View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Diposit Digital de Documents de la UAB

Universitat Autònoma
de Barcelona

Aplicación de la ingeniería del software sobre la herramienta MATE: Common y DMLib

Memoria del proyecto

de Ingeniería Técnica en

Informática de Gestión

realizado por

Noel De Martín Fernandez

y dirigido por

Eduardo César Galobardes

Escola d'Enginyeria

Sabadell, Septiembre de 2011

Agradecimientos

Para empezar me gustaría agradecer la realización de este proyecto a Eduardo César y Anna Sikora. Gracias a los dos por brindarme la oportunidad de formar parte de este proyecto que ha resultado muy interesante y fructífero. También agradecerles la tutorización del progreso del proyecto y toda la ayuda prestada durante éste. Además especial agradecimiento a Anna por crear MATE, un programa que encuentro muy interesante y que espero que siga evolucionando el resto de los años.

Seguidamente le doy las gracias a Joan Piedrafita, por haber-nos dado acceso a las instalaciones y proporcionarnos un buen entorno de trabajo. También por haberme enseñado mucho, no solo en cuanto a este proyecto, sino en diferentes aspectos que me ayudarán en mi vida profesional.

Finalmente me gustaría agradecerle el apoyo y la amistad mostrados durante estos años a mis compañeros, Toni Pimenta y Rodrigo Echeverría. Os considero unos compañeros de trabajo y amigos geniales con los que no solo trabajo bien, sino que me divierto, cosa que aprecio mucho. Espero que el hecho de acabar la carrera no nos separe y sigamonos viendo el resto de los años.

FULL DE RESUM – PROJECTE FI DE CARRERA DE L'ESCOLA D'ENGINYERIA

Títol del projecte: Aplicación de la ingeniería del software sobre la herramienta MATE: Common y DMLib

Autor: Noel De Martín Fernández

Data: Septembre de 2011

Tutor: Eduardo César Galobardes

Titulació: Enginyeria Tècnica d'Informàtica de Gestió

Paraules clau

- Català: Arquitectura Paral·lela, Enginyeria del Software, Control de Qualitat.
- Castellà: Arquitecturas paralelas, Ingeniería del Software, Control de Calidad.
- Anglès: Parallel Architecture, Software Engineering, Quality Control.

Resum del projecte

- Català: Aquest projecte intenta implantar una metodologia de treball sobre MATE. MATE es una eina de sintonització d'aplicacions paral·leles sorgida de la tesis doctoral d'Anna Sikora a 2003. Vistos els resultats obtinguts, es va decidir donar un pas endavant i convertir-la en un producte software Open Source. Per fer-ho ha sigut necessari aplicar una serie d'estàndards i fer un proces de tests. En aquest treball s'ha creat part de la metodologia i s'han modificat dos dels mòduls principals.

- Castellà: Este proyecto trata de crear y implantar una metodología de trabajo sobre MATE. MATE es una herramienta de sintonización de aplicaciones paralelas surgida de la tesis doctoral de Anna Sikora en 2003. Vistos los resultados obtenidos con la aplicación, se decidió dar un paso adelante y convertirla en un producto software Open Source. Para ello ha sido necesario aplicar ciertos estándares y realizar un proceso de tests. En este trabajo se ha creado parte de la metodología y se han modificado dos de los módulos principales.

- Anglès: This project tries to create and establish a work methodology on MATE. MATE is a tuning tool for parallel applications born from Anna Sikora's doctoral thesis in 2003. After contemplating the application's results, it was decided to transform it into an Open Source software product. For that it's been necessary to apply certain standards and perform some tests. On this project part of the methodology has been created and two main modules have been modified.

Índice

1. Introducción...1

1.1 Antecedentes...1

1.1.1 Arquitecturas paralelas..1

1.1.2 Aplicaciones paralelas...2

1.1.3 Análisis de rendimiento...3

1.1.4 Sintonización de procesos..3

1.2 Sobre MATE...5

1.3 Objetivos del proyecto...5

1.4 Estructura del documento..7

2. Viabilidad y Plan del proyecto..8

2.1 Estudio de la situación actual...8

2.1.1 Contexto...8

2.1.2 Lógica del sistema..9

2.1.3 Descripción física...13

2.1.4 Diagnóstico del sistema..14

2.2 Requisitos funcionales y no funcionales..14

2.2.1 Requisitos funcionales...15

2.2.2 Requisitos no funcionales...15

2.3 Alternativas y selección de la solución...16

2.3.1 Alternativa 1 (H. Colaboración – Redmine)..16

2.3.4 Alternativa 2 (H. Colaboración – Trac)..16

2.3.4 Alternativa 3 (H. Control Versiones – CVS)...16

2.3.4 Alternativa 4 (H. Control Versiones – SVN)...16

2.3.5 Alternativa 5 (H. Desarrollo – Buildbot)..17

2.3.6 Alternativa 6 (H. Desarrollo – Tinderbox)..17

2.3.7 Solución propuesta...17

2.4 Planificación...18

2.4.1 Diagrama WBS...18

2.4.2 Fases y actividades del proyecto......19
2.4.4 Recursos del proyecto......20
2.4.5 Calendario del proyecto......23
2.4.6 Evaluación de riesgos......27
2.5 Viabilidad económica......28
2.5.1 Estimación de costes......28
2.5.2 Análisis coste-beneficio......30
2.6 Viabilidad técnica......30
3. Definición metodología......31
3.1 SQA......31
3.1.1 Modelos estándar......32
3.1.2 Modelos de aplicación o específicos de compañías......35
3.2 Perspectiva general......36
3.2.1 Guías y especificaciones......36
3.2.2 Herramientas......38
3.2.3 Elementos no documentados......39
3.3 Buildbot......40
3.3.1 Breve descripción de la herramienta......40
3.3.2 Configuración general......42
3.3.3 Organizadores......42
3.3.4 Constructores......44
3.3.5 Informes de estado......44
3.3.6 Guía de instalación y configuración......45
3.3.7 Script de instalación......45
3.4 Guía de estilo de codificación......46
3.4.1 Contenido......46
3.4.2 Evolución del documento......47
4. Trabajo en módulos Common y DMLib 48
4.1 Introducción a los módulos......48
4.1.1 Common......48
4.1.2 DMLib......53

4.2 Documentación......55

4.2.1 Comentarios inline......55

4.2.2 Comentarios doxygen......56

4.3 Cambios realizados......58

*4.3.1 Uso uniforme de string y char *......58*

4.3.2 Separación en namespaces......59

4.3.3 Separación de archivos fuente y de cabecera......59

4.3.4 Cambios puntuales......60

4.4 Desarrollo de nuevas Características......62

4.4.1 Sistema de excepciones......62

4.4.2 Sistema de configuración......66

5. Testing......68

5.1 Unit Testing y ECUT......68

5.2 Casos de prueba realizados......69

6. Conclusiones......75

6.1 El futuro de MATE......75

6.2 Conclusiones personales......76

Bibliografía......78

Índice de anexos......79

1. Introducción

1.1 Antecedentes

Debido a la naturaleza del presente proyecto, resulta pertinente empezar dando una breve visión de HPC (High Performance Computing). La computación ha sido una de las mayores herramientas para estudios científicos desde su aparición y la capacidad de cálculo ha ido aumentando en grandes proporciones desde entonces. La rama de las ciencias de la computación relacionada con esto se denomina computación de altas prestaciones. Consiste en obtener la mayor capacidad de cálculo posible utilizando supercomputadores y diferentes técnicas para aumentar el rendimiento, cómo programación paralela o distribuida.

1.1.1 Arquitecturas paralelas

El aspecto que presenta más interés para el proyecto son las arquitecturas paralelas. Hasta cierto momento se ha estudiado el procesamiento a nivel del procesador y se han aplicado técnicas de segmentación. Esto podría considerarse un primer mecanismo de paralelismo, ya que varias instrucciones consecutivas son ejecutadas de forma solapada. También se ha visto en los procesadores superescalares como se realiza algo de procesamiento paralelo lanzando dos o más instrucciones al mismo tiempo gracias a la presencia de varios cauces paralelos.

Sin embargo todos estos se ciñen a la arquitectura de Von Neummann con un procesador y memoria donde se guardan los datos y el programa, es decir, una máquina secuencial que procesa datos escalares. Este modelo ha ido evolucionando incluyendo elementos de paralelismo, pero sigue siendo una máquina de ejecución con un único flujo de instrucciones. No existe una frontera clara entre la arquitectura monoprocesador y las masivamente paralelas.

Podemos considerar una arquitectura paralela un escenario en el que se descompone un proceso secuencial en suboperaciones, y cada subproceso se ejecuta en un segmento dedicado que opera en forma concurrente con los otros segmentos. De esta manera cada segmento obtendrá un resultado parcial y se obtendrá el resultado del cálculo una vez todas las suboperaciones finalicen. Una de las técnicas para implementar este tipo de operaciones se trata del uso de aplicaciones paralelas o distribuidas.

1.1.2 Aplicaciones paralelas

El desarrollo de aplicaciones paralelas surgió a raiz de las arquitecturas paralelas mencionadas previamente. En estas aplicaciones el procesamiento se reparte en diferentes tareas independientes que se ejecutarán concurrentemente.

A la hora de trabajar con este tipo de programas, existen dos paradigmas básicos que implementan esta idea. El primero es el uso de memoria compartida. En este modelo, los procesos comparten una dirección de memoria donde leen y escriben de forma asíncrona. Para esto se utilizan sistemas de exclusión mutua como pueden ser semáforos o mutex. Una de las ventajas de este modelo es que el concepto de "posesión" de datos es inexistente, ya que no es necesario especificar la comunicación entre procesos para realizar intercambio de información. Sin embargo en cuanto a rendimiento los resultados de esta práctica son menos eficientes, ya que el control de memoria puede llegar a utilizar muchos recursos. El otro paradigma que podemos ver implementando estas aplicaciones es el paso de mensajes. En este modelo cada tarea utiliza su propios datos y es totalmente independiente de las demás en la fase de cálculo. Con esto es más fácil trabajar en máquinas diferentes y repartir la carga más fácilmente. Más tarde, una vez completada la tarea, se entregan los resultados utilizando una comunicación entre procesos. Aunque a la hora de implementar el programa pueda aumentar la complejidad, los resultados son mejores y se obtiene un mayor rendimiento.

Un sistema que implementa el segundo modelo de aplicación paralelas es MPI (Message Passing Library). Se trata de un protocolo de comunicaciones que soporta comunicación punto-a-punto y colectiva. La meta de este protocolo es conseguir un alto rendimiento, con buena escalabilidad y portabilidad. Hoy en día se sitúa como el modelo dominante utilizado en HPC.

1.1.3 Análisis de rendimiento

Como ya se ha mencionado la finalidad de HPC es aumentar el rendimiento de cálculo para operaciones costas. Por eso es importante conocer las herramientas de análisis de rendimiento más utilizadas.

La manera de realizar este análisis está enfocado al estudio de los factores que puedan afectar el rendimiento del sistema. Esto nos permitirá identificar los puntos importantes y de esta manera poder solucionar los problemas y aumentar las ventajas. Por lo tanto esto implica una combinación de medida, interpretación y representación de los atributos que representan el sistema. Se puede realizar de tres diferentes maneras. Por un lado es posible utilizar medidas reales sobre el sistema, el problema de utilizar esta técnica es que tiene poca flexibilidad y el coste puede llegar a ser muy alto. Por otro lado es posible generar simulaciones, de esta manera es posible realizar una buena cantidad de pruebas con una alta flexibilidad. Finalmente, la solución más rápida y precaria es realizar un modelado analítico. Se trata de crear una descripción matemática de ciertas características del sistema y atacar el proceso de análisis desde ese resultado.

	Flexibilidad	Coste	Exactitud	Precisión
Medida	baja	alto	alta	alta
Simulación	alta	medio	media	media
Modelado analítico	alta	bajo	baja	baja

1.1.4 Sintonización de procesos

Finalmente, entramos en el tema que incube a este proyecto más directamente, la sintonización de aplicaciones paralelas. Después de realizar estudios sobre este tipo de aplicaciones y recoger varios datos con métricas, uno de los sistemas que ha demostrado ser más eficiente en la práctica es la sintonización dinámica de procesos paralelos.

El problema con el análisis clásico es que el desarrollador debe tener experiencia en la programación de aplicaciones paralelas y el análisis de rendimiento para poder modificar el comportamiento de la aplicación. Para superar estas dificultades, las herramientas de sintonización de procesos paralelos aportan una serie de aspectos que permiten un análisis automático del rendimiento. Estas herramientas se basan en una serie de fases. Primero, recogen una colección de métricas de la aplicación en funcionamiento con una herramienta de monitorización. Una vez se han recogido estos datos, se procede a realizar el análisis automático. Este proceso busca problemas de rendimiento y bottlenecks. Muchos de estos problemas con aplicaciones paralelas han demostrado ser los mismos con años de experiencia en el análisis del rendimiento. Por eso estas herramientas poseen una base de conocimiento sobre los posibles bottlenecks y como encontrarlos. Una vez se han detectado los problemas, se genera una posible solución y se realiza la sintonización de la aplicación.

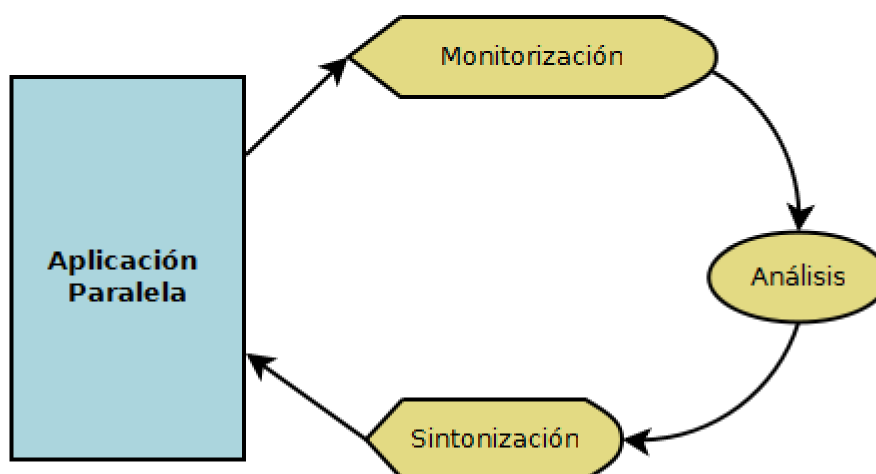


Figura 1.1 Proceso de sintonización de aplicaciones paralelas.

Existen diferentes técnicas a la hora de aplicar esta sintonización sobre la aplicación objetivo. Una de ellas es la sintonización dinámica a través de tunlets. Estos tunlets son librerías asociadas a estos problemas que contienen información sobre como solucionarlos. Para poder cooperar con el entorno, la implementación de cada tunlet está basada en la API de sintonización dinámica que proporciona el módulo de análisis.

1.2 Sobre MATE

MATE es un programa que implementa la sintonización de aplicaciones paralelas descrita en el punto anterior. La aplicación fue desarrollada junto a la tesis doctoral de Anna Morajko[1] en 2003 y desde entonces ha ido evolucionando. Las siglas de MATE significan "Monitoring, Analysis and Tuning Environment", en relación a las fases que realiza para llevar a cabo la sintonización. El ciclo principal de ejecución del entorno consiste en tres fases: monitorización de la aplicación objetivo, análisis de sus parámetros de ejecución y modificación de los mismos.

La primera fase de monitorización consiste en instrumentar de forma dinámica y automática la aplicación objetivo para conseguir información sobre su comportamiento. Para poder instrumentar de forma dinámica una aplicación se utiliza DynInst, una librería que permite insertar fragmentos de código (llamados snippets) dentro del programa en ejecución. Mediante esta librería MATE se ocupa de insertar uno de sus módulos, DMLib (Dynamic Monitoring Library), y recoger los datos necesarios. Los resultados obtenidos se comunican a través de la librería de paso de mensajes MPI.

En la fase de análisis se contrastan los resultados de la monitorización con los modelos teóricos de comportamiento para buscar bottlenecks, detectar sus causas y determinar soluciones. Todo esto se realiza en el módulo Analyzer que dispone de una serie de Tunlets como base de conocimiento.

Finalmente la fase de modificación consiste en la introducción de los cambios determinados en la fase de análisis en la aplicación objetivo. Este trabajo también lo realiza utilizando Dyninst. Todo este ciclo es continuo y por eso mismo en esta fase también se pueden realizar cambios sobre los parámetros de monitorización. De esta manera el análisis va mejorando con el tiempo de ejecución y la sintonización da mejores resultados.

1.3 Objetivos del proyecto

Debido a su envergadura, los objetivos de este proyecto se dividirán entre los diferentes miembros del equipo de desarrollo. A continuación se listarán los objetivos generales y se especificará la división de los mismos.

1 Anna Morajko, "Dynamic Tuning of Parallel/Distributed Applications", UAB, 2003.

	Objetivo	Prioridad	Miembro Asignado*
1	Crear especificaciones del entorno de desarrollo	Prioritario	Grupo
2	Implantar entorno de desarrollo	Crítico	
2.1	Herramienta de colaboración	Crítico	Rodrigo Echeverría, Antonio Pimenta
2.2	Herramienta de control de versiones	Crítico	Antonio Pimenta
2.3	Herramienta de construcción	Crítico	Noel De Martin
3	Construir la metodología de desarrollo.	Crítico	Grupo
3.1	Guía de estilo de documentación.	Crítico	Rodrigo Echeverría
3.2	Guía de estilo de codificación.	Crítico	Noel De Martin
3.3	Guía de estilo de documentación de código.	Prioritario	Rodrigo Echeverría
3.4	Guía de estilo de desplegamiento.	Prioritario	Antonio Pimenta
4	Aplicar la metodología y especificaciones a MATE.	Crítico	
4.1	Aplicación sobre las clases comunes (Common)	Crítico	Noel De Martin
4.2	Aplicación sobre el módulo DMLib.	Crítico	Noel De Martin
4.3	Aplicación sobre el módulo AC.	Crítico	Antonio Pimenta
4.4	Aplicación sobre el módulo Analyzer.	Crítico	Rodrigo Echeverría
5	Desarrollo de nuevas características	Secundario	
5.1	Crear un instalador para cualquier versión de Linux.	Secundario	Rodrigo Echeverría, Antonio Pimenta
5.2	Crear un lector de configuraciones flexible.	Secundario	Noel De Martin
5.3	Crear un sistema de cerrado de MATE.	Secundario	Rodrigo Echeverría, Antonio Pimenta
6	Crear documentación de MATE para futuros colaboradores y usuarios.	Prioritario	Grupo

1.4 Estructura del documento

Este documento está dividido en 6 capítulos principales.

El presente primer capítulo introduce el tema y finalidad del proyecto. Primero podemos encontrar la sección antecedentes que explica todo lo necesario para situarse en el contexto del programa MATE y las aplicaciones paralelas. Seguidamente se muestran los objetivos del proyecto y la distribución de estos.

En el segundo capítulo se describe la evolución que ha tenido el proyecto y como se ha llevado a cabo. El primer estudio de viabilidad muestra la situación actual y estudia la lógica del sistema y los diferentes aspectos necesarios antes de comenzar con el proyecto. En la planificación podemos encontrar todas las tareas realizadas, así como los recursos utilizados. También se muestra un análisis de coste-beneficio y la explicación de las tareas a realizar antes de comenzar con el proyecto.

En el tercer capítulo ya se entra en materia detallando la primera parte del proyecto, el desarrollo de la metodología. Primero se muestra una perspectiva general sobre las metodologías y se explica el por qué se ha decidido realizarla. A continuación se presenta la metodología general trabajada por las tres partes implicadas y finalmente se detallan en profundidad las herramientas y especificaciones trabajadas en este proyecto.

Los siguientes dos capítulos hablan de la segunda parte del proyecto, el trabajo realizado en los módulos de MATE. En el cuarto se describen los módulos trabajados y se da una explicación detallada de todos los elementos que contiene. Después se pueden encontrar explicaciones de todo el trabajo realizado a través del proyecto y los resultados obtenidos. En el capítulo cinco se describe el proceso de test realizado para asegurar el cumplimiento de los objetivos establecidos y se detalla el sistema que se ha seguido para ello.

Finalmente el último capítulo habla de los resultados del proyecto y el cumplimiento de los objetivos establecidos previamente. También acaba cerrando el documento con una reflexión sobre las repercusiones del proyecto y la evolución de MATE.

2. Viabilidad y Plan del proyecto

2.1 Estudio de la situación actual

Actualmente MATE está en una versión enfocada a la investigación y se busca modificar este enfoque aplicando una metodología de desarrollo firme y obteniendo una documentación clara para tratarse como producto. Además, existen otras aplicaciones con funcionalidades similares, también orientadas a la sintonización dinámica de rendimiento. Las más destacadas y cercanas a MATE son AutoPilot, y Active Harmony.

Por otro lado cabe destacar que MATE tiene un valor añadido ya que posee características únicas que lo distinguen de otras alternativas: no necesita insertar código fuera del tiempo de ejecución como Autopilot (puede realizar modificaciones en tiempo de ejecución) y basa su análisis en modelos de comportamiento y reglas básicas, no en heurísticas como Active Armony o en lógicas rebuscadas como Autopilot.

