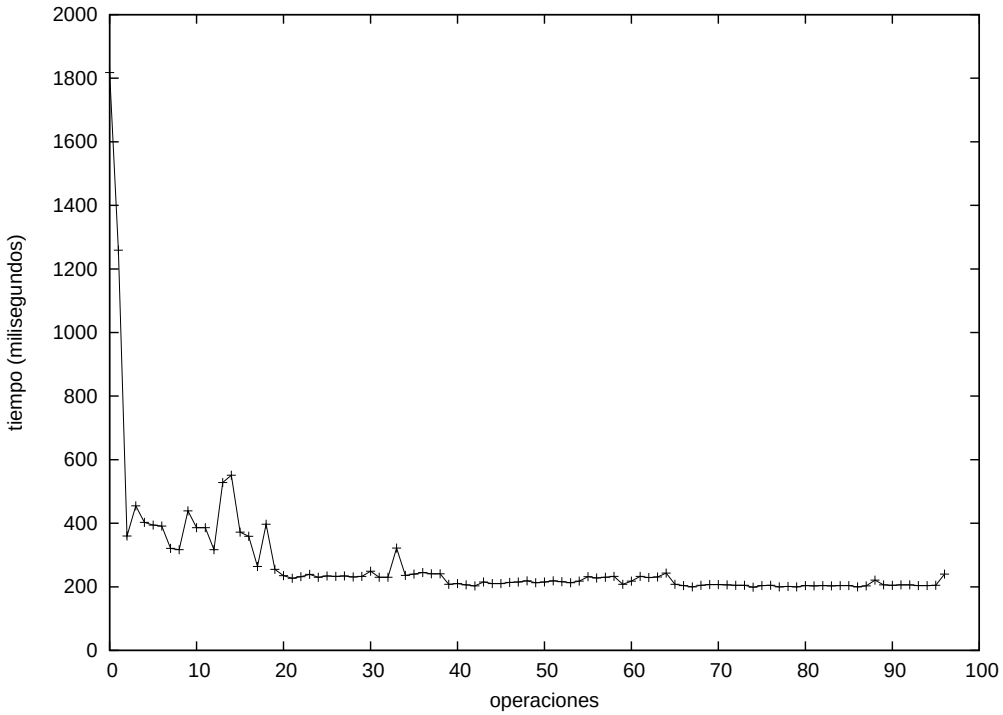

Figura 5.1: Tiempo de almacenamiento (Proyecto A)

Se puede observar como en los proyectos de tamaño medio el tiempo de almacenamiento es realmente bajo, con una media de 12 milisegundos por operación, y solo unas pocas operaciones, más complejas e infrecuentes, exceden los 20 milisegundos.

El proyecto C, de grandes dimensiones, necesita de más tiempo para almacenar las operaciones. Aun así, la mayoría de operaciones se almacenan en poco menos de 3 décimas de segundo, resultando en un tiempo promedio de 257 ms. Pese a que se trata de un tiempo elevado en comparación con proyectos más pequeños, en la práctica el usuario ha podido trabajar de forma fluida y sin tropiezos.

Espacio de almacenamiento

En las figuras 5.4, 5.5 y 5.6 se aprecia cómo aumenta el tamaño del fichero de historial a medida que se acumulan operaciones.

Para el proyecto A, de 1053 KB de tamaño original, el historial parte de una base de 40,2 KB, debido a que el primero *checkpoint* se almacena comprimido. Durante 93 operaciones