2.1.1 Contexto

El entorno de MATE ha sido probado con un amplio conjunto de aplicaciones paralelas y distribuidas obteniendo resultados satisfactorios de su funcionamiento, por lo que se puede decir que la herramienta presenta un grado de funcionalidad plena. Una vez se ha comprobado que el software es funcionalmente operativo y cumple con sus objetivos de diseño, el equipo de desarrollo de MATE pretende mejorar la calidad del mismo y ampliar el espectro de usuarios que puedan utilizarlo.

2.1.2 Lógica del sistema

En nuestro entorno es necesario realizar sintonización dinámica. Desde el punto de vista funcional podemos distinguir tres fases básicas descritas en los puntos siguientes.

Monitorización

Esta fase es la encargada de obtener información sobre la ejecución de la aplicación. Esto no es un proceso trivial ya para obtener medidas de rendimiento de la aplicación esta debe estar en marcha, por lo tanto esta tiene que incluir fragmentos de código que se encargan de captar eventos y notificar de estos al AC. Estos fragmentos pueden haber sido introducidos en el código original por el programador o se pueden introducir de forma directa en el programa compilado y en ejecución. En nuestro caso optamos por la segunda opción de nos da un grado de versatilidad mayor.

Para que esto sea posible la aplicación que queremos monitorizar debe ejecutarse bajo la tutela de un proceso de control y recolección [AC]. Este proceso se encarga, en una primera instancia, de introducir en la aplicación objetivo una serie de funciones baliza que monitorizan una parte específica de la aplicación, y posteriormente de almacenar y tratar los resultados. Es necesario también cargar en la aplicación una librería que contiene las herramientas necesarias para la monitorización de la aplicación [DMLib].

Estas medidas de rendimiento pueden ser de varios tipos, por ejemplo tiempos de ejecución de funciones clave o repeticiones de llamadas a una misma función. También se pueden medir las veces que ocurre un evento complejo.

No obstante, para conseguir obtener medidas de rendimiento, se debe conocer profundamente la aplicación que se desea optimizar. MATE utiliza unos modelos de rendimiento adaptados estrechamente a la aplicación objetivo. Estos proporcionan información sobre cómo obtener datos útiles para la medición del rendimiento y medidas a tomar para mejorarlo.

Por ultimo debe existir un medio adecuado para transmitir los datos recolectados a un proceso de análisis para obtener resultados y, si es posible, sintonizar la aplicación para mejorar su rendimiento.

Algunos aspectos que se deben tener en cuenta al trabajar en diferentes máquinas son las posibles diferencias en los relojes de sus procesadores y el tiempo de transmisión de los datos. Es importante que una serie de eventos que ocurren en la aplicación lleguen en ese orden al analizador.

Análisis

Una vez obtenidas las mediciones adecuadas un programador experto y conocedor de la aplicación a optimizar, sabría encontrar los cuellos de botella en su ejecución y podría proponer soluciones a estos. No obstante esta es una tarea pesada y duradera y como hemos dicho requiere un nivel de conocimiento muy alto.

Como alternativa existen métodos de análisis automático. Estas herramientas identifican problemas en la ejecución de la aplicación e incluso proporcionan soluciones a estas. Para que esto sea posible se le debe proporcionar a la herramienta de análisis una base de conocimiento sobre la aplicación así como posibles zonas críticas donde buscar problemas. El proceso de producir unos modelos que permitan al analizador automático identificar exitosamente estos problemas no es fácil. Además, a pesar de que se obtenga un modelo valido. Las soluciones proporcionadas serán estrictamente útiles para el comportamiento que tuvo la aplicación durante esa ejecución concreta.

Con MATE se intenta atajar este problema no solo automatizando el proceso de análisis sino además realizarlo de forma dinámica. Esto implica que el análisis de puede realizar mientras la aplicación se ejecuta, eliminando la necesidad de un archivo donde almacenar los datos de medición. Este método, pese a que conserva muchas de las desventajas del análisis manual, permite un análisis adaptativo en aplicaciones iterativas ya que durante la ejecución se puede modificar dinámicamente la monitorización e instrumentación de la aplicación.

En el caso de MATE el análisis se produce dinámica y automáticamente, sin necesidad de un archivo con los datos de monitorización, ya que el módulo Analyzer recibe los eventos recolectados.

Optimización

La fase de optimización (Tuning) es en la cual aplicamos los cambios adecuados para mitigar los problemas detectados. Estos cambios se deben hacer en el código de la aplicación ya que forman parte de esta, y por tanto es normalmente necesario cerrar la aplicación, modificarla, recompilarla y volverla a ejecutar. Si los cambios aplicados son adecuados se debería observar una mejora en el rendimiento.

No obstante este método de optimización requiere la atención directa del programador y además es necesario recompilar, y por tanto cerrar, la aplicación. Los cambios realizados podrían ser inútiles si en la siguiente ejecución el programa se comporta de forma distinta, debido, por ejemplo, a diferentes valores de entrada.

MATE proporciona un modelo de optimización autónomo que no requiere de la intervención del programador y que se realiza dinámicamente, es decir, sin cerrar la aplicación. Además, el hecho de que se sintonice la aplicación de forma dinámica permite que se realice de forma adaptativa, así aunque cambien las condiciones de la ejecución el programa sigue operando de forma eficiente.

En MATE la aplicación de estas variaciones es llevada a cabo por el AC (Aplication Controller), este dispone de un sintonizador (Tuner) que contiene las herramientas necesarias para instrumentar la aplicación dinámicamente. Usando la información proporcionada por el Analyzer el AC introduce los cambios necesarios en la aplicación mientras esta se ejecuta.

Estas tres fases tienen que estar realizándose continua, dinámica y automáticamente mientras el programa está en ejecución. Para que este método sea efectivo la aplicación objetivo debe ser iterativa (ejecución de un bucle que realiza de forma repetida una serie de instrucciones), y se obtiene la eficacia en procesos largos y que usen muchos recursos.

Módulos de MATE

Básicamente, MATE está formado por los siguientes módulos que operan conjuntamente, controlando e intentando mejorar el rendimiento de la aplicación objetivo.

- **AC (Aplication controler):** Se trata de un proceso daemon que controla la ejecución de la aplicación. Este proceso se inicia de forma manual en cada uno de los nodos y es el que se encarga de inicializar las tareas (aplicación) que se van a monitorizar.

  Lo primero que hace el modulo es cargar en la imagen del proceso que representa la aplicación una librería dinámica llamada DMLib, que contiene las herramientas necesarias para la monitorización dinámica de la aplicación. Esta librería debe inicializarse con los datos del analizador para que sea posible la comunicación entre los nodos y el módulo Analyzer.

  A continuación se buscan los puntos indicados por el Analyzer donde se deberán introducir los monitores que recopilan información de la ejecución. Esto es diferente para cada aplicación y viene indicado por unos módulos incluidos en el Analyzer.

  Una vez aplicada esta lista, y en línea, se procede a iniciar la aplicación, y una vez esta está iniciada los datos recopilados se envían como eventos al Analyzer directamente desde la aplicación usando funciones de la DMLib.

En este punto el AC ha de estar preparado para recibir instrucciones del Analyzer a la vez que controla la aplicación en ejecución ya que esta puede producir nuevos eventos que reportar. En cuanto se recibe una petición de sintonización (Tuning request), el AC informa a las tareas de que han de ser modificadas y estas a su vez pedirán al Tuner que las actualice, siempre esperando a que se dé el momento adecuado. Esta sincronización de las modificaciones es posible gracias a los "break points" (puntos de parada), que indican los lugares donde se debe insertar la instrumentación nueva y paran la ejecución de la aplicación para que estas modificaciones sean posibles.

Este ciclo de comunicación-modificación se realiza de forma cíclica hasta que se han realizados todas las modificaciones o la aplicación se ha cerrado.

- **DMLib (Dynamic Monitoring Library):** Una librería compartida que se carga dinámicamente dentro de las tareas de la aplicación para facilitar la instrumentación y recolección de datos. La librería contiene funciones que son responsables del registro de eventos con todos los atributos necesarios para entregarlos para análisis. Usamos la función *loadLibrary* de dyninst para cargar la librería una vez la tarea ya ha sido iniciada.

  Esta librería debe ser inicializada con los datos del Analyzer para hacer posible la transmisión de los datos recolectados. Esta inicialización se inserta como un snippet en la aplicación.

  Una vez cargada e inicializada esta librería implanta las conexiones necesarias vía proxy para comunicarse con el Analyzer.

- **Analyzer:** Es el proceso que se encarga de analizar el rendimiento de la aplicación, detecta problemas de rendimiento a tiempo real y solicita los cambios idóneos para mejorarlo.

  Mediante un sistema de captura de eventos este módulo obtiene información sobre la ejecución de la aplicación y aplica unas funciones específicas a la aplicación para identificar problemas y proporcionar al AC posibles soluciones. Este procedimiento es también cíclico y en cada iteración se manejan varios eventos que resultan en soluciones para el AC.

  Pese a que es el AC quien se ejecuta junto a la aplicación el Analyzer dispone de una abstracción de esta que usa para identificar los eventos con de cada una de las tareas que se ejecutan en los diferentes nodos.

– **Common:** Por último existe un módulo en MATE que contiene las clases compartidas que son usadas por los demás módulos. Este módulo cumple el objetivo de reutilización de código y encapsulación de los diferentes componentes de MATE.

2.1.3 Descripción física

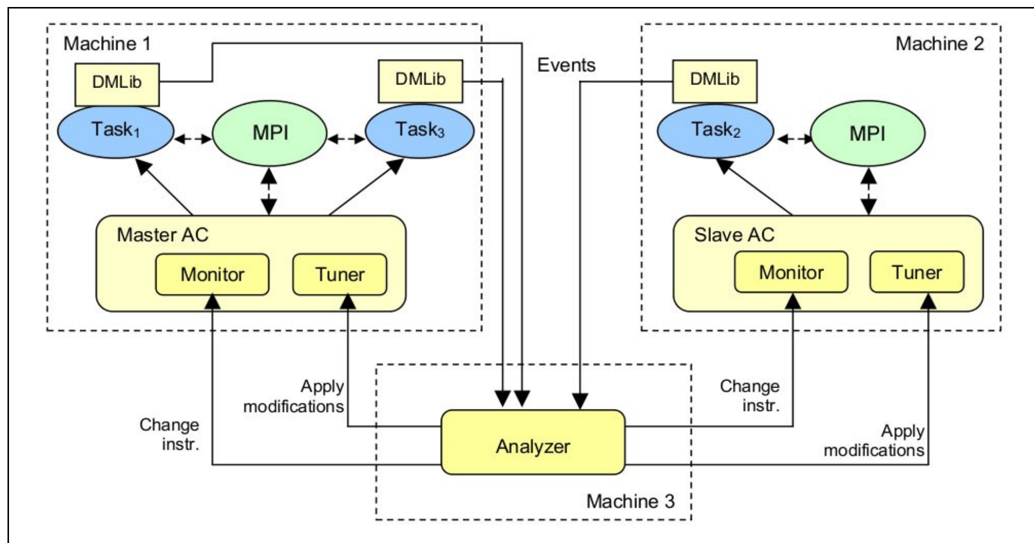


Figura 2.1 Esquema del sistema, extraído de la tesis de MATE.

En cuanto a aplicación, MATE se divide en dos partes diferenciables, el Analyzer y el Aplication Controller. Estos son los dos ejecutables que cumplen funciones distintas y se complementan para formar en entorno de sintonización que es MATE. Al ser una herramienta para programas distribuidos, MATE se ejecuta en diferentes máquinas, en concreto el modulo AC se ejecuta en cada uno de los nodos que ejecutan procesos de la aplicación objetivo, mientras que el Analyzer es un programa centralizado que se ejecuta en una sola máquina.

Al empezar la aplicación, MATE distribuye un proceso de AC en cada máquina para controlar el comienzo de las tareas.

Otro de los componentes de MATE es la librería DMLib (*dynamic monitoring library*), cuando comienza una nueva tarea MPI, el AC carga la librería compartida de monitorización en la memoria de la tarea para permitir la instrumentación de esta. Esto le permite al Analyzer añadir/eliminar eventos dinámicamente para recolectar información y realizar la sintonización.

2.1.4 Diagnóstico del sistema

Actualmente la aplicación está muy restringida y solo es capaz de funcionar en escenarios con características muy concretas. Además al ser un proyecto destinado a la investigación, carece de las metodologías de desarrollo necesarias para su expansión, como la documentación o sistemas de colaboración.

Esto se podrá solventar con un proceso de testeo dónde obtendremos una serie de errores a reparar y que definirán el curso de la segunda parte del proyecto. También será necesario crear una documentación de respaldo para permitir la evolución del producto.

2.2 Requisitos funcionales y no funcionales

Al tratarse de un proyecto que se basa en una aplicación ya desarrollada los objetivos naturales de un proyecto de desarrollo se ven llevados a un segundo plano.

Por tanto, aunque lo primero que puede surgir al pensar en requisitos sobre este proyecto sean requisitos de MATE (mejorar rendimiento de la aplicación, no sobrecargar demasiado la ejecución), sería un error enfocarlo de esa manera, ya que este proyecto no se trata de crear MATE, sino de crear una versión como producto software del mismo.

Esto es, desarrollar y aplicar una serie de procedimientos para producir, a partir de la versión existente de MATE, una aplicación adecuada a un uso general y proporcionar las herramientas necesarias para la continuidad de su desarrollo.

2.2.1 Requisitos funcionales

Debido al hecho de que MATE es un programa plenamente operativo los requisitos funcionales se reducen a que la funcionalidad actual se mantenga y a añadir algunos pequeños módulos que hagan más cómodo el uso de MATE aunque no modifiquen su funcionamiento principal, como por ejemplo un sistema de Shutdown (Apagado). El resto de requisitos aparecerán al realizar los test de prueba sobre los cuales trabajaremos para modificar el programa.

1.- Mantener funcionalidad actual de MATE.

2.- Adaptar a nuevos entornos.

3.- La aplicación debe cerrarse de forma controlada.

4.- Permitir lectura de archivos de configuración.

2.2.2 Requisitos no funcionales

En nuestro caso los requisitos no funcionales se basan en homogeneizar la codificación y documentación. Por lo tanto harán referencia a las guías de especificación. También serán encontrar los posibles errores del sistema mediante un sistema cíclico de testeo.

Dado que, como hemos explicado anteriormente, los requisitos funcionales básicos de MATE ya están satisfechos, la carga de nuestro proyecto se encuentra en los no funcionales. Estos, pese a no añadir funcionalidad a la aplicación incrementan su calidad.

1.- Homogeneizar codificación y documentación.

2.- Realizar proceso de testeo.

3.- El programa debe funcionar en todas las distribuciones de linux.

2.3 Alternativas y selección de la solución

Considerando que el proyecto consiste en crear una metodología de desarrollo productiva que sirva como base a los futuros desarrolladores de MATE, para posteriormente aplicarla al software existente, las alternativas se definen en el marco de herramientas de desarrollo, colaboración y en la selección de la propia metodología.

2.3.1 Alternativa 1 (H. Colaboración – Redmine)

Redmine es una herramienta de colaboración que actúa como solución todo-en-uno ya que posee soporte multiproyecto, acceso basado en roles, sistema de seguimiento, gestor de calendarios y diagramas de Gantt, soporte a wikis y foros y compatibilidad con diversos gestores de versiones concurrentes. En cuanto a coste, encontramos que se trata de una herramienta de código libre.

2.3.4 Alternativa 2 (H. Colaboración – Trac)

Trac es una herramienta de gestión de proyectos que enlaza una base de datos de errores de software, un sistema de versiones y el contenido de una wiki de colaboración. En cuanto a coste, encontramos que se trata de una herramienta de código libre y gratuito.

2.3.4 Alternativa 3 (H. Control Versiones – CVS)

CVS (Concurrent Versions System) es una aplicación cliente-servidor donde el servidor se encarga de guardar un historial de las diferentes versiones de cada uno de los archivos que componen un proyecto; los clientes pueden acceder a estos archivos de forma directa o bien remotamente. En cuanto a coste, encontramos que se trata de una herramienta de código libre y gratuito.

2.3.4 Alternativa 4 (H. Control Versiones – SVN)

SVN (Subversions) es una herramienta de control de versiones que tiene la peculiaridad de que mantiene un único número de versión para un conjunto de archivos, de forma que lo que conserva es un estado determinado del proyecto en general. Además, soporta el acceso desde redes, permitiendo a usuarios modificar los archivos desde distintas ubicaciones. En cuanto a coste, encontramos que se trata de una herramienta de código libre y gratuito.

2.3.5 Alternativa 5 (H. Desarrollo – Buildbot)

Buildbot es una herramienta de desarrollo software iterativa que automatiza los procesos de compilación y testing. Posee soporte para control de versiones (CVS, SVN, etc.). En cuanto a coste, encontramos que se trata de una herramienta de código libre y gratuito.

2.3.6 Alternativa 6 (H. Desarrollo – Tinderbox)

Tinderbox es una suite que proporciona capacidades de continua integración, básicamente permite manejar proyectos software y probar su funcionamiento en diversas plataformas. En cuanto a coste, encontramos que se trata de una herramienta de código libre y gratuito.

2.3.7 Solución propuesta

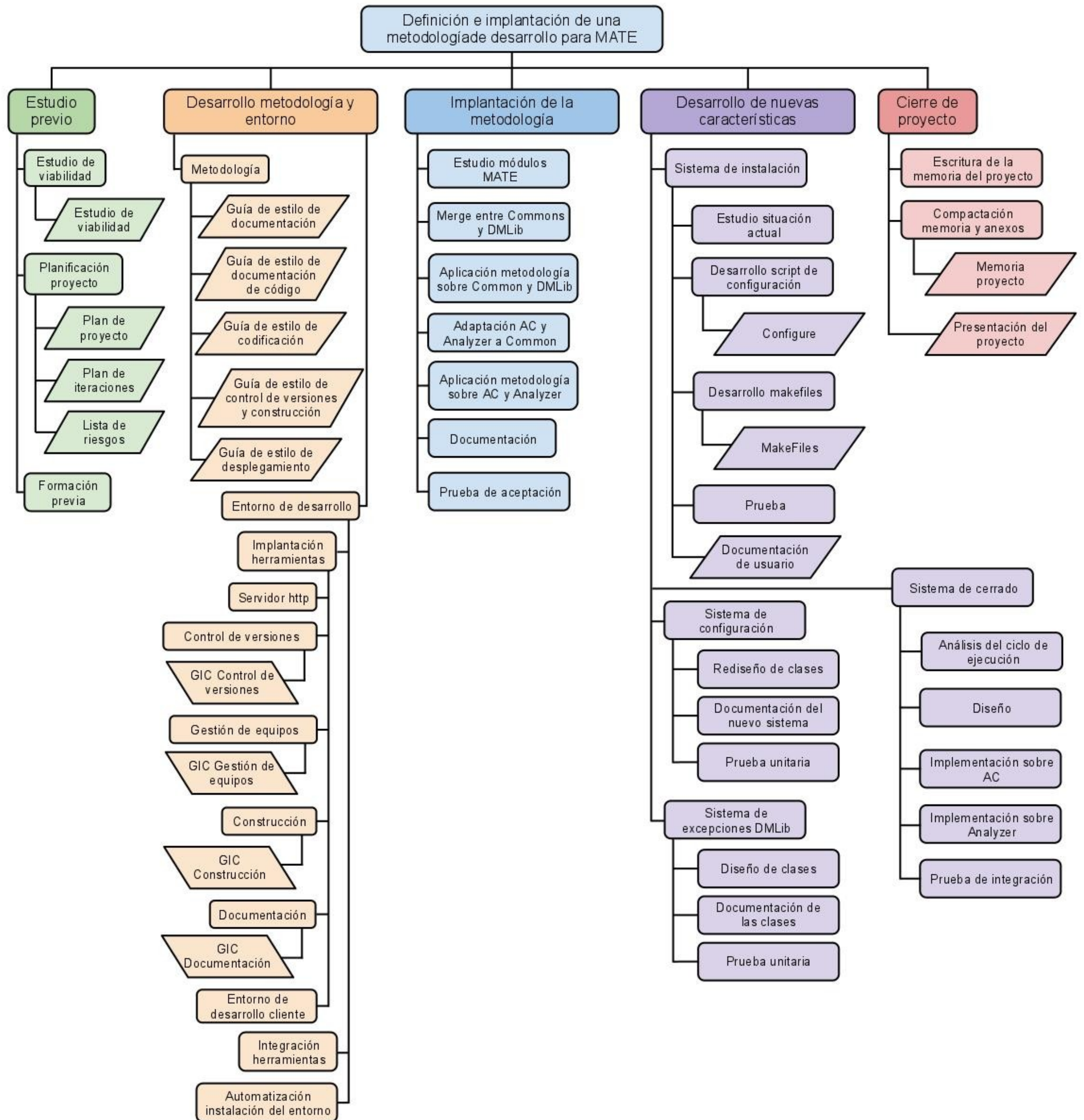
En cuanto a la herramienta de seguimiento se utilizará Redmine por dos razones: primero, es más completa y ofrece en una misma aplicación todas las herramientas que se necesitan y, segundo, la experiencia del director de proyecto con esta herramienta servirá como guía.

Sobre la herramienta de control de versiones se utilizará SVN, ya que interesa más guardar el proyecto por versiones en general, sin hacer hincapié en los archivos individuales.

Y finalmente la herramienta de desarrollo que se utilizará es Buildbot por su capacidad de integración con SVN y porque nuestros intereses no se dirigen especialmente al testing multiplataforma, sino a un ciclo iterativo de compilación-testing-recodificación.

2.4 Planificación

2.4.1 Diagrama WBS



2.4.2 Fases y actividades del proyecto

Fase	Actividad	Descripción	Iterativa
Estudio previo	Estudio de viabilidad	Estudio para analizar las posibilidades y mejores alternativas para realizar el proyecto y si estás son posibles con los recursos disponibles en el tiempo requerido.	No
	Planificación del proyecto	Análisis sobre las tareas que compondrán el proyecto, su calendario, los recursos necesarios para ejecutarlas y los riesgos que comportan a la consecución del proyecto.	No
	Formación previa	Estudio sobre los temas relacionados con paso de mensajes (MPI) y sintonización de procesos (Dyninst)	No
Desarrollo metodología y entorno	Desarrollo guías de estilo	Creación de los documentos que conforman la base de la metodología a implantar: guías de estilo de documentación, codificación, construcción, control de versiones, etc.	Si
	Implantación entorno	Selección de las diferentes herramientas que forman el entorno de desarrollo, implantación e integración de las mismas.	Si
Implantación metodología	Estudio módulos MATE	Estudio sobre el código en su versión original de cada módulo de MATE.	No
	Combinación DMLib y Commons	Eliminación de las clases redundantes entre Commons y DMLib y refactorización de AC y Analyzer en consecuencia.	Si
	Documentación y refactorizacion	Documentación sobre el código de cada módulo y refactorización derivada de las guías de estilo.	No
	Prueba unitaria	Prueba unitaria de las clases que componen cada módulo.	No
	Documentación	Extracción y compilación de la documentación sobre código.	No
	Prueba de aceptación	Prueba del sistema completo para detectar posibles errores en la refactorización.	No
Desarrollo de nuevas características	Sistema de instalación	Desarrollo de un sistema de instalación capaz de automatizar la búsqueda de dependencias y la compilación en el mayor grado posible.	Si
	Sistema de	Desarrollo de un sistema de lectura de configuraciones	Si

	configuración	para MATE fácilmente ampliable con diferentes tipos de entradas.	
	Sistema de excepciones de DMLib	Desarrollo de un sistema para DMLib de detección y notificación de errores.	Si
	Sistema de cerrado	Desarrollo de un sistema capaz de cerrar el entorno de forma centralizada y controlada.	Si
Cierre de proyecto	Escritura y compilación de la memoria	Escritura de la memoria del proyecto y compilación del documento junto con los anexos que lo acompañan.	No
	Exposición del proyecto	Exposición del proyecto ante un tribunal para su evaluación.	No

2.4.4 Recursos del proyecto

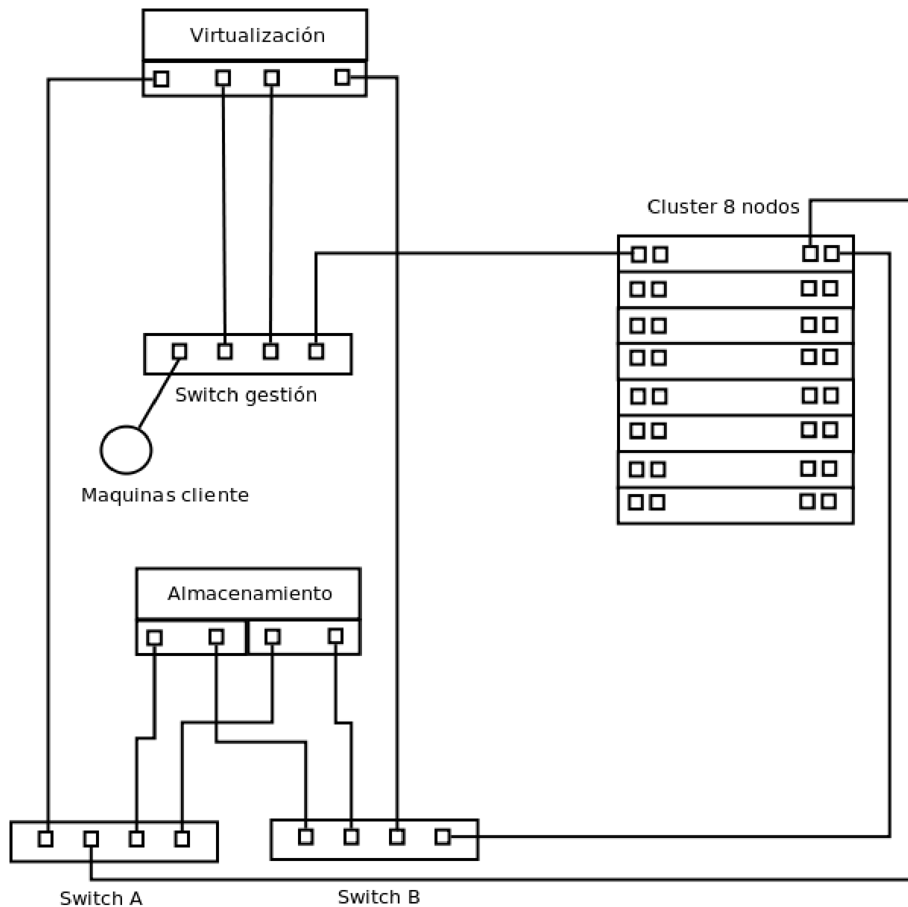
Recursos humanos

- 3 Programadores-Analistas: Noel De Martín, Rodrigo Echeverría, Toni Pimenta
- 1 Project Manager: Joan Piedrafita

Recursos de infraestructura

Para la realización del proyecto se dispone de la infraestructura que podemos observar en la figura 2.2. Este sistema consiste en una red SAN (Storage Area Network) que almacenará los datos y un cluster de 8 nodos dónde se ejecutarán las aplicaciones paralelas. Existen dos switches de control que se ocuparán del cluster y de los programas cliente, y un switch de gestión para la virtualización. En este último switch se podrán conectar las máquinas cliente, en nuestro caso portátiles, para realizar las pruebas necesarias.

Figura 2.2 Diagrama de infraestructura.



Los detalles técnicos de la infraestructura son:

- Servidor de proyectos, colaboración y builds virtualizado
  - Equipo: Dell PowerEdge R515
  - Procesador: 2 x AMD Opteron 4122 (4 Cores – 2.2 Ghz – L1 3MB / L2 6MB – 95W TDP)
  - Memoria: 8GB Memory for 2 CPUs, DDR3, 1333MHz (8x1GB Single Ranked UDIMMs)
  - Disco: 2x 250GB, SATA, 3.5-in, 7.2K RPM Hard Drive (Hot Plug)
- 8 Nodos de computo cluster DiskLess
  - Equipo: Dell PowerEdge R610
  - Procesador: 2 x Intel Xeon E5620 (4 Cores – 2,4 Ghz – 12 MB Cache – QPI 5,86 Gb/s)
  - Memoria: 12GB DDR3, 1333MHz ECC (12x1GB)
- SAN

- Almacenamiento: DELL™ PowerVault™ MD3200i, 6 discos SAS 7.2k rpm 500 GB
- Red de gestión: 2 x SWITCH ETHERNET DELL PowerConnect 5424

- Otros
  - Sistema de alimentación: SAI 8000VA APC
  - Switch control cluster:SWITCH ETHERNET DELL PowerConnect 5448
  - Switch gestión: SWITCH INFINIBAND SDR DE 8 PUERTOS, 4X, 1U.
  - Switch Red MATE: SWITCH ETHERNET DELL PowerConnect 5424
  - Rack PDU (8 Tomas + Ethernet)
  - Chasis Rack 42U
  - CABLE INFINIBAND 2 METROS CON CONEXION X4
  - Cable de interconexión - RJ-45 (M) - RJ-45 (M) - 2 m - UTP - (CAT 6)

2.4.4.1 Calendario de recursos

Los recursos humanos se utilizarán durante todo el proyecto, sin embargo el cluster y el almacenamiento solo se utilizarán en la segunda parte del proyecto haciendo las pruebas necesarias con aplicaciones paralelas para comprobar los resultados. El resto de recursos materiales también se utilizarán durante todo el proyecto.

2.4.4.2 Asignación de recursos

Fase	Actividad	Recursos asignados
Estudio previo	Estudio de viabilidad	Personal.
	Planificación del proyecto	Personal.
	Formación previa	Personal.
Desarrollo metodología y entorno	Desarrollo guías de estilo	Personal, ordenadores personales, local.
	Implantación entorno	Personal, ordenadores personales, local, servidor de proyectos.
Implantación metodología	Estudio módulos MATE	Personal, ordenadores personales, local.

	Combinación DMLib y Commons	Personal, ordenadores personales, local, servidor de proyectos.
	Documentación y refactorizacion	Personal, ordenadores personales, local, servidor de proyectos.
	Prueba unitaria	Personal, ordenadores personales, local, servidor de proyectos.
	Documentación	Personal, ordenadores personales, local, servidor de proyectos.
	Prueba de aceptación	Personal, ordenadores personales, local, servidor de proyectos y cluster de computadores.
Desarrollo de nuevas características	Sistema de instalación	Personal, ordenadores personales, local, servidor de proyectos.
	Sistema de configuración	Personal, ordenadores personales, local, servidor de proyectos.
	Sistema de cerrado	Personal, ordenadores personales, local, servidor de proyectos y cluster de computadores.
Cierre de proyecto	Escritura y compilación de la memoria	Personal, ordenadores personales.
	Exposición del proyecto	Personal, ordenadores personales.

2.4.5 Calendario del proyecto

El proyecto se realizará durante en el segundo cuatrimestre del curso 2010/11.

2.4.5.1 Dependencias

Al tratarse de un modelo lineal, o en cascada, cada fase empezará al terminar la anterior, a excepción de aquellas fases más delicadas que presentarán iteraciones para introducir cambios y correcciones una vez se hagan las pruebas necesarias.

Las fases del proyecto están representadas en el primer nivel de la jerarquía del diagrama WBS (sección 2.4.1).

La primera fase será el estudio previo, que incluye entrevistas con el cliente, estudios de viabilidad y planificación, así como toda la formación previa necesaria para los analistas.

La segunda fase consiste en el desarrollo del entorno y la metodología. Está fase consiste, por una parte, en la selección de un conjunto de herramientas que conformen un entorno de trabajo para futuros desarrolladores de MATE y la implantación de las mismas (documentación del sistema de instalación y configuración y la automatización del mismo) y, por la otra, en la confección de una serie de manuales (guías de estilo) que sirvan como una referencia sobre el “cómo hacer” para los futuros desarrolladores.

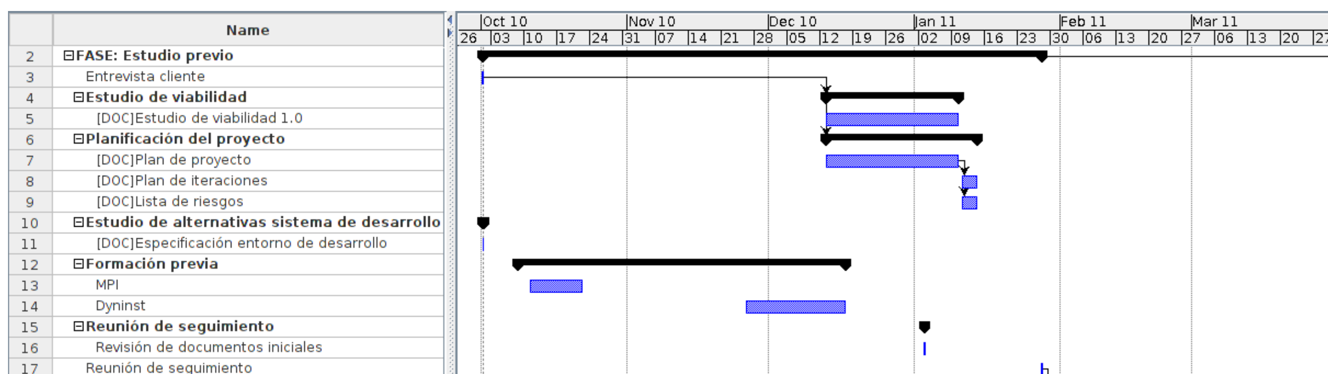
La tercera y cuarta fase se ejecutaran en paralelo. La tercera fase consiste en adaptar el código a las guías de estilo creadas y a la prueba unitaria del mismo. La cuarta fase consiste en el desarrollo de nuevas características para mejorar la calidad del software en general.

Finalmente, la fase de cierre de proyecto consiste en acabar la memoria del proyecto, compactarla con sus anexos y entregarla, además de la exposición del proyecto ante un tribunal.

2.4.5.2 Calendario temporal

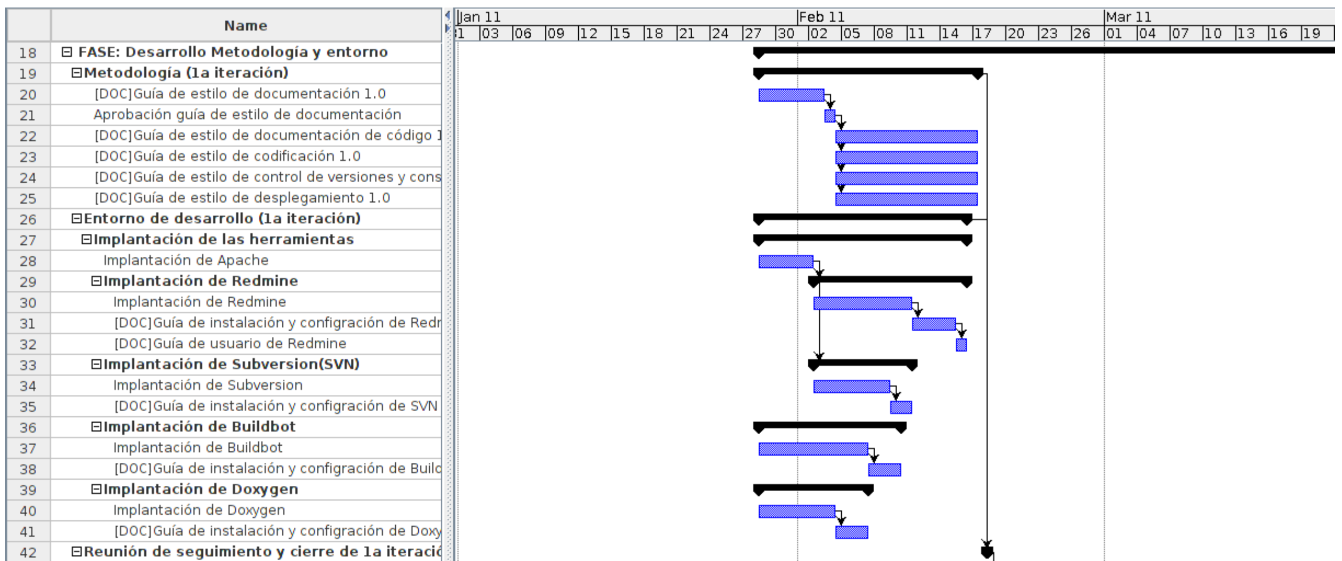
La duración total estimada del proyecto es 248 días que, con una dedicación media de 4h/día, implican 992 horas de trabajo a distribuir entre los tres miembros del equipo.