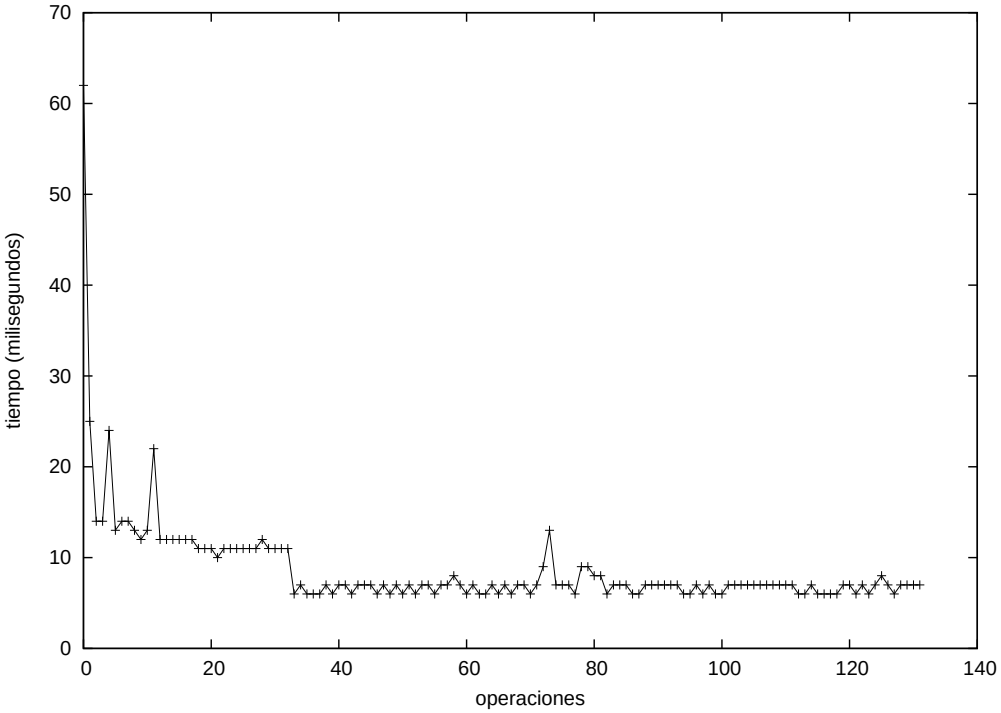

Figura 5.2: Tiempo de almacenamiento (Proyecto B)

almacenadas en forma de diferencias, el fichero crece hasta los 53,85 KB, lo que resulta en una media de 150 bytes por operación almacenada.

Para los proyectos B y C, las medias son de 527 y 739 bytes, respectivamente.

5.2. Conclusiones

Se puede afirmar que el proyecto ha cumplido con todos los objetivos principales, respetando los requisitos y limitaciones impuestos. Esto se ha llevado a cabo no sin dificultad, especialmente en lo referente a la representación del estado de la aplicación y el cálculo de las diferencias, apartados para los cuales ha sido necesario idear soluciones distintas a las que se tenían en mente en un comienzo.

Pese a estos contratiempos, el principal problema al que ha habido que enfrentarse ha sido la falta de tiempo. Esto es debido principalmente a la necesidad de dedicar más tiempo a otros proyectos de la empresa, y también a la dificultad para ajustarse a una planificación y rutina de desarrollo estricta. Esta falta de tiempo ha sido en parte culpable de que la funcionalidad de Histórico de cambios no se haya podido implementar en su totalidad. Por suerte esta funcionalidad había sido catalogada como objetivo secundario, y solo se exigía su estudio.

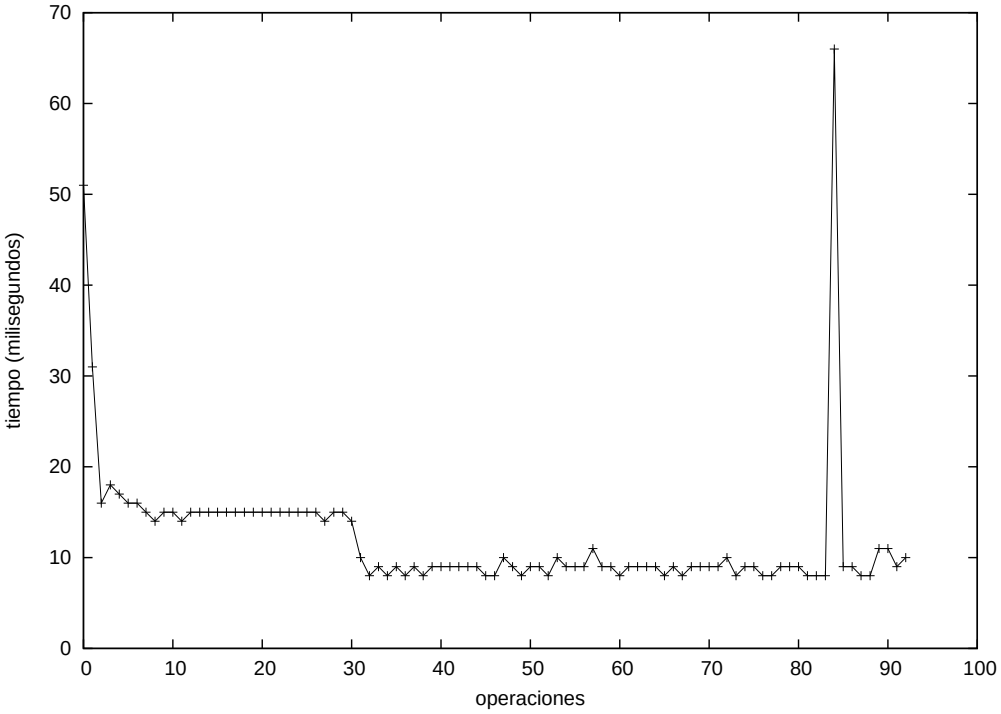

Figura 5.3: Tiempo de almacenamiento (Proyecto C)

5.3. Trabajo futuro

5.3.1. Compactación del historial

Antes de poder incluir la librería en la versión definitiva del programa, la cual se enviará a los clientes, es necesario pulir diversos aspectos de su funcionamiento. En su estado actual, se acumulan todas las operaciones desde el inicio del proyecto, además de checkpoints completos cada cierto número de operaciones. Esto no es aceptable, pues tras varios días trabajando el fichero de historial podría alcanzar un tamaño excesivo.

Para solucionar este problema, se debe incluir un mecanismo de filtrado de información en el fichero, bien por antigüedad de los datos o bien limitando el fichero a un tamaño máximo. Seguramente mediante una combinación de los dos. La idea es que, al cargar una sesión de disco, se procese el fichero de sesión y se eliminen las operaciones y/o *checkpoints* que no cumplan las condiciones de filtrado. Este filtrado tendría dos partes independientes: Por un lado, se eliminarían todas las operaciones a partir de un punto dado, dejando solo los *checkpoints*, y por otro, se eliminarían algunos checkpoints, aumentando el espacio entre ellos a medida que nos alejamos en el tiempo.

El principal motivo por el cual no se ha implementado este filtrado es la falta de tiempo para estudiar en profundidad los límites a imponer. Así, se mantendrá la librería en

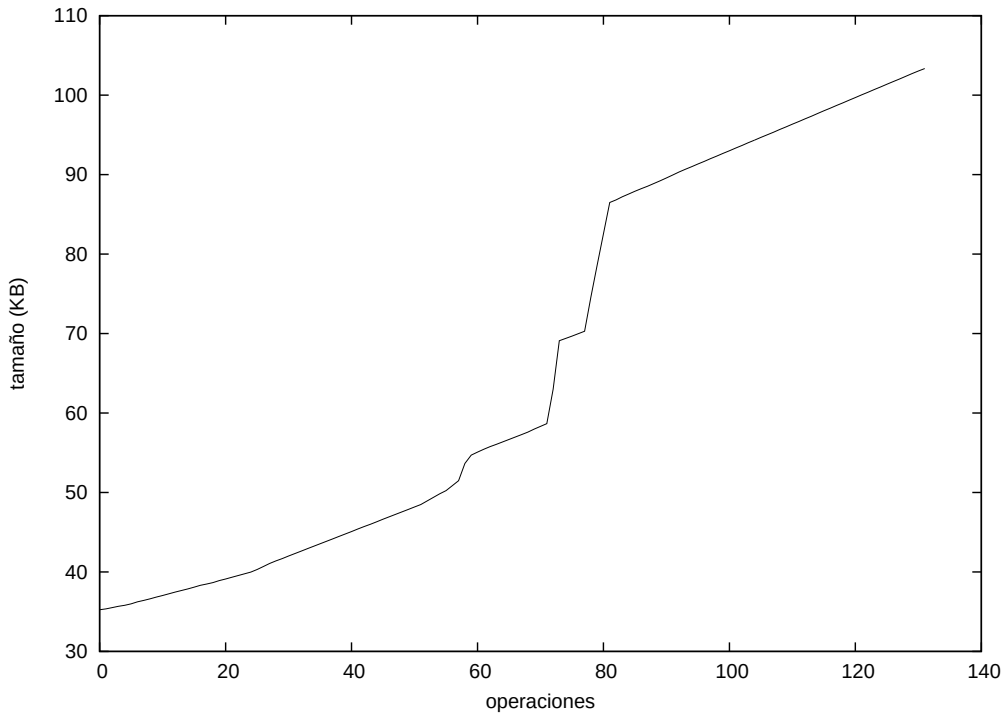

Figura 5.4: Tamaño de almacenamiento (Proyecto A)

fase pruebas durante algunos meses hasta que se encuentre la solución más acertada al problema, para lo cual es necesario haber podido inspeccionar como evolucionan los proyectos durante un tiempo extendido.

5.3.2. Historial de cambios

El objetivo de esta funcionalidad es poder ofrecer una regla de tiempo que abarque desde el inicio del proyecto hasta el presente, de manera que el usuario pueda seleccionar una fecha y hora concretas para examinar el estado del proyecto en esa fecha o crear una rama en el desarrollo a partir de ese punto.