Fase 1: Estudio previo

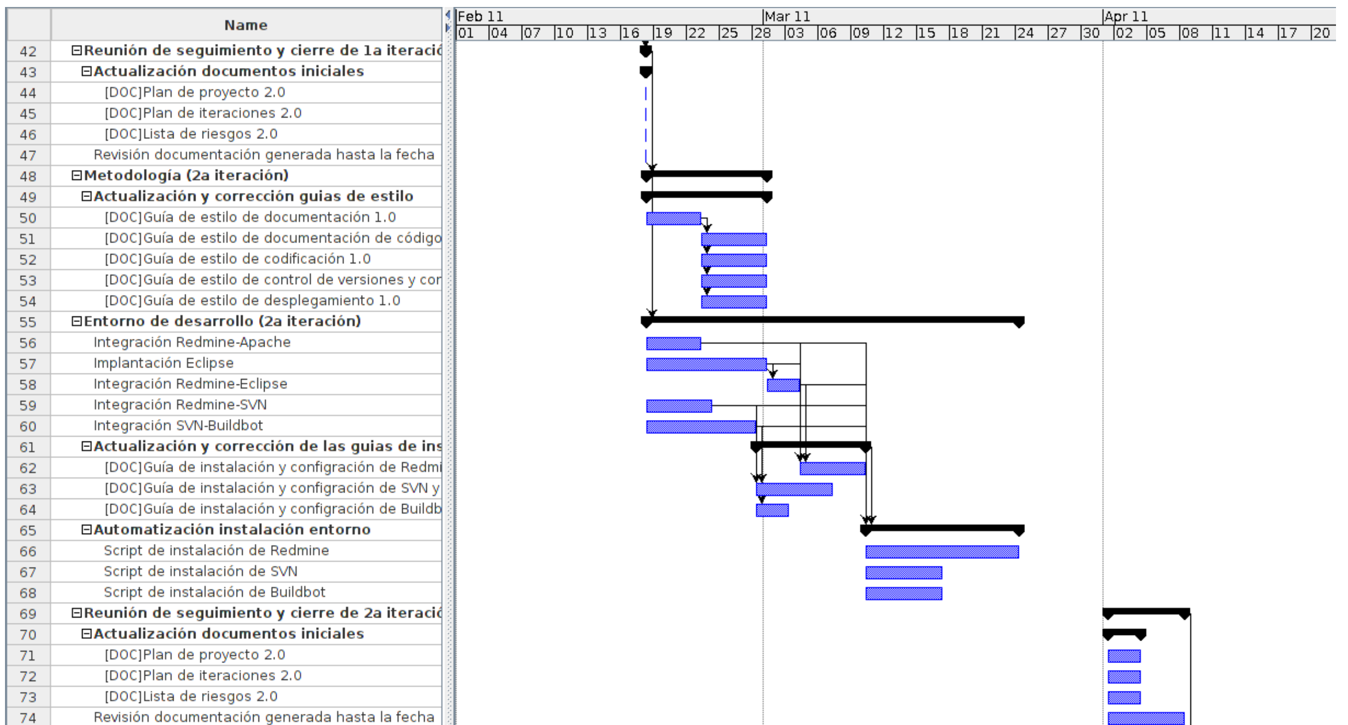


Fase 2: Desarrollo metodología y entorno

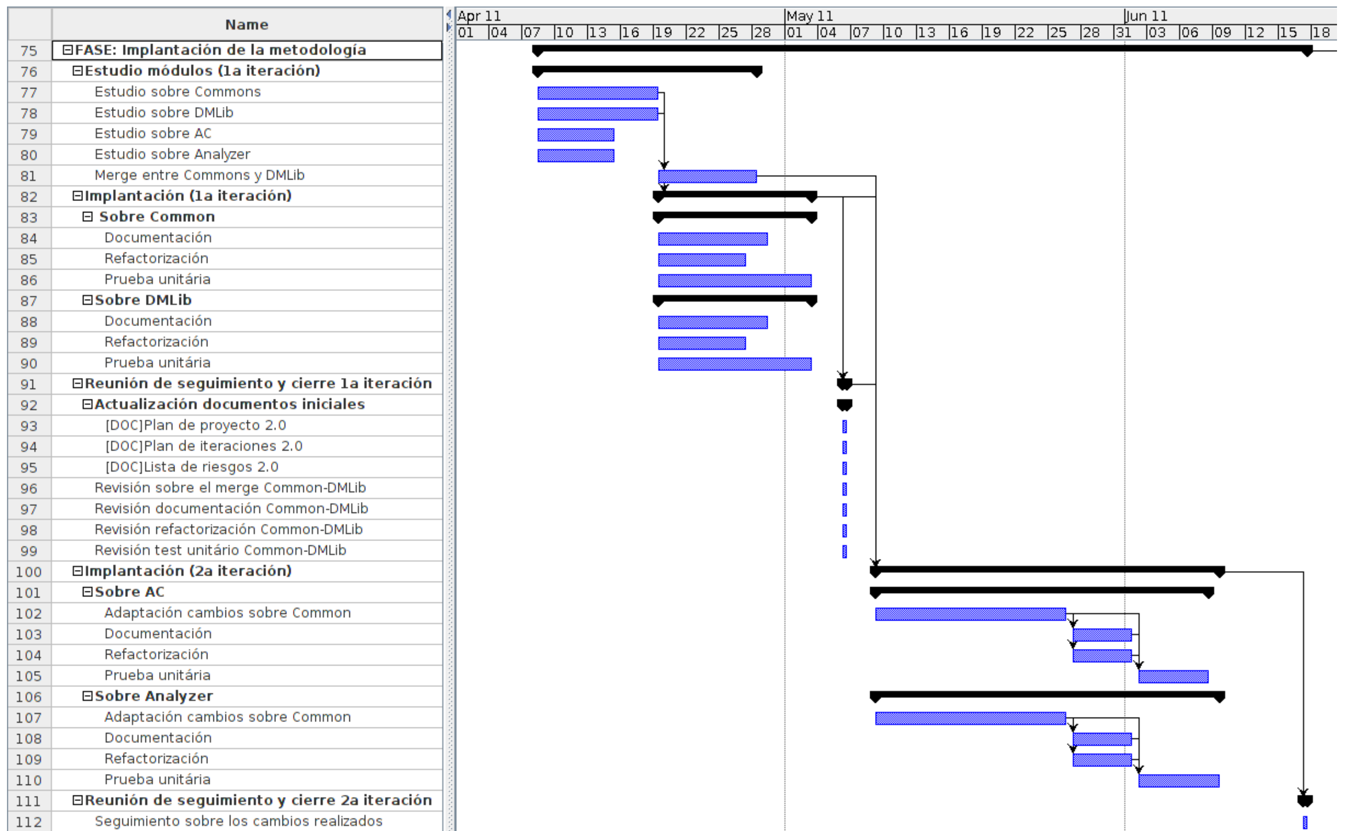
1ª iteración



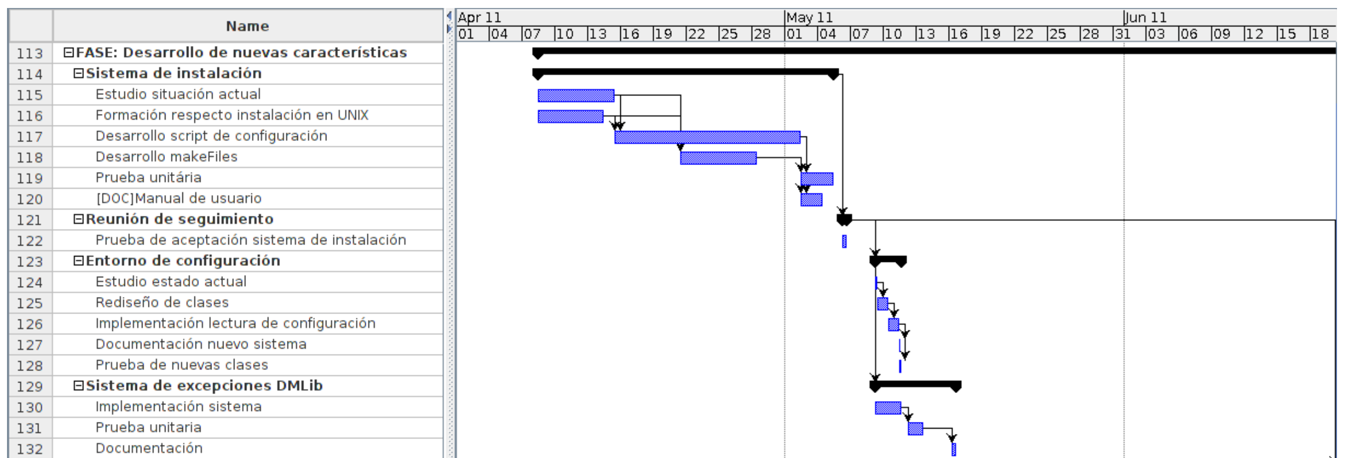
2ª iteración

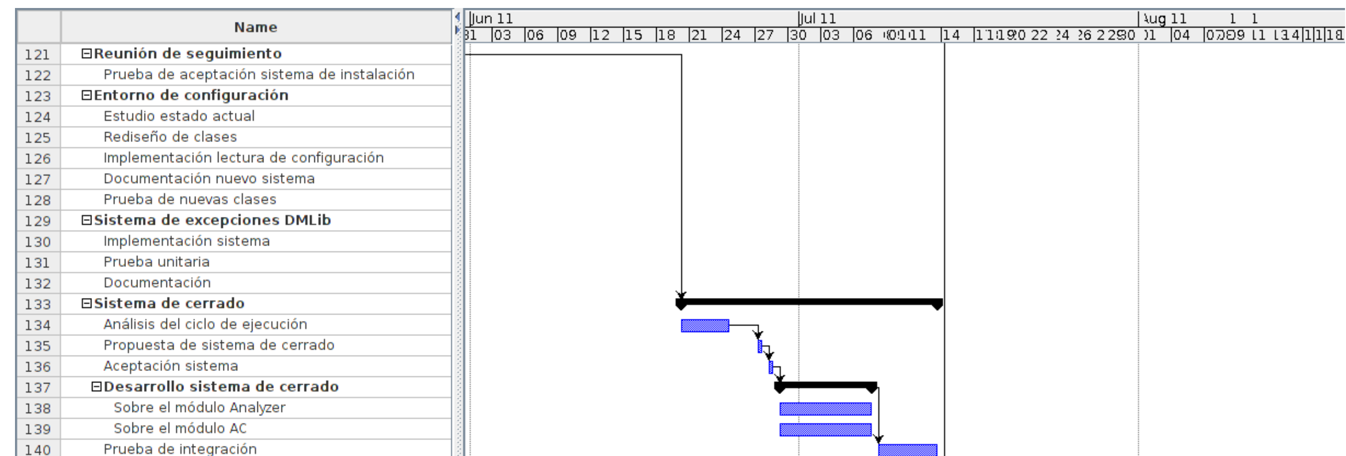


Fase 3: Implantación de la metodología

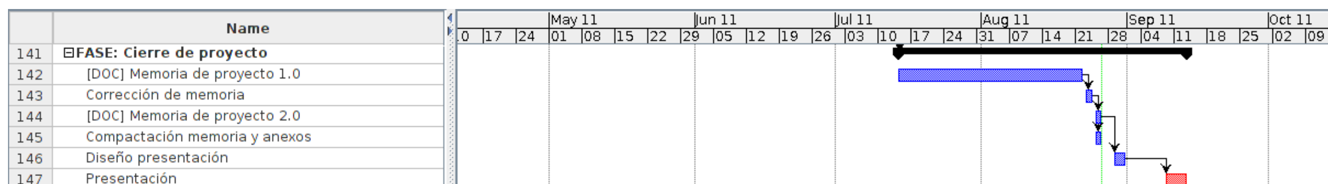


Fase 4: Desarrollo de nuevas características





Fase 5: Cierre de proyecto



2.4.6 Evaluación de riesgos

Lista de riesgos

1. Incumplimiento de plazos de entrega: Alguno de los plazos establecidos no se cumple.
2. Herramientas inadecuadas: problemas con las herramientas de desarrollo.
3. Incumplimiento de alguna norma: Repercusiones legales por el incumplimiento de alguna norma.
4. Abandono del proyecto de algún miembro: Uno de los miembros decide abandonar el proyecto.
5. Implementación incorrecta: Problemas a la hora de modificar MATE.

Catalogación de riesgos

Riesgo	Probabilidad	Impacto
1	Media	Marginal
2	Baja	Marginal
3	Baja	Crítico
4	Baja	Crítico
5	Baja	Marginal

Plan de contingencia

Riesgo	Plan de contingencia
1	Reuniones de seguimiento.
2	Planificación inicial.
3	Cotejar acciones futuras con las normativas presentes en el documento de viabilidad. El NDA hace...
4	Reorganizar planificación.
5	Consultar expertos, reunión de seguimiento.

$$Coste\ total = 92.500 + 47.213{,}91 + 49.488 + 14.900{,}45 = 204.102{,}36\ €$$

2.5 Viabilidad económica

2.5.1 Estimación de costes

Coste del personal

El personal ha utilizar será el siguiente:

- 3 Programadores-Analistas: Noel De Martín, Rodrigo Echeverría, Toni Pimenta
- 1 Project Manager: Joan Piedrafita

Y por lo tanto, el coste estimado sería el siguiente:

Recurso	Coste
3 Analista-programador	64.500 €
1 Project Manager	28.000 €
Total	**92.500 €**

Coste de los recursos

Se pueden observar los costes de todos los recursos en la siguiente tabla:

Recurso	Coste
Servidor	2.220,76 €
8 nodos cluster	23.902,08 €
SAN	8.173,86 €
Otros	12.917,21 €
Software	0 €
Total	**47.213,91 €**

Costes indirectos

Recurso	Coste
Electricidad	5.389,65 €
Consumibles	1.475 €
Telefonía	708 €
Limpieza	4.241,8 €
Mantenimiento	1.062 €
Gestión	2.124 €
Total	**14.900,45 €**

Otros costes

Recurso	Coste
Personal de soporte	24.000€
Alquiler local	25.488€
Total	**49.488 €**

2.5.2 Análisis coste-beneficio

Sobre este resultado hace falta matizar dos aspectos. Primero que los costes de personal incluyen a los diferentes autores de los respectivos proyectos, por lo tanto, se reducen a 0. Por otra parte, los costes de recursos se irán amortizando en los sucesivos proyectos referentes a MATE. El resto de costes quedan en negativo, pero el beneficio que comporta el proyecto en cuestiones de investigación y promoción lo compensa.

2.6 Viabilidad técnica

En este proyecto se puede decir que la viabilidad técnica es fácilmente justificable, dados diferentes factores de peso.

Sobre la primera parte, el desarrollo de una metodología, se dispone de tres técnicos capacitados para llevar a cabo este tipo de tarea. Observando la planificación se puede ver que se han tenido en cuenta las diferentes dificultades que se puedan encontrar y el aspecto técnico no es ningún problema.

La segunda parte de el proyecto, el trabajo a desarrollar en la herramienta MATE, también dispone de una serie de ventajas que aseguran su viabilidad. Ya que MATE es fruto de la tesis doctoral de una de las tutoras del proyecto, Anna Sikora, es posible realizar cualquier consulta de aspecto técnico. En cuanto a conocimientos necesarios para modificar el programa, se han cursado una serie de clases de máster en la UAB sobre las materias necesarias (DynInst y MPI).

3. Definición metodología

3.1 SQA

Un aspecto importante dentro del proyecto es la utilización de SQA, del inglés Software Quality Assurance. Se trata de un modelo sistemático y planeado de todas las acciones necesarias para asegurar la calidad esperada del producto final, así como la correcta aplicación de estándares y procedimientos adoptados. Es algo que se aplica durante todo el proceso de desarrollo, y se rige por el SQAP (Software Quality Assurance Plan), dónde se definen las actividades específicas a llevar a cabo dentro del proyecto. Dado el proceso de desarrollo de MATE, que se espera que continúe más allá del proyecto actual, es necesario establecer una serie de métodos que consoliden los objetivos marcados en cada etapa.

Según los modelos de ampliación de defectos, el coste de los fallos detectados en un producto software es mayor cuanto más tarde se detectan. Para ilustrar la reducción del coste con la detección anticipada de errores, podemos considerar una serie de costes relativos que se basan en datos de proyectos de software reales[2]. Suponiendo que un error descubierto en la fase de diseño del producto cuesta 1,0 unidad monetaria, este mismo error descubierto antes de realizar el proceso de test costará 6,5 unidades, durante las pruebas 15 unidades, y después de la entrega entre 60 y 100 unidades. El mismo razonamiento se puede aplicar a otros recursos de un proyecto como pueden ser tiempo o rendimiento. Es aquí donde reside la importancia de un buen proceso de SQA ya que nos permite tener un buen seguimiento durante todo el proyecto.

A la hora de desarrollar un SQAP dentro de un proyecto hay diferentes alternativas y modelos de los que escoger. Todos ellos listan una serie de aspectos importantes a tener en cuenta en el momento de evaluar la calidad del software. Una vez detectados estas cualidades críticas, se pueden cuantificar con una serie de métricas y así poder determinar la calidad actual del producto.

2 Defect amplification model [IBM81] "Implementating Software Inspections", Notas del curso, IBM Systems Sciences Institute, IBM Corporation, 1981

3.1.1 Modelos estándar

Por un lado podemos encontrar diferentes modelos estándar. Estos modelos son aplicables a cualquier proyecto y determinan el nivel de calidad con atributos generales. Además han sido utilizados en diferentes proyectos y por lo tanto se tiene una perspectiva de los resultados esperados.

Uno de los primeros modelos existentes y en el que se basan la mayoría de los actuales es el modelo de McCall, que, en un principio, fue creado para las fuerzas aéreas de los Estados Unidos en 1977. Principalmente está enfocado a los desarrolladores del sistema y al proceso de desarrollo. En este modelo McCall intenta unir la perspectiva de usuarios y desarrolladores, centrándose en unas características de la calidad del software que reflejan tanto la visión de los usuarios como las prioridades de los desarrolladores. El modelo presenta tres características para medir la calidad del software: revisión (habilidad para adoptar cambios), transición (habilidad para adaptarse a otros entornos) y operación (sus características operativas).

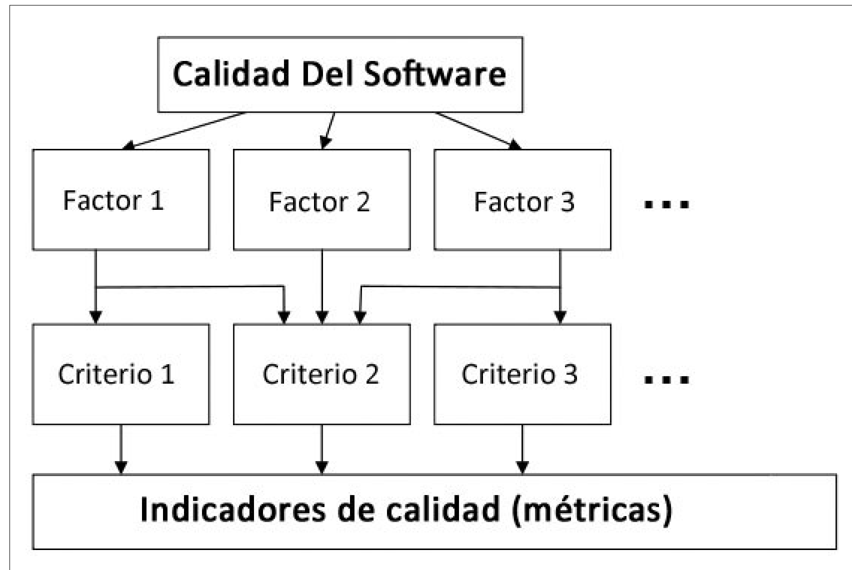


Figura 3.1 Jerarquía FCM de McCall.

El modelo también es llamado FCM (Factor, Metrics, Criteria) porque detalla tres tipos de características en una jerarquía (figura 3.1) dónde las más importantes se denominan factores. Por debajo podemos observar diferentes subcaracterísticas denominadas criterios y, finalmente, tenemos las métricas para determinar el nivel de satisfacción de cada una de estas subcaracterísticas.

Otro modelo a mencionar es el FMEA (Failure Mode and Effects Analisis). Como su nombre indica se basa en analizar problemas potenciales, principalmente en una época temprana del ciclo de desarrollo donde es más fácil tomar acciones para solucionarlos. FMEA se utiliza para identificar fallos potenciales en los sistemas, para determinar su efecto sobre la operación del producto, y para identificar acciones correctivas para atenuar las faltas. Podemos encontrar diferentes tipos siguiendo este modelo según su enfoque: Sistema (enfocado a funciones globales del sistema), Diseño (enfocado a componentes y subsistemas), Proceso (enfocado a procesos de fabricación y ensamblaje), Servicio (enfocado a funciones del servicio) y Software (enfocado a funciones del software).

Para terminar, los modelos estándar más seguidos son las normas ISO (International Organization for Standardization). Esta organización ha desarrollado una serie de normas y modelos para la evaluación de la calidad aplicables a productos generales, adaptándose en ciertos casos a la producción de software. En este modelo los conceptos de calidad se aplican más en el producto terminado que en el proceso de desarrollo. Estas normas hacen posible que se sigan patrones de calidad generalmente aceptados con los que se logran métricas para determinar las cualidades de un producto, teniendo en cuenta que en la práctica existen dos tipos de calidad: externa (referente a los clientes) e interna (referente a funcionalidad del software).

De todos los estándares que presenta, el más interesante para el proyecto es el ISO 9126, que está enfocado directamente a productos software. Este modelo está basado en el de McCall, ya que determina la calidad del software en base a una herencia de características. Como podemos observar en la figura 3.2, posee 6 características principales (factores siguiendo el modelo de McCall). Las 6 características principales son funcionalidad, fiabilidad, usabilidad, eficiencia, Mantenibilidad y portabilidad. Además de esto cada aspecto tiene diferentes subcaracterísticas a evaluar.

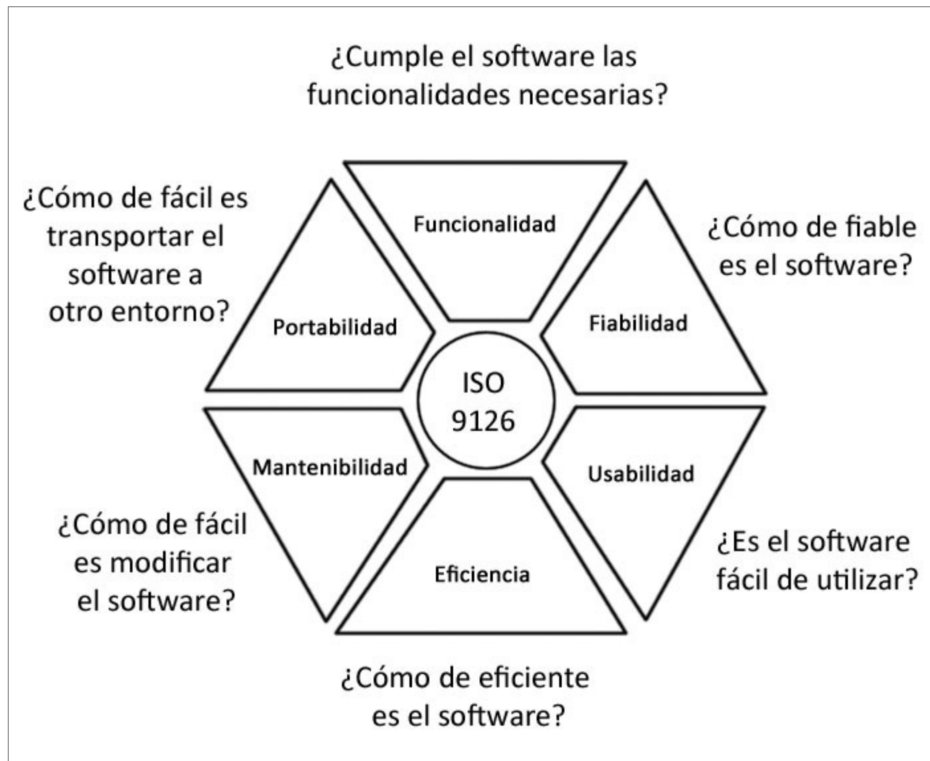


Figura 3.2 Modelo ISO 9126

Dentro del proyecto se da énfasis a los aspectos más importantes para el software. Por una parte dentro de fiabilidad es importante tener en cuenta la tolerancia a fallos y la facilidad de recuperación, en el proyecto cubre este aspecto añadiendo ciertas nuevas características al programa (sección 4.4) y realizando una serie de tests (capítulo 5). También se ha trabajado en el factor de usabilidad, concretamente en las subcaracterísticas de comprensibilidad y facilidad de aprendizaje. Esto se ha mejorado con la documentación generada para MATE y los diferentes elementos de soporte. El tema de la eficiencia es algo importante, tratándose de un software de aumento de rendimiento, y es algo que se ha acabado de definir consolidando todos los aspectos del programa en su estado previo. Finalmente el hecho de haber creado la metodología con diferentes guías y directrices de trabajo hace que el mantenimiento del software en el futuro sea mucho mayor y facilita el análisis y test del producto.

3.1.2 Modelos de aplicación o específicos de compañías

Además de los modelos estándar vistos en el capítulo anterior, existen los de aplicación específicos para compañías. Estos no hablan de características generales sino que detallan controles de calidad ajustables a necesidades particulares.

Uno de los que vale la pena mencionar es el llamado modelo CMMI (Capability Maturity Model Integration). Este modelo determina una serie de procesos y actividades a llevar a cabo para asegurar la calidad del producto, sin entrar en detalles de cómo realizar estas actividades. Según se van completando las actividades propuestas se dice que el proyecto está en diferentes niveles de madurez: incompleto (nivel 0), inicial, administrado, definido, cuantitativamente administrado y optimización (nivel 5). Dispone de una lista de áreas de acción, y en cada una de ellas se proporciona la lista de prácticas a llevar a cabo. Actualmente incluye de 22 áreas de proceso[3] que podemos observar en la Tabla 3.1.