Pese a que en la versión actual ya es posible ofrecer dicha funcionalidad, no ha sido posible desarrollar el interfaz por falta de tiempo. Además, se ha descubierto que esta funcionalidad podría ser muy útil combinada con otro proyecto de la empresa, por lo que se ha preferido detener su desarrollo para analizar en un futuro la forma de combinar ambos proyectos.

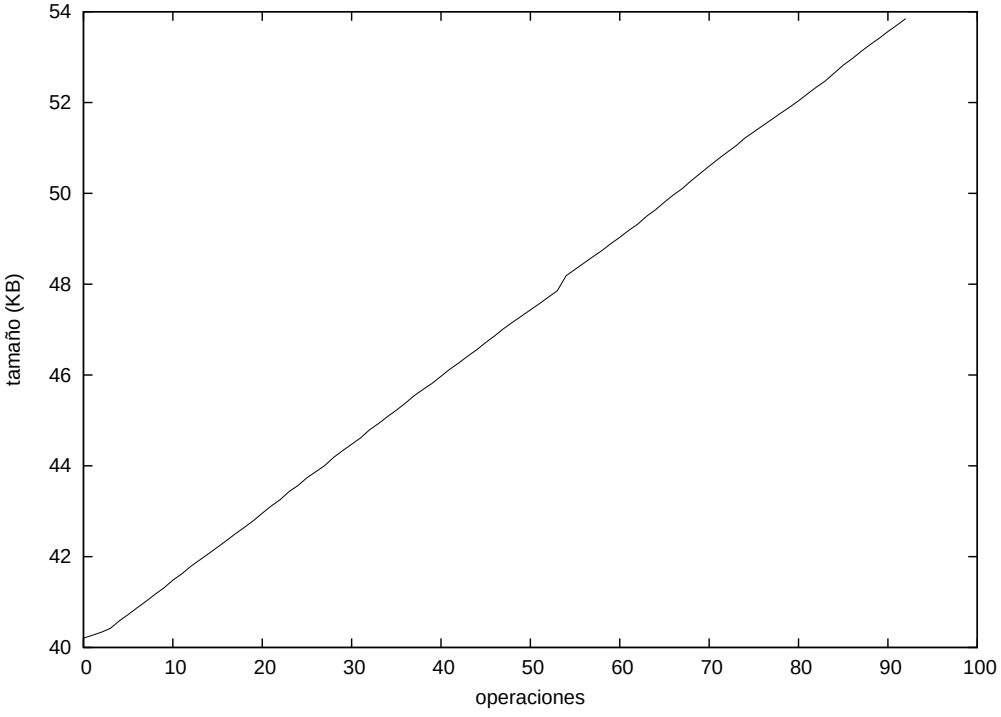

Figura 5.5: Tamaño de almacenamiento (Proyecto B)

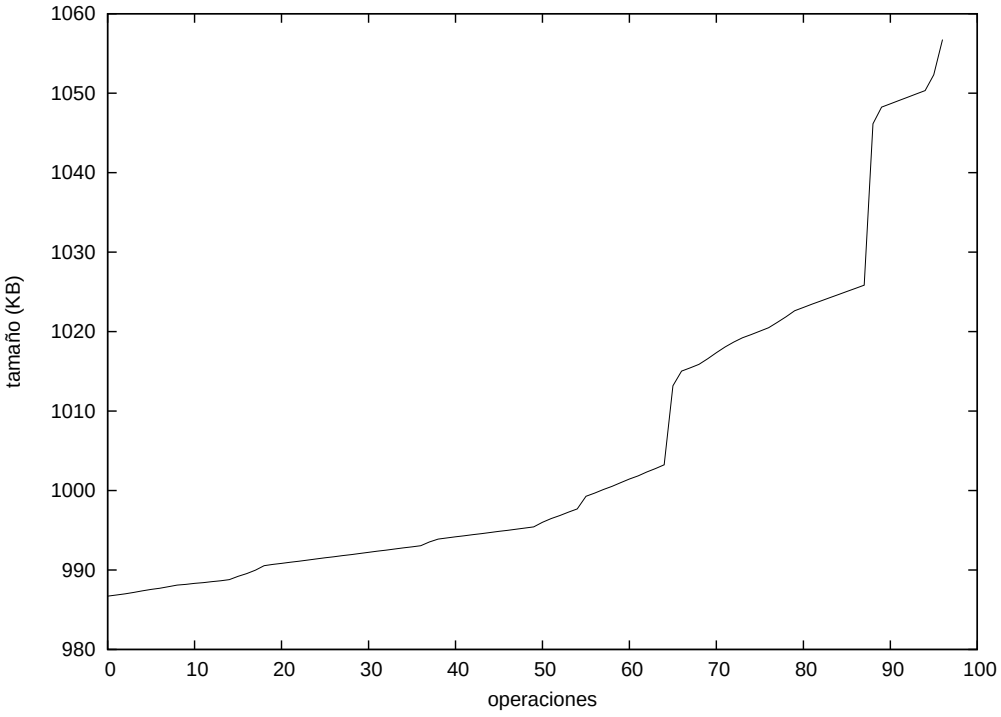

Figura 5.6: Tamaño de almacenamiento (Proyecto C)

Bibliografía

Adobe, Developers Asociation, 'Tagged Image File Format (TIFF) Specification', Tech Rep. Revision 6.0 final (1993).
URL: *http://www.adobe.com/Support/TechNotes.html*

Cioroinau, Andrei, 'Auto-save JSF forms with Ajax', *IBM developerWorks* (2007).
URL: *http://www.ibm.com/developerworks/web/library/wa-aj-jsf1.html*

Collin, Lasse, 'A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA', (2005).
URL: *http://tukaani.org/lzma/benchmarks*

Duell, J., Hargrove, P. and Roman., E., 'The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart', Tech Rep. LBNL-54941, Berkeley Lab (2002).

Hess, Joey, 'Keeping your life in SVN', (2005).
URL: *http://kitenet.net/ joey/svnhome/*

Hunt, J. W. and McIlroy, M. D., 'An Algorithm for Differential File Comparison', Tech Rep. 41, Bell Laboratories (1976).

Korn, D., MacDonald, J., Mogul, J. and Vo, K., 'The VCDIFF Generic Differencing and Compression Data3284', *Request for Comments*, no. 3284 (2002).

MacDonald, J., 'Versioned file archiving, compression and distribution', (1998).
URL: *ftp://ftp.xcf.berkeley.edu/pub/xdelta/*

Microsoft, 'How Word creates and recovers the AutoRecover files', *Microsoft Knowledge Base*, no. 107686 (2007).
URL: *http://support.microsoft.com/kb/107686/en-us/*

Moolenaar, Bram, 'VIM Reference Manual', (2003).
URL: *http://www.vim.org/htmldoc/recover.html*

Roman, Eric, 'A Survey of Checkpoint/Restart Implementations', Tech Rep. LBNL-54942, Berkeley Lab (2002).

Stone, J., Stewart, R. and Otis, D., '(RFC 3309) Stream Control Transmission Protocol (SCTP) Checksum Change', Tech Rep. 3309, Network Working Group (2002).
URL: *http://tools.ietf.org/html/rfc3309*

Suel, Torsten and Memon, Nasir, *Handbook of Lossless Compression*, chap. Algorithms for Delta Compression and Remote File Synchronization (Academic Press, 2002).

Trendafilov, Dimitre, Memon, Nasir and Suel, Torsten, 'zdelta: An Efficient Delta Compression Tool', Tech Rep. TR-CIS-2002-02, Polytechnic University (2002).

Tridgell, Andrew, *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, Australian National University (1999).

Vratislav, Jan, *Cascading undo control*, Master's thesis, Czech Technical University (2008).

Firmado: Adrián Serrano Reina
Bellaterra, 6 de Septiembre de 2009

Resum

Partint de les mancances quant a salvaguarda de dades presents en un programa de disseny tècnic, es du a terme un estudi de les tècniques necessàries per a solucionar aquestes i es detalla la seva posterior implementació en forma de llibreria. Així, s'aconsegueix oferir les funcionalitats de recuperació de dades, anul · lació d'operacions i històric de canvis, sense afectar de forma apreciable a les prestacions del programari.

Resumen

Partiendo de las carencias en cuanto a salvaguarda de datos presentes en un programa de diseño técnico, se lleva a cabo un estudio de las técnicas necesarias para solucionar estas carencias y se detalla su posterior implementación en forma de librería. Así, se consigue ofrecer las funcionalidades de recuperación de datos, anulación de operaciones e histórico de cambios, sin afectar de forma apreciable a las prestaciones de la aplicación.

Abstract

Given the insufficient data safeguard measures present in a technical design program, a study and its later implementation are carried out in order to fix these lacks. In this way, the data recovery, operation undoing and change history features are implemented without a noticeable impact on the application performance.