CMMI® for Development, Version 1.2 (CMMI-DEV, V1.2)		
Causal Analysis and Resolution (CAR)	Configuration Management (CM)	Decision Analysis and Resolution (DAR)
Integrated Project Management +IPPD (IPM+IPPD)	Measurement and Analysis (MA)	Organizational Innovation and Deployment (OID)
Organizational Process Definition +IPPD (OPD+IPPD)	Organizational Process Focus (OPF)	Organizational Process Performance (OPP)
Organizational Training (OT)	Product Integration (PI)	Project Monitoring and Control (PMC)
Project Planning (PP)	Process and Product Quality Assurance (PPQA)	Quantitative Project Management (QPM)
Requirements Development (RD)	Requirements Management (REQM)	Risk Management (RSKM)
Supplier Agreement Management (SAM)	Technical Solution (TS)	Validation (VAL)
Verification (VER)		

Tabla 3.1 Áreas de proceso del modelo CMMI, versión 1.2

3 CMMI for Acquisition versión 1.2, CMMI for Developement versión 1.2 y CMMI for Services versión 1.2

El último modelo a comentar es el modelo GQM (Goal-Question-Metric). A diferencia de los modelos vistos anteriormente, este modelo no se basa en características generales de los productos, sino que es aplicable solamente al proyecto en concreto. Toma este enfoque con la idea de que un programa de mesura de calidad puede dar mejores resultados si se diseña con las metas en mente. Simplemente se trata de trabajar realizando tres pasos: crear una lista con las metas del proyecto, a partir de la lista generar unas preguntas que determinen si la meta se ha cumplido y finalmente decidir qué atributos hacen falta medir para responder estas preguntas.

3.2 Perspectiva general

Una de las tareas a realizar para este proyecto ha sido generar el soporte que necesita el proceso de desarrollo de un programa como MATE. Para definir una metodología de trabajo, se han estudiado diferentes alternativas sobre las herramientas a utilizar y se han visto los diferentes aspectos a cubrir con guías y especificaciones. Finalmente ha resultado en una serie de documentos que definen todo el proceso de desarrollo. Estos documentos no están cerrados y están abiertos a las modificaciones que sean pertinentes, pero resultan una buena base para el futuro avance de MATE.

3.2.1 Guías y especificaciones

El primer conjunto de documentos a generar para respaldar la metodología han sido unas guías de estilo y unos documentos de especificación. Estos elementos ayudan a mantener un formato uniforme durante todos los artefactos del proyecto y agilizan el proceso de desarrollo. Dada la naturaleza del proyecto, dónde los documentos están abiertos a revisiones hasta la finalización de este, ya se habrán tenido en cuenta diferentes problemas. Por eso ayudarán a futuros integrantes del equipo de desarrollo a no cometer los mismos errores.

El resultado consiste en 5 documentos donde se tienen en cuenta diferentes aspectos de la metodología a tener en cuenta:

- **Guía de estilo de documentación:** Esta primera guía trata de homogeneizar el formato de todos los documentos generados para esta metodología y en el futuro desarrollo de MATE. Este documento abarca diferentes aspectos a tener en cuenta a la hora de escribir documentación. Por un lado especifica el formato en que la información se establece, dando unas secciones comunes para todos los documentos y una serie de recomendaciones a seguir siempre que sea posible. Seguidamente detalla el formato común a seguir con los títulos de sección, el tipo de letra a utilizar, tipo de paginación, etc. Es la única guía que trata de algo que no sea puramente desarrollo.

- **Guía de estilo de documentación de código:** La documentación a la que se refiere esta guía es la generada a través del código, en este caso utilizando la herramienta doxygen. Para comentarios inline, dirigidos exclusivamente a desarrolladores, está la guía de estilo de codificación. En esta guía se detalla el uso de tags de doxygen, así como la manera en la que se deben listar los atributos y unas recomendaciones generales de redacción. También habla de qué partes es necesario documentar y cuales son meramente opcionales.

- **Guía de estilo de codificación:** En esta guía encontramos diferentes directrices a seguir a la hora de generar código de MATE. Es una guía orientada al lenguaje C++ y detalla diferentes aspectos importantes a la hora de mantener una homogeneidad en el código. Además de la propia codificación también comenta algo de la localización de los ficheros y algunas prácticas a seguir. Se puede encontrar más información sobre esta guía en la sección 3.4.

- **Especificación de deployment:** Este documento trata de especificar los métodos a seguir para obtener una versión final de la aplicación. Es una especificación orientada a la última fase del proceso de desarrollo. Después de haber trabajado de forma local con el producto, es necesario realizar una serie de pasos para poder utilizarlo de manera general. Para ello el documento detalla las fases a seguir (Release, activación y desactivación, modificaciones y desinstalación), las herramientas a utilizar y los módulos necesarios.

- **Especificación de control de versiones y build:** Esta especificación trata de establecer unas pautas a seguir para tener un buen control sobre el producto y sus versiones. Al tratarse de un proyecto desarrollado por diferentes personas, es importante tener un sistema para poder trabajar concurrentemente. Por lo tanto intenta detallar los mecanismos que se aplicarán para llevar un control de la evolución del proyecto, así como la construcción y tests de cada etapa.

3.2.2 Herramientas

Otro conjunto de artefactos a generar para la metodología son especificaciones y guías de instalación de las herramientas a utilizar. La importancia reside en utilizar siempre las mismas herramientas con el mismo entorno y configuración, ya que han sido probadas para el proceso de desarrollo inicial y han dado buenos resultados.

Las herramientas documentadas finalmente son las siguientes:

- **Buildbot:** Buildbot se trata de una herramienta de automatización de construcción. Es importante para comprobar que el producto actúa como se espera en diferentes entornos y bajo diferentes circunstancias. Para facilitar el uso de esta herramienta se han generado dos elementos. El primero es una guía de instalación y configuración. Incluye diferentes detalles sobre la configuración de la herramienta para utilizarla de la manera deseada en nuestro proyecto. Por otro lado también se ha escrito un script de instalación de Buildbot. Este script instala tanto Buildbot como sus dependencias de la manera deseada para trabajar con MATE. Se puede encontrar información más detallada sobre estos artefactos en las secciones 3.3.7 y 3.4.

- **SVN + Apache:** Para poder controlar las versiones se dispone de Subversion (abreviado como SVN). Se trata de una herramienta que permite el almacenamiento organizado de archivos y controla los cambios que se efectúan en ellos. Además permite acceso a los archivos de forma remota. Pese a no se ser una herramienta específicamente diseñada para el desarrollo software cumple las funciones necesarias y en nuestro caso nos ofrece un servidor común para almacenar y compartir el trabajo conjunto durante el proyecto. Además para disponer de una visión más cómoda podemos usar apache para que nos muestre los contenidos del repositorio en formato html.

- **Redmine:** Redmine es una herramienta dedicada a la gestión y planificación de proyectos con interfaz web. El uso de esta herramienta en el proyecto MATE es importante cómo vía de conexión de todos los elementos comentados anteriormente. Por un lado permite gestionar todo el desarrollo del proyecto con herramientas de planificación como pueden ser diagramas de Gantt, calendarios, foros, etc. También posee un soporte de usuarios con diferentes roles para gestionar el proceso de desarrollo con más eficacia. Por otro lado será el punto el front-end de nuestro sistema, ya que es capaz de integrarse con Buildbot y SVN, por lo tanto una vez realizadas las configuraciones necesarias, será el punto en el que todos los desarrolladores trabajaran.

- **Doxygen:** Doxygen es un sistema de documentación para código en varios lenguajes, entre los cuales se encuentran C++ y C, lo que nos permitirá usarlo en este proyecto. Este programa nos permite la creación rápida y cómoda de documentación a partir de una análisis exhaustivo del código. Además nos puede servir de asistencia a la hora de comprender la estructuración y funcionamiento de grandes cantidades de código pues tiene el capacidad de mostrar de forma esquemática los componentes de este. Con esto podremos generar una documentación externa al código, en formato html, para los usuarios de las librerías de MATE.

3.2.3 Elementos no documentados

Además de todos los elementos documentados con las guías y especificaciones, que se utilizarán durante todo el desarrollo de MATE, este proyecto en concreto ha utilizado ciertas herramientas de manera local que no se han documentado. La razón es que se han escogido por familiaridad y comodidad, pero cualquier herramienta con estas características puede cumplir los requisitos.

Por un lado el editor de código utilizado ha sido Eclipse Helios. Éste permite la compilación de todo el código de forma automática (con un makefile que es posible editar realizando ciertas configuraciones). Posee ciertos mecanismos de automatismo a la hora de documentar e incluso escribir código. También es muy útil ya que dispone de varios plugins que han facilitado las tareas a realizar en esta fase del proyecto.

Para realizar las tareas asignadas se han instalado dos plugins en Eclipse. El primero se denomina eclox, y permite trabajar de manera mucho más cómoda con doxygen. Con este plugin es posible generar la documentación del código en diferentes formatos como pueden ser html, pdf, etc. También se ha utilizado la librería graphicviz para generar automáticamente diagramas de clases y herencia.

El otro plugin utilizado ha sido ECUT, se trata de un plugin que integra el uso de CPPUnit con Eclipse y permite crear casos de uso y test de manera sencilla. También compila y ejecuta las pruebas mostrando los resultados de forma gráfica y analizando los resultados.

3.3 Buildbot

Como se ha mencionado anteriormente una de las herramientas utilizadas ha sido Buildbot, y dado el ámbito del proyecto ha sido necesario automatizar los procesos de testing. Por lo tanto Buildbot se trata de una herramienta necesaria y parte de la metodología a crear era una guía de instalación. En el momento de realización del proyecto la versión de buildbot era la 0.8.3, y tanto la guía adjunta como las especificaciones presentes están basadas en ella.

3.3.1 Breve descripción de la herramienta.

Buildbot es un sistema para automatizar el ciclo de construcción y test necesario en la mayoría de proyectos software para validar cambios. Construyendo automáticamente cada parte del proyecto cada vez que se realizan cambios se detectan rápidamente los posibles problemas, antes de que otros desarrolladores se vean afectados por el fallo. El desarrollador responsable puede detectarse sin la intervención humana y ser notificado. Realizando las construcciones en diferentes plataformas es más fácil ver el comportamiento del programa para desarrolladores que no tienen acceso a diferentes entornos. Además de eso se puede obtener información del programa para que sea más fácil de mejorar y detectar los puntos críticos. Buildbot es una herramienta gratuita.

Para realizar el ciclo de testing, Buildbot puede estar instalado en diferentes máquinas con entornos de interés, de este modo se podrán realizar diferentes pruebas automatizadas y contrastar los resultados en distintos escenarios.

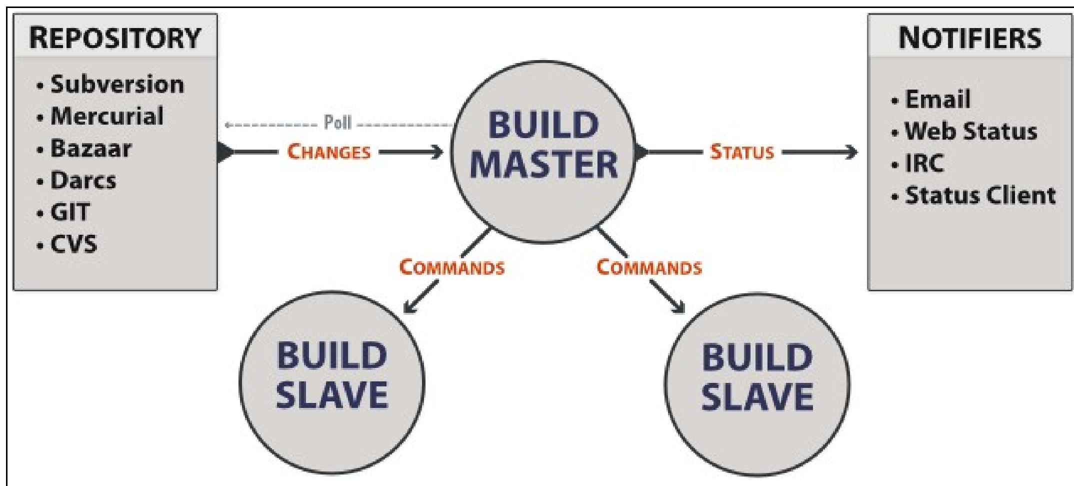


Figura 3.3 Esquema del sistema, extraído de la documentación oficial de Buildbot.

Buildbot dispone de dos tipos de agentes para realizar su función. El primero y más importante es el buildmaster (también llamado buildbot) y se encarga de manejar toda la estructura del programa. Éste observa el repositorio dónde se encuentran los ficheros del proyecto y detecta todos los cambios que se realizan cada vez que se actualiza. Una vez detectados los eventos notifica a través de diferentes canales a los desarrolladores pertinentes y manda las acciones a realizar a las diferentes máquinas del sistema. El segundo es el buildslave y podemos tener varios agentes de este tipo en diferentes máquinas. Son los encargados de realizar las acciones programadas por el buildmaster en sus máquinas locales y recoger los resultados. Se puede observar un esquema mostrando esto en la figura 3.3.

La configuración del buildmaster se almacena en el archivo master.cfg, y es ahí donde se realizan las especificaciones de los procesos de compilación, la cantidad de buildslaves, etc. Este fichero está escrito en python y la configuración se lee de una variable tipo diccionario, donde cada clave representa cada uno de los aspectos a configurar.

3.3.2 Configuración general

Cómo información general se registran cuatro tipos de entidades: los buildslaves, el repositorio utilizado, la identidad del proyecto y la base de datos.

Para registrar buildslaves en el buildmaster es necesario especificar dos claves en el diccionario. La primera es 'slaves' y es una array de objetos tipo BuildSlave, cada uno de ellos dispondrá de un nombre y una contraseña con la cual podrán registrarse diferentes máquinas e intercambiar notificaciones con el buildmaster. La otra clave necesaria es 'slavePortnum' y se trata de un entero que denominará el puerto asociado a la aplicación por el cual se comunicarán los dos agentes.

Los ficheros de nuestro proyecto estarán guardados en un repositorio, y el buildmaster se encargará de seguir su evolución y detectar los cambios. Para configurar la conexión con el repositorio se utiliza la clave 'change_source' y es un objeto del módulo buildbot.changes. Dependiendo del tipo de repositorio puede ser de la clase SVNPoller, GitPoller, etc.

Sobre la identidad del proyecto se pueden indicar diferentes cosas, como el nombre del proyecto (en la clave 'projectName') o la URL (clave 'projectURL'). Pero es imprescindible indicar el lugar dónde buildbot mostrará los resultados obtenidos y la información sobre estos. Esto se indicará en la clave 'buildbotURL'.

Finalmente es necesario especificar una conexión a una base de datos dónde buildbot guardará información sobre los resultados de las construcciones, estado actual, acciones pendientes, etc. Esto se indica en la clave 'db_url'.

3.3.3 Organizadores

Para poder realizar construcciones y pruebas sobre el código se dispone de diferentes organizadores (*schedulers*) que indican a los constructores (*builders*) cuándo deben realizar el proceso de compilación. Se configuran en la clave 'schedulers' y se trata de una lista.

Podemos hablar de diferentes tipos de organizadores:

- **Filtro de cambios:** Detecta cambios realizados en el repositorio filtrando los valores de cuatro atributos importantes (proyecto, repositorio, *branch* y categoría). Se realizará el proceso de construcción solo si todas las condiciones sujetas a estos atributos se cumplen. Por ejemplo, es posible que solamente queramos hacer pruebas con un proyecto llamado "ejemplo1" y en una *branch* llamada "branch3".

- **Organizador de *branch* única:** Este suele ser el tipo de organizador más utilizado. Escucha a una sola *branch* del repositorio y activa un temporizador cada vez que se realizan cambios. Cuando se realiza un nuevo cambio, el temporizador se reinicia y si llega al tiempo máximo establecido, indica a los constructores correspondientes que deben activarse.

- **Organizador de varias *branch*:** Funciona igual que el organizador de *branch* única pero observando diferentes *branch* de un mismo repositorio. Todas ellas tendrán un temporizador independiente del resto.

- **Organizador dependiente:** A diferencia de los demás organizadores, éste no presta atención al repositorio sino que realiza acciones dependiendo de los resultados de otras construcciones. Por lo tanto, activará sus constructores solamente después de haber realizado otros tipos de pruebas con éxito. Tiene sentido cuando se utilizan pruebas con diferentes partes del código y queremos realizar tests de integración.

- **Organizador periódico:** Simplemente activa los constructores cada x segundos, sin tener en cuenta los cambios realizados en el repositorio o los resultados de otras pruebas. Hay otros organizadores de este tipo para casos más concretos (por ejemplo el organizador nocturno).

- **Organizador "*try*":** Como particularidad este organizador puede activar constructores para partes de código que no han sido subidas al repositorio. Tiene este nombre porque permite utilizar el comando "*try"* de buildbot.

- **Organizador activado (*triggered*):** Se trata de un organizador que sólo realiza acciones cuando lo indica otro constructor. También puede indicarse que espere a terminar el proceso de construcción actual para activarse. La ventaja sobre el organizador dependiente es que es más versátil a la hora de asociarlo con otros constructores.

3.3.4 Constructores

Como se ha mencionado previamente los encargados de realizar el proceso de compilación y las pruebas deseadas son los constructores (*builders*). Cada uno se encarga de un tipo de construcción diferente y puede ser utilizado por varios buildslaves. Generalmente son independientes entre ellos, pero dependiendo de la configuración del sistema y los organizadores pueden existir dependencias. Además tienen asociada una fábrica de construcción, que genera instancias de construcción con versiones concretas del código.

Cada constructor se define en la lista en la clave 'builders'. Aquí se indicará el tipo de fábrica a utilizar (BasicBuildFactory, GNUAutoconf, BasicSVN, etc.) y las acciones (*steps*) a realizar en el proceso de construcción.

3.3.5 Informes de estado

Buildbot dispone de un sistema para notificar a todos los interesados de los resultados de las construcciones y cualquier acción realizada. Cada uno de estos métodos se denomina objetivo de estado (*status target*). Éstos se indican en la clave 'status' del diccionario y se pueden encontrar de los siguientes tipos:

- **Informe por web:** La información de los resultados se puede mostrar en forma de web con diferentes vistas. La más utilizada es la vista en cascada, que muestra los eventos en una línea de tiempo, con múltiples detalles de las diferentes construcciones y con links para ver más detalles como logs o código fuente. Existen otros tipos de vistas no tan utilizadas, cómo pueden ser en forma de cuadrícula o vistas con solo las últimas acciones.

- **Informe por email:** Otro es el sistema por email, Buildbot es capaz de notificar a los usuarios interesados y bajo las circunstancias deseadas. Un típico uso de esto es para notificar a los desarrolladores cuando algún cambio que han realizado ha causado algún error en las pruebas.

- **Agente IRC:** Este sistema crea un agente IRC que permite a los usuarios realizar peticiones de información.

- **Otros:** Existen otro tipo de objetivos de estado e incluso es posible crear plugins para diferentes canales utilizando la interfaz buildbot.interfaces.IStatus.

3.3.6 Guía de instalación y configuración

Como parte de la metodología a desarrollar se ha creado una guía de instalación y configuración de buildbot. Esta guía es un documento que espera utilizarse para establecer el entorno de desarrollo en el proyecto presente y los futuros relacionados con MATE.

La guía está formada por 4 capítulos básicos. El primero es una introducción al documento, parte común para todos los elementos de la metodología (como está indicado en la guía de estilo de documentación). A continuación el siguiente capítulo hace una pequeña descripción de buildbot para tener una perspectiva del programa, e introducir los aspectos básicos necesarios para saber utilizarlo. También habla de la versión a la que va dirigida la guía y la estructura que se espera tener. En el tercer capítulo se explica el proceso de instalación, tanto del buildmaster como de los buildslave, y se listan las dependencias necesarias para un correcto funcionamiento. Finalmente en el capítulo 4 se habla de la configuración de las diferentes características de Buildbot.

El propósito del documento es introducir el programa y dar una visión básica a los conceptos de Buildbot, ya que el uso esperado para el desarrollo de MATE es limitado y no es necesario un uso avanzado de la herramienta.

3.3.7 Script de instalación

Además de la guía también se escribió un script que instala automáticamente Buildbot y las dependencias necesarias. El script está escrito en bash y para funcionar debe estar instalado python.

La instalación la realizará en el directorio '/opt' ya que de esta manera tenemos el entorno buildbot y sus dependencias controlados y será más fácil realizar cualquier cambio necesario en el futuro. Además de esta manera se aumenta la compatibilidad entre sistemas y facilita la importación del entorno. Para conseguir esto el primer paso que realiza el script es descargar las fuentes del programa, éstas son las versiones presentes en cuanto se comenzó a desarrollar el proyecto y que actualmente pueden estar obsoletas. Todas ellas se sitúan en el directorio '/tmp' y se descomprimen. El siguiente paso se trata de la instalación de las dependencias y la librería de buildbot. Esto se realizará dentro de una carpeta que crearemos llamada 'MATE' y por lo tanto el directorio de instalación para todas ellas será '/opt/MATE'. Finalmente se eliminan las fuentes de '/tmp' y se crean enlaces dinámicos apuntando a las carpetas 'opt/MATE/bin' y '/opt/MATE/lib/python' desde carpetas existentes en los paths del sistema.

3.4 Guía de estilo de codificación

Otro de los elementos a desarrollar con la metodología ha sido una guía de estilo de codificación para el lenguaje de programación C++.

Este documento recoge un conjunto de normas y estilos que se utilizarán en la programación: indentación (sangrado), nombres de variables, elección de estructuras, distribución de los archivos, prácticas de programación, etc. De esta manera proporciona una coherencia entre los programas escritos por diferentes personas para facilitar la interacción y modificación de los mismos.

El uso de una guía de estilo presenta diferentes ventajas. En cuanto al proceso de desarrollo, favorece la colaboración del equipo ya que representa un formato común a seguir. También formaliza el modo de trabajo de manera que se mantenga una homogeneidad durante todo el proceso de desarrollo. Otro punto a tener en cuenta es la integración de nuevos miembros al equipo de trabajo, si el código es más legible facilita su entendimiento y por lo tanto acelera la incorporación. Dado el origen del proyecto MATE y sus características se esperan diferentes cambios de personal, por lo que este último aspecto acaba siendo esencial.

Por otro lado, observando el software como producto, un 80% del coste se debe al mantenimiento[4] y una guía de estilo lo facilita en gran medida. Además le da una visión de unidad al código final siguiendo una estructura y nomenclatura únicas. También lo provee de unas métricas de calidad de software y esto le aporta otro valor al producto. Finalmente una guía de estilo ayuda a que no se repitan errores pasados y favorece el funcionamiento en entornos no testeados.

3.4.1 Contenido

La guía consta de 8 capítulos dónde se definen varios aspectos del estilo de codificación.

El primer capítulo es una introducción al documento y habla de la estructura y alcance de este. Esta sección es común en toda la documentación de la metodología, y deja claro que se trata de una guía para el proyecto MATE.

4 "The benefits of Coding Standards", Nigel Cheshire y Richard Sharpe.

El segundo habla de normas a aplicar en cuanto a ficheros. Para este proyecto se decidió separarlos en archivos de cabecera (.h) y archivos fuente (.cpp). El contenido de los archivos de cabecera serán todas las declaraciones generales de clases, métodos, constantes, etc. También habla de algunos casos a tener en cuenta como el uso de los *namespace* o los métodos *inline*. En los archivos fuente se encontrará la implementación de los métodos presentes en el archivo de cabecera. En algunos casos, y con la debida justificación, estos archivos podrán omitirse. Finalmente habla de los nombres a utilizar y organización física.

Los capítulos del tercero al sexto forman el cuerpo de la guía y es dónde están las normas a seguir a la hora de escribir código. En el tercero encontramos indicadas las características generales. Básicamente habla del formato a seguir de codificación, sin entrar en ningún tipo de detalles en cuanto a escenarios concretos. Sobre esto último habla más el capítulo 4, el cual indica los formatos a seguir para casos particulares, como pueden ser asignaciones, llamadas a métodos, sentencias condicionales, etc. En el capítulo cinco se deja de hablar de formato para entrar en detalles sobre las declaraciones. Se habla de maneras de cómo declarar múltiples entidades en la misma línea y cuándo es posible hacerlo. Para acabar este bloque de la guía el capítulo 6 habla de la nomenclatura a seguir a la hora de nombrar clases, métodos, variables, etc.

Finalmente los dos últimos capítulos acaban hablando de casos particulares. El capítulo 7 introduce varias prácticas de programación a seguir siempre que sea posible, y el capítulo 8 habla de la manera de utilizar comentarios en ciertas situaciones. Aun así, no profundiza en el uso de comentarios ya que se han creado guías específicas para eso.

3.4.2 Evolución del documento

A pesar de empezar a trabajar pronto en la metodología no se pudo acceder al código hasta que los documentos sobre esta ya estaban acabados. Por lo tanto al ver el estado de MATE se realizaron cambios en la guía a fin de acomodarla al estilo previamente utilizado y de esa manera realizar solamente cambios necesarios y de diseño sobre el código. También al ser un código más complejo al habitual había diferentes aspectos que no se habían tenido en cuenta y por lo tanto se tuvieron que añadir diferentes detalles.

4. Trabajo en módulos Common y DMLib

4.1 Introducción a los módulos

Antes de entrar en detalle con el trabajo realizado en este proyecto, es importante conocer la estructura y funcionamiento de los módulos trabajados. Aunque se han realizado diferentes cambios y se han añadido nuevas características, el funcionamiento básico es el mismo y no ha cambiado.

4.1.1 Common

El primer módulo de MATE trabajado en este proyecto ha sido Common. Este módulo, a diferencia de los otros tres, no consiste en una entidad propia, no realiza ningún proceso que pueda reflejarse en un diagrama de flujo ni un orden de ejecución. Se trata de un paquete de clases que utilizan dos o tres de los otros módulos y, por lo tanto, resulta pertinente tenerlos aislados y así eliminar la redundancia de clases. Cómo base de funcionamiento para el resto de módulos, la importancia reside en el funcionamiento básico de las clases y por eso se han realizado una serie de casos de prueba (ver sección 5.2).

En este paquete podemos encontrar los siguientes archivos, que definen diferentes clases y métodos:

- **ActiveObject.h:** Este archivo contiene la definición de la clase ActiveObject. Se trata de una clase abstracta que encapsula un thread del sistema operativo de la librería POSIX. Se utilizará para crear los diferentes hilos del programa, y derivan de él las clases ShutDownManager y EventCollector.

- **Adress.h:** En este archivo encontramos también una sola clase, la clase Adress. Se trata de una clase que almacena y trabaja con direcciones de sockets de la familia AF_INET.

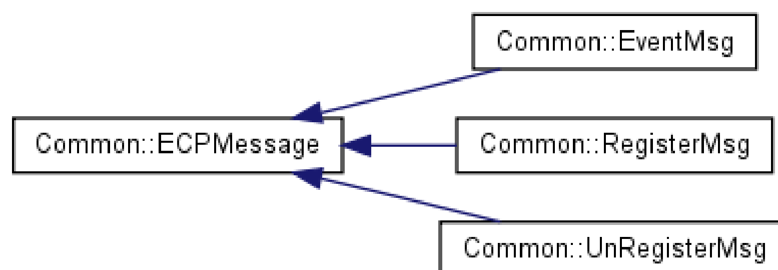
- **ByteStream.h y OutputStream.h:** En el archivo OutputStream.h se define la clase del mismo nombre. Se trata de una clase que representa un flujo de salida de datos en diferentes formatos. Es una clase abstracta y por lo tanto deberá tener clases derivadas. Podemos encontrar una de ellas dentro de Common, la clase ByteStream. Es una clase que encapsula un flujo de bytes.

- **Config.h y Configmap.h**: En un principio el archivo Config.h solamente contenía la clase Config, más tarde se ha incluido una nueva clase llamada ConfigHelper (ver sección 4.4.2). La clase Config es básicamente un diccionario que guarda información en base a secciones y claves. Se utiliza para aplicar diferentes configuraciones en MATE. Esta clase contiene un objeto de la clase ConfigMap, que implementa el diccionario mencionado que trabaja con secciones y claves.

- **ConfigReader.h:** Este archivo contiene archivos para trabajar con la lectura y carga de configuraciones. En un principio también contenía solamente una clase, ConfigReader, pero se ha modificado para tal de tener más versatilidad (ver sección 4.4.2).

- **Datetime.h y TimeValue.h:** La clase Datetime simplemente adapta la estructura tm declarada en la librería time.h con un uso más cómodo y funciones para trabajar en ellas. Dispone de una serie de setters para obtener los elementos de la estructura y el tiempo almacenado siempre será el de la máquina local al crear el objeto. Por otro lado la clase TimeValue adapta la estructura timeval, también declarada en time.h. A diferencia de DateTime esta clase tiene sobrecargados una serie de operadores para facilitar el manejo de estos.

- **di.h:** En este archivo encontramos diferentes clases que actúan como interface para la API de DynInst. Además de eso también dispone de ciertas acciones a realizar con DynInst encapsuladas en métodos, con la finalidad de tener un mejor control interno sobre las operaciones a realizar.



- **ECPMsg.h, ECPMsgHeader.h y ECPProxy.h:** Todos estos archivos están relacionados con el Event Collector Proxy. El objeto de esta clase se encargará de establecer una comunicación unidireccional con el Analyzer. De esta manera DMLib enviará los informes generados con meta data sobre los eventos de interés a través de diferentes mensajes.

- **Env.h y Paths.h:** El archivo Env.h solamente contiene un método llamado *ExpandPath*. Simplemente se encarga de expandir rutas que puedan contener variables de entorno o transformar rutas relativas en absolutas. Por otro lado, el archivo Paths.h tiene la función MakeDirectory, que se ocupa de crear un directorio con la ruta indicada.

- **Event.h, EventHandler.h, EventMap.h y EventMsgWritter.h:** Estos ficheros contienen la declaración de la clase Event y clases relacionadas. La clase Event se encarga de almacenar qué información deberá guardar el Analyzer sobre los eventos de interés.

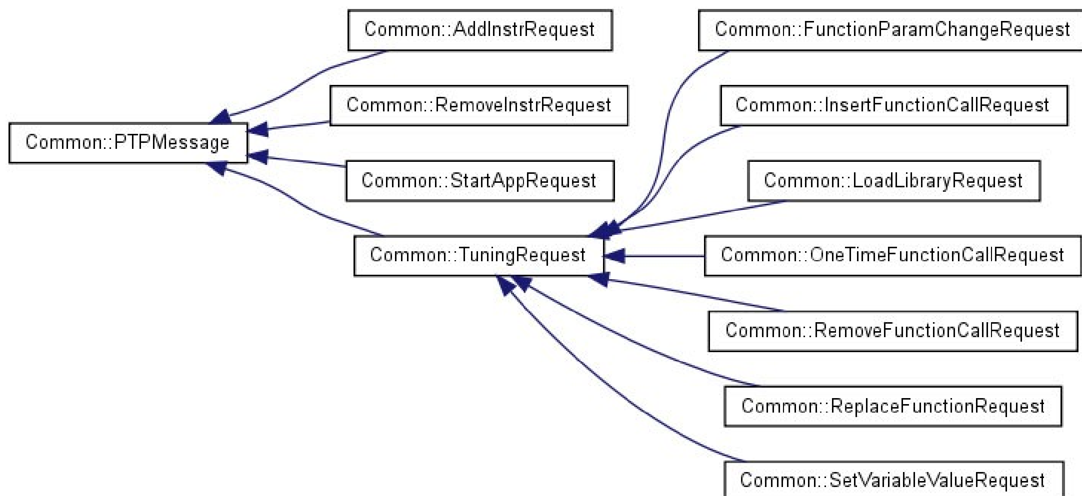
- **Exception.h y derivados:** El archivo Exception.h contiene la declaración de la clase Exception. Esta clase se encarga de almacenar información sobre algún error indicado por el programa. Para ello dispone de un código de error un mensaje indicando la causa. En un principio solamente existían la clase Exception y SysException, pero después de trabajar en los módulos se han añadido nuevos tipos (ver sección 4.4.1).

- **FuncDefs.h:** En este fichero encontramos las definiciones de las clases FuncDefs y FuncDef. La clase FuncDef contiene información de una función que se desea monitorizar en la función objetivo. La clase FuncDefs es simplemente una estructura de almacenamiento de objetos de la clase FuncDef.

- **NetSer.h y Serial.h:** Estos archivos contienen clases relacionadas con el proceso de serialización. Este proceso trata de transformar objetos y información de una máquina en un flujo de bytes para poder ser transmitido a través de una conexión. En el caso de MATE esta transmisión se realizará a través de Sockets.

- **Pipe.h:** Lo que se encuentra en este fichero es la clase Pipe. Esta clase trata de conectar la salida de un proceso a la entrada de otro, de tal manera que se cree una conexión y los datos se pasen automáticamente de uno a otro.

- **Process.h:** En este archivo se encuentran las declaraciones de tres clases. La primera es la clase abstracta Process, implementa la ejecución de un nuevo thread (utilizando la función *fork*) y ejecuta el método run(), a sobrecargar por las clases derivadas. Las otras dos clases son derivadas de esta y se denominan ExecProcess y RemoteProcess. ExecProcess realiza una llamada a un programa en la máquina local, pasando como valores de entrada la ruta y los argumentos. RemoteProcess ejecuta remotamente un programa en otra máquina, utilizando el programa Remote Shell (rsh).



- **PTPMsg.h, PTPMsgHeader.h y PTPProtocol.h**: Las clases que se pueden encontrar dentro de estos archivos están relacionadas con mensajes que intercambiarán el Analyzer y el sintonizador. Las siglas significan Protocolo de sintonización de rendimiento (Performance Tuning Protocol en inglés). Cada uno de estos mensajes consta de una petición al Tuner para realizar una de las siguientes acciones: petición de cargar librería, petición de sintonización, petición de asignación de variable, petición de reemplazo de función, petición de inserción de función, petición de ejecución de función, petición de eliminación de llamada a función, petición de cambio de parámetros, petición de inserción de instrucción, petición de eliminación de instrucción y petición de arranque de aplicación. Cada una de estas peticiones es una clase derivada de la clase padre PTPMsg.

- **Queue.h**: En este archivo vemos la declaración de la clase Queue. Se trata de una clase que implementa una estructura de datos utilizando FIFO (First Input First Output). Al utilizar un sistema de template se pueden guardar objetos de cualquier clase. Además tiene implementados sistemas de control de concurrencia con semáforos.

- **Reactor.h:** Este archivo contiene la declaración de la clase Reactor y otras necesarias para funcionar. Esta clase implementa el patrón de diseño del mismo nombre. Consiste en manejar peticiones de handles (identificadores de sockets) para diferentes procesos realizando peticiones de manera concurrente.

- **Socket.h:** En este archivo encontramos la declaración de las clases Socket, SocketBase y ServerSocket. Son simplemente interfaces para la librería sys/socket.h. SocketBase implementa sockets de cualquier tipo, con una serie de atributos por defecto. Aunque los métodos de la clase SocketBase están en un fichero fuente .cpp, los métodos de Socket y ServerSocket se encuentran junto a la declaración siendo tipo inline.

- **StringArray.h:** La clase StringArray se trata de una estructura de datos que almacena strings.

- **sync.h:** En este archivo encontramos declaraciones de elementos de sincronización para acceso concurrente. Contiene las clases Semaphore y Mutex, ambos garantizan exclusión mutua en el uso de recursos compartidos.

- **SysLog.h:** Dentro de SysLog.h vemos la declaraciones del log del sistema, así como varias clases relacionadas. Se trata de un registro de los eventos ocurridos en el sistema relevantes para el usuario, y dispone de entradas con diferentes niveles de importancia (debug, info, warning, error y fatal).

- **Thread.h:** En este archivo solamente encontramos la definición de la clase Thread. Define una interfaz para crear y manejar la ejecución de métodos en nuevos hilos.

- **Types.h y Utilis.h:** Estos dos archivos contienen diferentes elementos básicos que se utilizan alrededor de MATE. Entre estos elementos podemos encontrar declaraciones de tipos, enumerados y clases.

Además de esta descripción básica de los ficheros, se pueden encontrar más detalles en la documentación generada en html.

4.1.2 DMLib

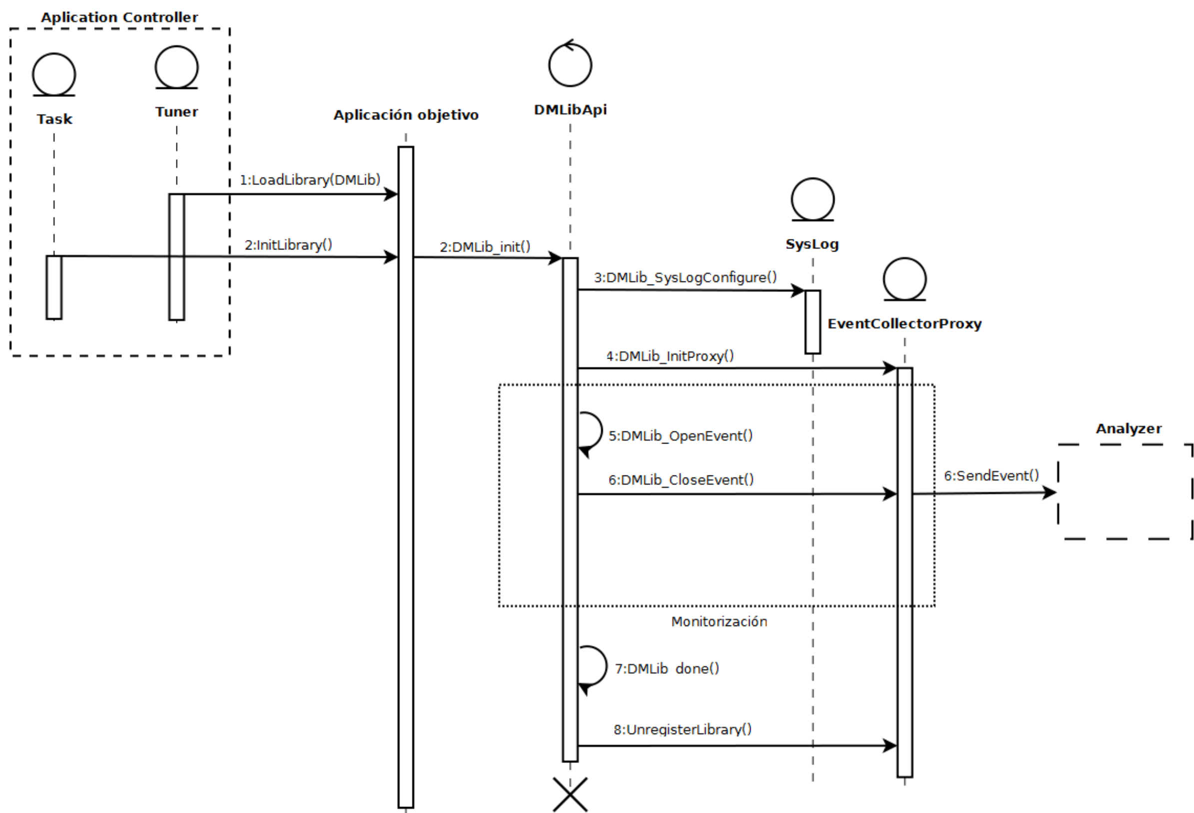


Figura 4.1 Diagrama de secuencia de DMLib

El módulo DMLib (Dynamic Monitoring Library) implementa la funcionalidad para monitorizar eventos y se compilará como una biblioteca compartida. En la librería podemos encontrar funciones para registrar eventos en el formato adecuado para entregarlos y ser analizados. En el diagrama de la figura 4.1 podemos observar los pasos necesarios para monitorizar los eventos. Primero el Tuner carga la librería dentro de la aplicación objetivo (utilizando el método BPatch_thread::loadLibrary() de Dyninst).

Esta librería posee las siguientes funciones públicas:

```
bool DMLib_Init (std::string const & taskName,
                 std::string const & analyzerHost,
                 int analyzerPort,
        std::string const & configFile = "DMLib.ini");
void DMLib_SetDiff (int lowDiff, int highDiff);
void DMLib_OpenEvent (int eventId,
                      InstrPlace instrPlace,
                       int paramCount);
void DMLib_AddIntParam (int value);
void DMLib_AddFloatParam (float value);
void DMLib_AddDoubleParam (double value);
void DMLib_AddCharParam (char value);
void DMLib_AddStringParam (std::string const & value);
void DMLib_CloseEvent ();
void DMLib_Done ();
```

La primera función, *DMLib_Init*, se encarga de inicializar la librería con la información del proceso a monitorizar, la localización del Analyzer y la configuración. Durante este paso, DMLib establece una conexión vía TCP/IP con el Analyzer para más tarde enviar información de los eventos monitorizados. Para corregir la diferencia del clock entre máquinas se utiliza la función *DMLib_SetDiff*. Esta nos permite mantener un clock global y poder trabajar con el mismo timestamp en los eventos generados.

Para empezar a monitorizar eventos, se empieza utilizando la función *DMLib_OpenEvent*. Esta función empezará inicializando todos los atributos necesarios: el identificador del evento y el número de atributos. El parámetro instrPlace índica en que parte de la función se va a guardar la información. Una vez se ha inicializado, se utilizan las funciones *DMLib_AddXXXParam*. Esta acción deberá hacerse el mismo número de veces que atributos se han indicado en el parámetro paramCount utilizado al inicializar el evento. Estos métodos permiten el registro de attributos del evento. Finalmente una vez se hayan definido todos los atributos, se utiliza la función *DMLib_CloseEvent* para cerrar el evento y enviar la información al Analyzer.

Para terminar, la función *DMLib_Done* se ocupa de cerrar la librería y liberar toda la memoria

ocupada. También se encargará de notificar al Analyzer de que ha acabado la fase de monitorización y cierra la conexión.

4.2 Documentación

Una de las mayores tareas a realizar sobre el código era documentarlo adecuadamente para facilitar el proceso de integración de futuros miembros del equipo de desarrollo. También generar una documentación para usuarios a modo de referencia. Dado el tipo de proyecto y entidades relacionadas se ha visto pertinente realizar los comentarios en inglés, ya que la nomenclatura del código ya estaba en este idioma y es el más portable.

Se pueden observar dos tipos de comentarios dentro del código, los comentarios *inline* y los comentarios doxygen.

4.2.1 Comentarios *inline*

Éstos están destinados únicamente a programadores con acceso al código fuente. Por lo tanto se trata de comentarios relacionados directamente con la implementación y el flujo de ejecución del programa. El formato a seguir es empezando con '//' y nunca utilizando '/* …. */'. Hay varias razones para utilizar este formato; por un lado utilizando asteriscos puede llegar a confundirse con comentarios doxygen, y por otro lado dificultan el comentario de bloques. Aquí podemos observar un ejemplo ilustrando este último caso:

```
/*
/* un comentario */
printf("queremos comentar esto");
/* otro comentario */
printf("y esto tambien");
*/ <--- no funciona!!
```

La mayoría de estos comentarios ya estaban presentes en el código inicial y no se han modificado, sin embargo se han añadido varios en partes dónde se veía necesario. Un buen ejemplo de su uso lo podemos encontrar en el método *AnalyzeLine* de la clase ConfigReader:

```
// ignore empty lines
if (line.size() == 0)
        return;
    // skip white chars
    int idx = 0;
    int len = line.size();
    while (isspace (line[idx]) && idx < len)
        idx++;
    // check if line is all white
    if (idx == len)
        return;
    switch (line[idx])
    {
        case '#': // comment - ignore
            break;
        case '[': // section
            ReadSection (config, &line[idx]);
            break;
        default: // key/value or error
            ReadKeyValue (config, &line[idx]);
            break;
    }
```

4.2.2 Comentarios doxygen

El otro tipo de comentarios añadidos son los referentes a la documentación externa de las clases y funciones. En éstos solamente se incluyen comentarios referentes al funcionamiento, así como explicación de parámetros, excepciones o cualquier dato que pueda serle útil a un usuario. Se encuentran íntegramente en los archivos de cabecera y siguen la guía de documentación de código generada con la metodología de trabajo. En la versión inicial no había ninguno de estos comentarios, por lo tanto se han tenido que realizar todos desde cero.

Doxygen es muy configurable y por lo tanto existen muchos formatos entre los que escoger, finalmente se ha seguido el formato '/** …. */' para cada bloque de comentarios y '@' para los *tags*. Se han escogido estos caracteres porque coinciden con el formato de JavaDoc, con el que ya se tenía familiaridad. Estos son algunos de los *tags* utilizados:

- **@class <nombre>:** Determina que los comentarios a continuación son referentes a la clase <nombre>.

- **@brief {descripción}:** Indica el comienzo de un parágrafo que se utilizará como descripción breve. Normalmente este parágrafo indica que tipo de información o comportamiento encapsula la clase. Si se trata de un método dará una breve descripción del comportamiento. El siguiente parágrafo será la descripción extendida.

- **@version {versión}:** Indica en qué versión se encuentra la clase.

- **@since {versión}:** Indica desde qué versión del proyecto la clase está presente.

- **@author {nombre}:** Indica el autor de la clase.

- **@extends <nombre>:** Determina que la clase hereda de la clase <nombre>.

- **@param <nombre>:** Da una descripción del parámetro con nombre <nombre>.

- **@return {descripción}:** Describe el valor de retorno del método.

- **@throws <nombre>:** Indica que el método puede lanzar excepciones del tipo <nombre>.

Aunque esos son los *tags* más utilizados también se han utilizado otros como @code, @endcode, etc. Con todo esto doxygen es capaz de generar la documentación en diferentes formatos y, utilizando GraphicViz, también genera diferentes diagramas de clases, herencia, dependencias, etc.

4.3 Cambios realizados

Otra de las tareas principales del proyecto ha sido realizar cambios necesarios en el código de MATE. Aunque la mayoría han estado enfocados a la estandarización, o a realizar ajustes para hacer el programa más compatible con otros entornos, también se han llevado a cabo diferentes modificaciones en algunas funcionalidades.

4.3.1 Uso uniforme de *string* y *char **

Uno de los problemas que presentaba el código inicialmente es que no tenía uniformidad en el uso de cadenas y carácteres. A fin de tener un formato común en todos los archivos fuente, se estandarizaron todos los usos de la clase *string* y *char **. Además, esto dio pie a diferentes modificaciones ya que utilizando una clase en vez de un tipo primitivo, se tenían muchas más facilidades para manipular los datos.

Aun teniendo en cuenta esto, un aspecto importante de este cambio es que en principio no puede causar ningún tipo de problema para el uso de las clases ya implementado en otros módulos. La razón es que en C++, para un argumento de tipo string, es posible pasar tanto una cadena literal como una array de caracteres y automáticamente se llamará a uno de los constructores de la clase. Podemos observar esto con el siguiente ejemplo:

```
void funcionDePrueba(std::string foo);
char * cad;
string str;
//Trabajamos con las variables cad y str
//Diferentes posibles llamadas de la funcion
funcionDePrueba("esta es una cadena literal");
funcionDeprueba(cad);
funcionDePrueba(str);
```

Como caso general se ha aplicado este cambio a todas las funciones que recibían cadenas de nombres o entidades introducidos por un usuario (paths del sistema, nombres de archivos, nombres de hosts, etc.). Las excepciones dónde se ha mantenido el formato de char * han sido cuando se utilizaba como buffer y cuando era necesario trabajar directamente con el espacio reservado en memoria.

4.3.2 Separación en namespaces

A la hora de desarrollar un nuevo producto software hace falta tener en cuenta diferentes aspectos para que no exista ningún tipo de problema a la hora de importar módulos de las librerías generadas, en este caso será necesario utilizar las librerías de MATE a la hora de desarrollar tunlets. Uno de estos problemas es resolver las posibles colisiones que puedan existir con nombres de clases o métodos. En nuestro proyecto se pueden observar dentro del módulo Common diferentes clases con nombres muy comunes y que pueden traer problemas (Exception.h, Queue.h, OutputStream.h, etc.).

Una manera de solucionar este problema es utilizando namespaces. Se trata de un sistema que se encarga de agrupar diferentes entidades bajo un nombre, y por lo tanto es posible tener diferentes clases y métodos con el mismo nombre presentes si utilizan diferentes namespaces.

Los dos namespaces creados han sido DMLib y Common. En el primero, DMLib, solamente se han incluido las clases ECPProxy y EventMsgWritter ya que son las únicas que se utilizan exclusivamente con DMLib. También se han puesto bajo este namespace los métodos del archivo DMLibApi.h, que se trata de la interface para controlar a DMLib. Por otro lado el namespace Common se ha aplicado al resto de clases, este es el más importante ya que es aquí donde encontramos clases con nombres conflictivos.

Con este cambio es necesario tener en cuenta que se tendrán que modificar los archivos utilizando Common y DMLib en MATE para poder acceder a las clases. Simplemente es necesario poner "using namespace xxxx;" cada vez que se vaya a utilizar una clase que se encuentra dentro de un namespace determinado, por lo tanto no es un cambio muy drástico que pueda afectar demasiado al resto de módulos.

4.3.3 Separación de archivos fuente y de cabecera

Para adaptar el código de MATE a la guía de estilo de codificación (ver sección 3.4) el cambio más notable ha sido el de separar diferentes clases en archivos de cabecera (.h) y archivos fuente (.cpp). Algunos de los archivos ya estaban separados de esta manera pero no de manera uniforme y era necesario mantener una homogeneidad.

A la hora de crear una clase o algún método, una buena práctica de codificación es separar las declaraciones y prototipos de las implementaciones. De esta manera en el archivo de cabecera podemos encontrar la interface de la clase y es aquí dónde se realiza toda la documentación necesaria y se pueden observar claramente todos los elementos de esta. Además de ser cómodo para el programador también ayuda a futuros desarrolladores ya que es más fácil estudiar una clase por el archivo de cabecera y, en caso de querer conocer detalles sobre la implementación, solo es necesario abrir el archivo fuente.

También es un método que generalmente da mejores resultados de compilación, ya que los métodos que contienen la implementación en los archivos de cabecera son implícitamente métodos inline. Esto significa que el compilador no creará referencias a una implementación cada vez que un archivo incluya una cabecera, sino que copiará el código múltiples veces, causando así uso de recursos innecesarios. Sin embargo hay ciertas excepciones dónde es útil declarar métodos inline, en nuestro proyecto se puede observar esto en las clases de sockets. Al ser un elemento muy utilizado dentro del programa es bueno declarar estos métodos como inline, ya que no contienen mucho código y por lo tanto dará más velocidad de ejecución sin malgastar mucho espacio.

4.3.4 Cambios puntuales

Además de los cambios generales mencionados anteriormente, se han realizado otros cambios más puntuales que resulta pertinente comentar.

Por un lado en la clase ByteStream se trabajaba inicialmente con un buffer pasado a través del constructor. Ya que C++ soporta morfología de constructores, se ha añadido otro constructor que solamente requiere el tamaño del buffer y crea uno interno de manera que el usuario no necesite preocuparse del control del buffer.

Otro aspecto que ha sido necesario cambiar es la implementación de ciertas funciones en el archivo di.cpp. En el estado previo de MATE, se trabajaba con una versión antigua de DynInst y por lo tanto ha sido necesario adaptarlo a la nueva API. El ejemplo más claro es en el método *GetLineNumber* de la clase DiProcess.

Previamente este método se utilizaba de la manera siguiente:

```
BPatch_Vector< BPatch_statement > lines;
if (!(_bpProcess->getSourceLines (addr, lines)))
     throw DiEx ("Could not get information about
                    the line number from process");
```

Sin embargo, la función getSourceLines se ha marcado como *deprecated* y por lo tanto no es posible utilizarla. Esta función pretendía retornar el archivo y el número de línea dónde se encuentra un proceso. Por lo tanto se ha modificado la implementación previa por una búsqueda manual de los datos necesarios:

```
BPatch_Vector< BPatch_statement > lines;
if (_bpProcess->getSourceLines (addr, lines)){
    line = lines[0].lineNumber();
    for (int i = 0; i < length; i++){
        fileName[i] = lines[0].fileName()[i];
    }
} else {
    throw DiEx ("Could not get information about
             the line number from process");
}
```

Sobre la API de DMLib, se han añadido diferentes características como se menciona posteriormente. Sin embargo un cambio puntual a mencionar es un nuevo parámetro que se ha añadido. En la versión inicial se asumía que el archivo de configuración se denominaba DMLib.ini y no se podía cambiar la ruta. Como mejora para el uso de la interfaz ahora es posible especificar la ruta en un nuevo parámetro llamado configFile. Este cambio no solo resulta en una mejora de la interfaz, además no afecta el resto del programa ya que tiene un valor por defecto que es el mismo que antes y no será necesario darle un valor en todas las llamadas.

El último y quizá más importante de estos cambios ha sido la refactorización de las clases referentes a ECPMessage. Inicialmente existían colisiones con Common y Analyzer, ya que los tres tipos de mensaje y el header estaba definido en los dos módulos. Dada la finalidad de Common, se ha removido la declaración de estas clases del módulo Analyzer y ahora solamente se pueden encontrar en el módulo Common. Además se ha cambiado el reparto de las definiciones, se han juntado la clase ECPMsg.h y derivadas en el mismo archivo, llamado ECPMsg.h.

4.4 Desarrollo de nuevas Características

Mientras una parte del proyecto consistía en aplicar la metodología creada previamente a MATE (ver capítulo 3), también contenía parte de desarrollo de nuevas funcionalidades.

4.4.1 Sistema de excepciones

Un elemento importante en todo producto software es la capacidad de detectar los errores de la mejor manera posible y poder notificar al usuario correctamente. La importancia del tratamiento de excepciones no reside solamente en la notificación para el usuario, ya que muchas de ellas son casos extremos que rara vez llegarán a ojos del usuario, sino que es algo esencial para la implementación de nuevas características.

En C++ las excepciones pueden ser cualquier variable, ya que al utilizar la palabra reservada throw es posible pasar como argumento cualquier elemento. En el estado inicial del código podíamos encontrar dos tipos de excepciones.

Por un lado podíamos encontrar cadenas de carácteres como excepciones:

```
throw "mensaje de error";
```

Este uso del sistema de excepciones es bastante pobre ya que, aunque para el ojo humano dé información, a la hora de programar es imposible realizar un buen tratamiento de excepciones. Además de la causa, el programa tampoco puede determinar qué tipo de excepciones va a capturar y por lo tanto es más difícil continuar con la ejecución de la aplicación.

Por otro lado podíamos encontrar la clase Exception:

```
throw Exception("mensaje de error");
```

Esta forma de utilizar el sistema de excepciones es mejor, ya que al tener el soporte de una clase es posible realizar más controles y guardar los datos necesarios en cada objeto para que la aplicación tenga toda la información necesaria para tratar el error. Sin embargo sigue sin resolver el segundo problema, solo tenemos un tipo de excepción y, aunque podamos saber de qué se trata después de capturarlo, no podemos realizar una discriminación previa.

4.4.1.1 Nueva jerarquía de excepciones

Después de observar lo comentado en el apartado anterior, se decidió implementar una nueva jerarquía de excepciones para poder realizar un mejor tratamiento.

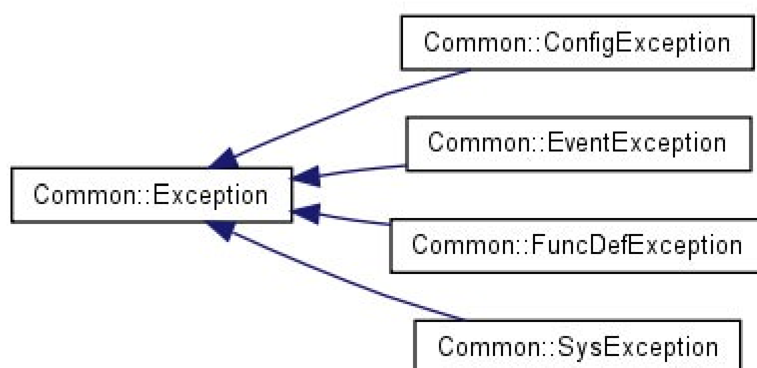


Figura 4.2 Diagrama de clases de la nueva jerarquía de excepciones.

Como se puede observar en la figura 4.2, la clase padre es Exception. Esta clase ya existía previamente y, por lo tanto, no será necesario cambiar código en otros módulos para adaptarlos a los cambios realizados. Sin embargo por las diferentes razones mencionadas en el apartado anterior es recomendable realizar un buen tratamiento de excepciones y tener en cuenta la causa del error. Las nuevas clases de excepciones son para diferentes situaciones dentro de Common, pero en un futuro también es posible implementar nuevos tipos heredando de la clase padre.

Esta clase, Exception, posee tres atributos de tipo *protected*. Esto significa que solo pueden acceder a ellos métodos de esta clase o derivadas. Por lo tanto las diferentes implementaciones podrán utilizar estos atributos como base. Los atributos son: _err, que contiene el código del error (normalmente se trata de errno, dentro de errno.h); _msg, una string con el mensaje de error para mostrar al usuario; y _objName, también una cadena que se puede utilizar para denominar la entidad que ha causado el error, en cada tipo de excepción se utilizará diferente. Además de estos atributos también tiene definidos dos constructores y tres métodos tipo *getter*. Las clases derivadas de esta deberán implementar los métodos para mostrar el mensaje, llamados Display().

El resto de excepciones implementadas son:

- **FuncDefException:** Se utiliza esta clase de excepciones en los objetos tipo FuncDefs, una clase encargada de almacenar definiciones de funciones a monitorizar. Los usos actuales en el código de MATE son referentes a la búsqueda y inserción de definiciones a la clase. En un futuro proceso de desarrollo estas excepciones podrían utilizarse para un mejor control de la monitorización.

- **ConfigException:** Esta clase de excepciones se utilizan en todos los elementos de la lectura de configuraciones. Indican que ha habido un error intentando localizar la fuente o en el formato de la configuración. Normalmente el atributo _objName se utiliza para el elemento que está dando problemas, cómo puede ser el path del archivo proporcionado (en el caso del FileConfigReader) o la línea de la configuración dónde se encuentra el error de formato. También aparece a la hora de intentar leer algún elemento no existente de la configuración, y el nombre suele ser la clave a la que se intenta acceder.

- **EventException:** Este tipo de excepciones está pensado para ser utilizado en todas las acciones referentes a la captura de eventos dentro de MATE. En el módulo trabajado en éste proyecto, Common, solamente se utiliza en la clase EventMap y tiene un uso similar al comentado anteriormente para las configuraciones. Al intentar acceder a elementos no existentes del mapa de eventos, se lanzará una excepción indicándolo. En un futuro esta clase podría ser utilizada alrededor de MATE ya que la captura de eventos es una parte importante del proceso de sintonización.

- **SysException:** Las excepciones del sistema (system exceptions) son las más generales y utilizadas en Common. Se tratan de diferentes tipos de errores referentes con el flujo interno de MATE y que, en principio, no están causados por agentes externos. Actualmente las clases que lanzan excepciones de este tipo son Address, Pipe, Process, Reactor, Socket, Sempahore, Syslog y Thread. Este marco tan amplio parece contradecir los motivos explicados previamente (sección 4.4.1), pero está justificado por el motivo de que el tratamiento de estas es más interno y dependiente del contexto.

4.4.1.2 Detección de errores en DMLib

Inicialmente el módulo DMLib contenía un control de errores básica, pero después de añadir el nuevo sistema de excepciones había varias excepciones no controladas. Por lo tanto era necesario definir un nuevo sistema que tratara estas excepciones y notificara correctamente al usuario.

Como hemos visto anteriormente (sección 4.1.2) DMLib realiza tres diferentes acciones al iniciarse. En el estado inicial del software la función *DMLib_Init* simplemente llamaba a tres funciones que realizaban esta acción sin ningún tipo de retorno. Para tener un control más completo la mayoría de funciones que antes eran tipo void ahora devuelven un booleano, indicando si la operación ha tenido éxito o no.

En la primera operación, iniciar el log del sistema, los errores se notifican por la salida estándar. Aunque esto no sea una buena práctica, ya que no quedará constancia de los errores iniciales en el log, es inevitable ya que todavía no estará iniciado. Todos los mensajes de error mostrados son directamente los asociados a las excepciones lanzadas.

El siguiente paso es iniciar la variable de MPI necesaria, si hay algún error en esta fase ya se notificará en el log del sistema, con un nivel de gravedad de "error". Ya que este error no viene asociado a ninguna excepción el mensaje proporcionado está escrito directamente en la función que realiza esta acción.

Para terminar se inicia la conexión de DMLib con el analyzer. Esta vez se vuelve a utilizar el log del sistema con un nivel de gravedad de "error" si sucede algún problema. También volvemos a disponer de excepciones y por lo tanto los mensajes se generarán automáticamente con el tratamiento de las excepciones.

4.4.2 Sistema de configuración

Para todo programa destinado a un uso personal, es importante ser capaz de añadir cierto nivel de configuración propia. En MATE esto tiene especial importancia ya que está pensado para ser ejecutado en diferentes entornos y bajo varias situaciones. Por lo tanto una parte que merece cierto hincapié es la capacidad del programa para aceptar configuraciones externas y la modularidad para facilitar la inserción de nuevos parámetros a tener en cuenta.

Inicialmente el sistema implantado para leer configuraciones consistía en una sola clase, llamada FileConfigReader y que contenía toda la implementación para leer configuraciones en el formato adecuado. Esto era una buena solución, ya que en principio no era necesaria la capacidad de poder leer configuraciones en otros formatos. Sin embargo en esta fase del desarrollo, dónde se pretende hacer a MATE más modular y versátil, un cambio en el diseño del sistema de configuración ha sido algo útil y necesario.

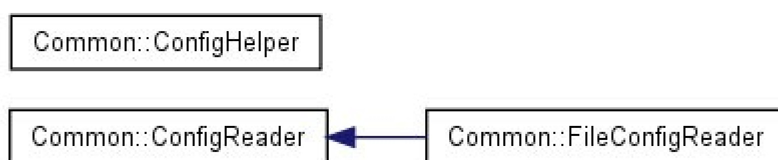


Figura 4.3 Diagrama de clases del nuevo sistema de configuración.

En el nuevo diseño que podemos observar en la figura, los detalles del formato ya no están ligados a la clase *FileConfigReader*, sino que se encuentran en la implementación de la clase abstracta *ConfigReader*. Esta clase contiene diferentes métodos privados que contienen la lectura de la configuración por secciones (*ReadSection* y *ReadKeyValue*). Además de estas posee un método protegido que será el que las clases derivadas utilizarán para analizar cada línea de texto, este método se denomina *AnalyzeLine*. Finalmente, todas las clases derivadas deberán contener los métodos necesarios para leer de cada tipo de elemento de entrada, pero deberán implementar el método *Read()*. Ésta función debe realizar la lectura línea a línea utilizando el método *AnalyzeLine* y finalmente retornar un objeto de tipo Config con la información parseada correctamente. Aunque este nuevo sistema permite la lectura de diferentes tipos de entradas, se ha mantenido la lectura de archivos y no se ha añadido ninguna, esto se realizará en la siguiente fase de desarrollo de MATE.

Además de el nuevo conjunto de clases derivadas de *ConfigReader* también se ha creado una clase estática llamada *ConfigHelper*. Esta clase dispone de una agrupación de métodos para trabajar con las configuraciones y utilizan diferentes tipos de elementos. Actualmente solamente está implementada la habilidad de leer la configuración desde un archivo. En el futuro todas las capacidades de MATE para trabajar con configuraciones, ya sea para leer o guardar, estarán incluidas en esta clase.

5. Testing

5.1 Unit Testing y ECUT

Como ya hemos visto en el capítulo 3 es importante asegurar que además de cumplir los objetivos establecidos, el resultado tenga la calidad esperada. Además, tratándose el módulo trabajado (Common) de el fundamento sobre el que funciona MATE, las funcionalidades básicas tienen una importancia adicional.

Para asegurar el nivel de calidad deseado, se han realizado una serie de casos de prueba unitarios. Se trata de comprobar el funcionamiento de componentes discretos del producto final. Las pruebas deben ser independientes entre ellas y esto servirá para comprobar el funcionamiento unitario antes de realizar tests de integración o de sistema. Normalmente estas pruebas tratan de unos test controlados dónde se simula el contexto real en el que deberán trabajar las clases. En estos contextos se provee una seria de inputs conocidos y se miden los outputs obtenidos. Posteriormente se comparan estos resultados con los outputs esperados y de esta manera se determina el éxito o fracaso de cada prueba.

Aunque estos test necesitan ser independientes de las clases, existen componentes que necesitan otros elementos del sistema para poder funcionar. Cuando son necesarios, se utilizan lo que se denomina "*stubs*" y "*drivers*". Por un lado, los stubs se tratan de simulaciones de sub-unidades que se necesitan para poder trabajar. Por otro lado los drivers son componentes de los cuales la clase probada forma parte.

Para llevar a cabo la implementación de estas pruebas se ha utilizado la librería CPPUnit. en esta librería podemos encontrar una clase llamada TestFixture y que utilizaremos para realizar las pruebas de cada clase. Cada clase de test derivará de la clase TestFixture y para cada aspecto que queramos probar se creará un método. Existen dos métodos especiales, *setUp*() y *tearDow*n() que se ejecutarán antes y después de la ejecución de cada prueba respectivamente. Dentro de cada caso de prueba se utilizan funciones para comprobar si el resultado es el esperado, como por ejemplo CPPUNIT_ASSERT() o CPPUNIT_ASSERT_EQUAL(). Una vez implementados los métodos, se llama a cada uno de ellos con la función CPPUNIT_TEST().

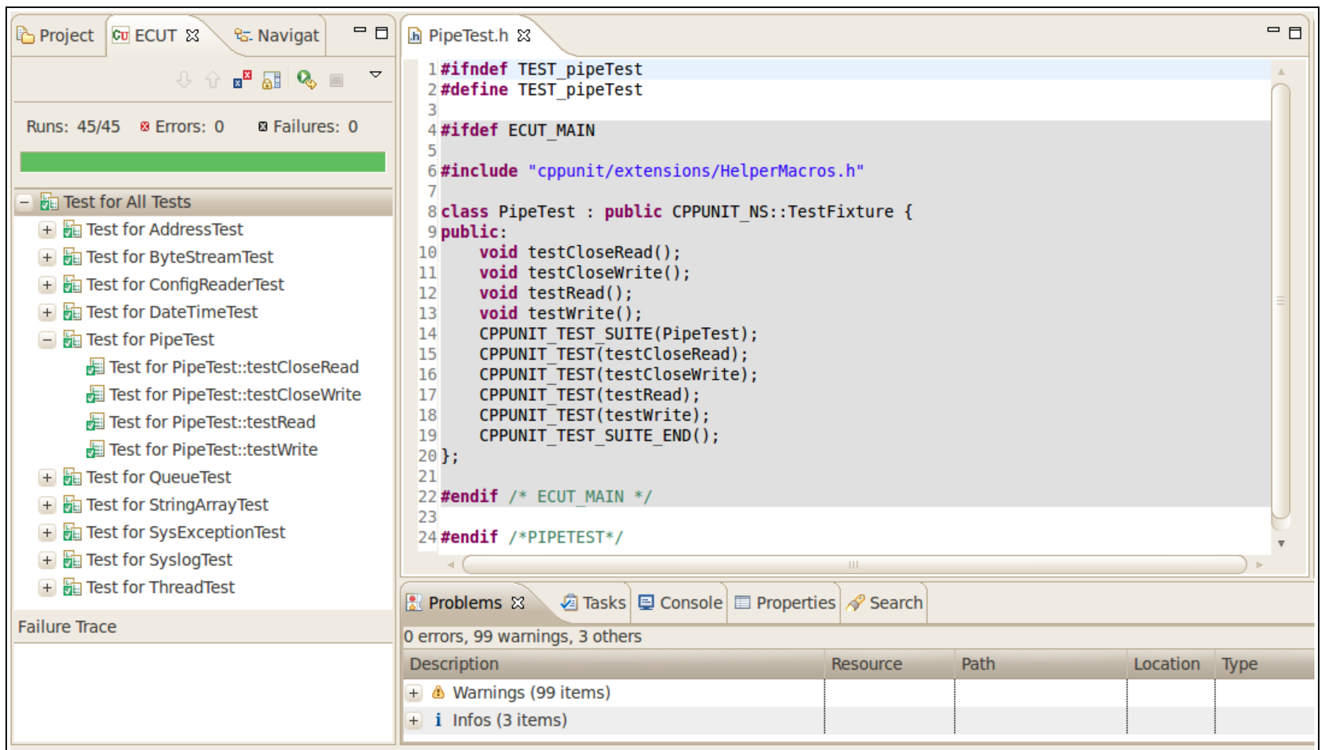


Figura 5.1 Interfaz del plugin ECUT en Eclipse.

Cómo asistente para estos casos de prueba se ha utilizado el plugin ECUT para Eclipse. Con este plugin es posible crear las clases automáticamente y asociar cada prueba a métodos concretos de cada clase. Además en el momento de ejecutar los casos de prueba muestra los resultados de forma gráfica y proporciona datos adicionales de interés, como pueden ser el tiempo de ejecución o la herencia de los tests. En la figura 5.1 podemos observar como muestra estos resultados.

5.2 Casos de prueba realizados

El resultado final consiste en 10 casos de prueba unitarios, cada uno tratándose de una unidad del módulo Common que se han considerado importantes. También se dispone de una carpeta con archivos auxiliares llamada "objects". En esta carpeta encontramos archivos de configuración necesarios para el funcionamiento de algunas clases.

Clase Address

Ya que esta clase se trata de una clase muy básica, que simplemente almacena información sobre direcciones de socket, las pruebas realizadas son bastante simples.

La primera prueba trata de comprobar que el tratamiento de tamaño de las direcciones es correcto. Ya que la tiene sobrecargado el cast a (sockaddr_in) es posible utilizar la función sizeof() para averiguar cual sería el resultado correcto. Lo comprobamos con los tres tipos de constructores posibles y así también se asegura el correcto funcionamiento de estos.

En el segundo test simplemente comprobamos que el método GetHostName() funciona correctamente. Para eso solamente es necesario comparar la cadena de retorno con la cadena "localhost" y ver que el resultado es el mismo.

Clase ByteStream

Los casos de prueba realizados en esta clase comprueban diferentes aspectos de la clase, realizando pruebas unitarias sobre los métodos más relevantes.

Por un lado comprueba los constructores, viendo que el apuntador al buffer interno es el correcto. También comprueba que el nuevo constructor (ver sección 4.3.4) funciona correctamente y reserva el espacio adecuado.

Otro aspecto comprobado en el test es la correcta habilidad para escribir y acceder a los datos. Primero realiza pruebas de la función Write(), introduciendo dos sentencias y comprobando que al obtener el contenido las lee correctamente. Seguidamente comprueba que aunque el buffer interno tenga las mismas características, no lo considere como igual al utilizar la función GetData(). Y para terminar utiliza la función GetDataSize() para comprobar la correcta mesura del tamaño.

Para acabar encontramos la prueba de la función Reset(). Esta función debe dejar el buffer vacío, y lo comprobamos empezando con datos en el buffer y observando que al resetear nos lo encontramos vacío de nuevo.

Clase ConfigReader

Ya que esta clase a sido modificada (sección 4.4.2) es necesario realizar pruebas para asegurarnos de que no contienen ningún bug que pueda afectar el producto final.

Dada la simplicidad de el resultado básico que debemos obtener, solamente es necesario realizar tests sobre el método Read(). Aunque externamente de un resultado muy simple esto aseguro que la funcionalidad de lectura y de localización de archivos funciona correctamente. Primero realizamos una lectura de un archivo que no existe y comprobamos que, efectivamente, se lanza una excepción indicándolo. Después de esto leemos un archivo con el formato adecuado y comprobamos que el contenido se ha leído correctamente.

Clase DateTime

En esta clase encontramos diferentes tests debido a la cantidad de funcionalidades que ofrece la clase. Aún así todos ellos son simples getters que muestran el correcto estado de los datos dentro del objeto.

Podemos observar que disponen de un setUp() inicial que se ocupa de guardar la fecha actual tanto en el objeto a testear como en una string literal. El resto de funciones comprueban el correcto acceso y conversión al año, mes, día, hora, minuto y segundo.

Clase Pipe

La funcionalidad básica de esta clase es simple y, por lo tanto, es posible asegurar su funcionamiento realizando solamente dos tipos de pruebas.

Las primeras van dirigidas al control del flujo de datos. Para controlar si es posible escribir o leer de alguno de los extremos del Pipe, se utilizan las funciones *CloseWrite*() y *CloseRead*(). Las pruebas de estos métodos simplemetne comprueban que los canales están abiertos y una vez cerrados están, efectivamente, cerrados.

Los otros casos de prueba ya tienen más interés ya que comprueban la correcta lectura y escritura de datos si los canales están abiertos. Para poder comprobar el resultado y simular mejor un caso real, se realiza un fork() para comunicar dos threads a través del pipe. Una vez creado el escenario ficticio simplemente se pasan datos a través del pipe y se comprueba que llegan correctamente al otro extremo.

Clase Queue

Como ya hemos mencionado previamente (sección 4.1.1) esta clase es capaz de almacenar objetos de cualquier clase. Para facilitar el proceso de obtención y comparación de objetos, se ha utilizado la clase string.

Las primeras pruebas que podemos ver son referentes a las comprobaciones de límites de la cola. Primero comprueba con una cola vacía que el método *IsEmpty*() retorne un resultado positivo, a continuación introduce datos para comprobar que lo indica correctamente y para terminar volvemos a vaciarla. El otro test es similar, ya que utiliza la función *IsFull*() para determinar lo contrario. También se realizan varias situaciones para comprobar que funciona en diferentes escenarios.

Por otro lado se asegura el funcionamiento del control interno viendo el correcto funcionamiento de las funciones *GetMaxSize*() y *GetCount*(). Mientras el primer método es comprobable directamente después de la inicialización del objeto, con el segundo es necesario realizar una comprobación más a fondo realizando la comparación cada vez que insertamos un nuevo elemento.

Finalmente se acaba comprobando las funciones de inserción y extracción. Los métodos *Get*() y *Put*() se prueban realizando inserciones y comprobando que el resultado retornado al extraer el objeto es el mismo. También se comprueba a parte la función *GetB*() que realiza una extracción utilizando mecanismos de exclusión mutua.

Clase StringArray

En los casos de prueba unitarios de esta clase comprobamos las funcionalidades básicas para manejar la estructura de datos.

Primero se comprueban las funciones *Add*() y *Get*(). Para asegurarnos de que tanto el orden como cantidad de strings guardadas es correcto, creamos un vector de C++ como estructura auxiliar. Seguidamente introducimos las mismas strings que tenemos en este vector en nuestro objeto y posteriormente comprobamos que el estado es el esperado. Otro test realizado es sobre la función *Grow*() que incrementa la capacidad de almacenamiento de el objeto.

Las otras pruebas realizadas giran en torno al acceso a atributos internos del objeto. La primera simplemente comprueba que el tamaño de strings almacenado es el correcto y la segunda determina que el acceso a la estructura interna que almacena los objetos da el resultado esperado.

Clase SysException

Para las clases de excepción solamente se ha realizado el test en una, SysException. Esto se debe a que todas ellas comparten la misma estructura interna y funcionamiento, por lo tanto realizar pruebas en una de ellas es suficiente.

Las primeras dos funciones simplemente comprueban que el mecanismo de tratamiento de funciones try... catch funciona correctamente. Esto se comprueba lanzando excepciones con los dos diferentes constructores y recogiendo el resultado. El método restante comprueba el correcto almacenamiento y tratamiento de códigos de error. Esto lo hace comprobando que un código de error conocido retorna el mensaje esperado.

Clase Syslog

Las pruebas realizadas a esta clase resultan ser las más extensas, dada la cantidad de métodos y funcionalidades de las que dispone.

Para empezar se realizan pruebas de los *getters* básicos. El primero se trata del timestamp, el momento en el que se realiza la inserción de la entrada en un objeto del tipo DateTime. Seguidamente se comprueban los métodos *GetSeverity*() y *GetMessage*() comparando los resultados dados con los esperados.

Por otro lado se realizan pruebas sobre los métodos de control sobre el log. Primeramente se comprueba con varios tipos de entradas que la función *Accept*() de el resultado correcto dependiendo del nivel de severidad del mensaje. A continuación se prueba la inserción de entradas en tres tipos de loggers. StreamLogger con entrada directa de caracteres, FileLogger que inserta los resultados en un fichero y SysLog que va imprimiendo por pantalla los mensajes con información. Para acabar con esta parte se comprueba el funcionamiento de la inserción de loggers dentro del log del sistema.

La última parte que se comprueba es la muestra de información con los métodos *ShowTimestamp*() y *ShowSeverity*(). Estos dos métodos se comprueban leyendo las cadenas que dan como resultado y buscando si los elementos tratados se muestran o no.

Clase Thread

Realizar pruebas sobre esta clase es importante, ya que MATE trabaja con Threads constantemente para realizar las tareas de cada módulo.

En el archivo con los casos de prueba encontramos tres funciones, una para cada prueba, que serán las ejecutadas en cada thread y una variable global tipo string. Los resultados de las pruebas se basarán en comprobar el estado actual de la string una vez terminado el thread. Para realizar un control más preciso y evitar lo mejor posible los problemas por prioridad de ejecución en el procesador, se utilizan varias llamadas a la función *sleep*().

6. Conclusiones

Después de realizar el trabajo sobre el proyecto se pueden analizar ciertos aspectos sobre el proceso y ver el cumplimiento de las expectativas iniciales.

Sobre la primera parte, el desarrollo de la metodología para MATE, es pertinente comentar que el resultado es satisfactorio y cumple los objetivos propuestos. Como se había planeado, la metodología es capaz de acoger un proyecto de la envergadura de MATE y permitir un desarrollo fluido manteniendo una serie de estándares que le den a la aplicación un aspecto de unidad y cohesión. Los resultados obtenidos son una serie de especificaciones y guías que respaldan todo el proceso de desarrollo. Además al haber realizado parte del trabajo sobre la aplicación junto a otras dos personas, ha sido posible ver los efectos de la metodología aplicada en la práctica y por lo tanto se ha podido corroborar su utilidad. En cuanto a los objetivos listados en el primer capítulo del documento, se puede observar que esta parte justifica el cumplimiento de los tres primeros.

El trabajo realizado sobre la aplicación, una vez asentada la metodología, ha sido la segunda parte del proyecto. Una vez aplicada la metodología a MATE se ha conseguido aportar el respaldo necesario para que pueda continuar evolucionando y el equipo de desarrollo pase a ser más flexible. No solo se ha conseguido aplicar ciertos estándares en cuanto al proceso de desarrollo, sino que también se dispone de una documentación más solida que facilitará el uso de MATE a nuevos usuarios. Finalmente las nuevas características añadidas a los módulos Common y DMLib hacen el sistema más fiable y menos propenso a errores desconocidos. Esto abarca el resto de objetivos propuestos y por lo tanto se puede considerar el resultado como satisfactorio.

6.1 El futuro de MATE

Aunque todos los objetivos establecidos se hayan cumplido, sigue faltando un paso siguiente en la evolución de MATE para considerarse un producto finalizado. Se han realizado pruebas unitarias sobre las clases de los módulos y la integridad de cada uno de ellos se puede considerar alta. Sin embargo, es necesario realizar el proceso de despliegue y observar el comportamiento de MATE frente a diferentes entornos y situaciones. Para ello será necesario realizar una serie de tests de integración y tests de despliegue, dónde se verá realmente el grado de funcionalidad que muestra la herramienta actualmente.

Además de establecer MATE como producto en cuanto a nivel de funcionalidad, también existen una serie de características que podrían aportar mucho a la herramienta de cara a ser accesible para un marco más amplio de usuarios:

- El sistema de configuración está preparado para leer la configuración desde diferentes entradas. Sin embargo, solamente está implementada la lectura de configuraciones a través de ficheros en un formato no estándar donde no se asegura ningún tipo de integridad. Como sugerencia en la evolución de la herramienta, sería posible trabajar en el sistema de configuración y implementar la lectura de diferentes formatos estándar como XML. De esta manera sería más fácil generar y modificar configuraciones para MATE y realizar comprobaciones sobre la configuración antes de cargarla directamente sobre el programa.

- Una de las ventajas de MATE es que no es necesario realizar ninguna operación sobre la aplicación antes de la ejecución. Aún así, cada vez que se ejecuta de nuevo la aplicación la sintonización de MATE es la misma. Por lo tanto una posible mejora en este proceso de sintonización es aportar un sistema de tratamiento de la información que permita disponer de una base de conocimiento asociada a cada aplicación particular. De esta manera se conservaría la ventaja de sintonizar en tiempo de ejecución pero sin el inconveniente de empezar de cero en cada ejecución.

- En el módulo DMLib encontramos una serie de instrucciones muy básicas para realizar el proceso de monitorización. Una posible mejora sobre esto en un futuro podría ser permitir la customización de este proceso para ser utilizado de manera más eficiente en cada aplicación particular.

6.2 Conclusiones personales

A nivel personal puedo decir que este proyecto ha aportado mucho y ha permitido que ponga en práctica varias de las disciplinas estudiadas en la carrera.

El hecho de haber formado parte de un equipo de desarrollo de varias personas me ha mostrado las dificultades que pueden surgir en la vida real y que no son posibles aprender de forma teórica. Tanto en aspectos de coordinación y trabajo en equipo como aspectos más técnicos a la hora de compartir información y trabajar en un mismo sistema que debe funcionar de manera unitaria.

El hecho de ser la primera experiencia en un proyecto de investigación también ha servido para ver la sinergia entre teoría y práctica, ya que aunque se trate de un proyecto nacido de investigación tendrá un futuro como producto software.

En cuanto a la parte técnica la segunda parte del proyecto me ha ayudado mucho a mejorar el dominio del lenguaje de programación C++ y ver la magnitud de un programa real en el que se ha trabajado durante muchas horas.

Bibliografía

Links

- The Message Passing Interface (MPI) standard - http://www.mcs.anl.gov/research/projects/mpi/
- Dyninst API - http://www.dyninst.org/
- C/C++ API - http://ta-lib.org/d_api/d_api.html
- Buildbot Documentation - http://buildbot.net/buildbot/docs/current/
- Doxygen Manual - http://www.stack.nl/~dimitri/doxygen/manual.html
- Library Programming HowTo - http://www.faqs.org/docs/Linux-HOWTO/Program-Library-HOWTO.html
- SQA definition - http://www.sqa.net/

Libros

- Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The complete reference", Vol. I, 2nd edition. Mit Pr,1996.
- Bjarne Stroustrup, "The C++ Programming Language", 3rd edition. Addison-Wesley, 1997.
- William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, Marc Snir, "MPI: The complete reference", Vol. II, 2nd edition. Mit Pr, 1998.

Documentos

- FN/Mats Henricson & Erik Nyquist, "Programming in C++, Rules and Recomendations", 1992.
- M. Xenos, "Usability Perspective in Software Quality" , 2001.
- Anna Morajko, "Dynamic Tuning of Parallel/Distributed Applications", UAB, 2003.
- Nick Jenkins, "A Software Testing Primer", 2008.
- Andrea Martínez, "Sintonización dinámica de aplicaciones MPI", UAB, 2010.

Índice de anexos

Anexo 1. Acta de seguimiento del proyecto número 1.

Anexo 2. Acta de seguimiento del proyecto número 2.

Anexo 3. Acta de seguimiento del proyecto número 3.

Anexo 4. Acta de seguimiento del proyecto número 4.

Anexo 5. Acta de seguimiento del proyecto número 5.

Anexo 6. Acta de seguimiento del proyecto número 6.

Anexo 7. Script instalación de Buildbot + dependencias.

Anexo 8. Código fuente de los casos de prueba unitarios realizados en Common.

Anexo 9. Documentación de módulos Common y DMLib generada con doxygen.

Anexo 10. Guía de estilo de codificación.

Anexo 11. Guía de instalación de Buildbot.