ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

TESIS DE GRADO

"DESARROLLO Y EVALUACIÓN DE UN MODELO DE PROCESOS PARA EL DESARROLLO DE SOFTWARE LIBRE BASADO EN EL MODELO BAZAR."

Previa a la Obtención del Título de Ingeniero en Computación
Especialización Sistemas Multimedia

PRESENTADA POR:
LUIS ANTONIO GALÁRRAGA DEL PRADO
ALEJANDRO MANUEL MORENO CÉLLERI

GUAYAQUIL - ECUADOR

2008

AGRADECIMIENTO

A nuestras familias

por siempre estar ahí cuando las

necesitábamos.

A nuestros profesores

por guiarnos cuando estuvimos perdidos.

A nuestros valiosos amigos.

A Dios.

DEDICATORIA

Con cariño,

a todas las personas

que de una u otra forma ayudaron

a culminar este trabajo.

TRIBUNAL DE GRADO

PRESIDENTE

Ing. Holger Cevallos Ulloa

DIRECTOR DE TESIS

Ing. Verónica Macías

MIEMBROS PRINCIPALES

Ing. Marisol Villacrés

Ing. Carmen Vaca

DECLARACIÓN EXPRESA

"La responsabilidad por los hechos, ideas y doctrinas expuestas en esta tesis, nos corresponden exclusivamente; y, el patrimonio intelectual de la misma, a la Escuela Superior Politécnica del Litoral"

(Reglamento de exámenes y títulos profesionales de la ESPOL)

Luis Antonio Galárraga del Prado

Alejandro Manuel Moreno Célleri

RESUMEN

En este documento se presentará la propuesta de un modelo de procesos para desarrollar software libre llamado MOCCA (Modelo Controlado para Código Abierto). La tesis está separada en 6 capítulos donde se tratarán diversos aspectos del modelo como su concepción, su desarrollo y los resultados obtenidos durante su aplicación.

En el Capítulo 1 se presentan los antecedentes necesarios para entender el entorno del software libre. Se identifica la situación actual del software libre a nivel mundial, su aceptación y los métodos usados en su desarrollo. Además se describen algunos estudios en torno al desarrollo de aplicaciones libres y se analiza la necesidad de un modelo de procesos estandarizado a la hora de desarrollar dicho tipo de aplicaciones.

En el Capítulo 2 se hace una introducción a conceptos básicos de la Ingeniería de Software necesarios para el entendimiento del modelo propuesto. En este capítulo se presentan los diferentes modelos existentes para el desarrollo de software, el Catedral y el Bazar, siendo el primero mayoritariamente usado en desarrollo de software privativo y el otro en software libre. También se explica que son las métricas y su importancia para la Ingeniería de Software. Por último, se discute el uso de métricas en el desarrollo de software libre.

En el Capítulo 3 se presenta el modelo de procesos propuesto, bautizado como MOCCA, Modelo Controlado para Código Abierto. Se describen los pasos que se deben seguir para poder emprender en el desarrollo de software libre, los cuales incluyen: definición de aspectos iniciales, análisis y diseño de la solución, implementación, proceso de estabilización, liberación del software y medición de la salud del proceso.

En el Capítulo 4 se describe la aplicación que desarrollamos usando el modelo propuesto. Se presentan los motivos por los cuales dicha aplicación sería útil, un análisis de cómo debería ser implementada, las herramientas que escogimos para desarrollarla y un diseño de la aplicación.

En el Capítulo 5 se hace un análisis del modelo propuesto tomando en cuenta la aplicación a desarrollar. Se toman decisiones en cuanto al ambiente de desarrollo necesario, las herramientas de soporte que se van a usar para el control del proceso y las métricas necesarias para la evaluación del modelo en este caso particular.

En el Capítulo 6 se analizan los datos obtenidos a lo largo del período de desarrollo. En este análisis se estudia la actividad que tuvo el proyecto a través de los diferentes canales de comunicación disponibles, posibles fallas en el proceso, entre otras cosas.

Para finalizar, se obtienen las conclusiones con respecto a la validez del modelo para el desarrollo de software libre. Se analizan sus fortalezas y debilidades, y se proponen mejoras.

ÍNDICE GENERAL

INDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

El desarrollo de software ha ido evolucionando a través del tiempo de diferentes maneras y no hay razón alguna para creer que vaya a dejar de evolucionar en el futuro. No son solamente las herramientas que se usan, sino los lenguajes de programación escogidos, los programas que se implementan, la forma de desarrollar el software, los que cambian a medida que el tiempo pasa.

En búsqueda de mejorar la eficiencia y aumentar la productividad, se ha recurrido a utilizar diferentes herramientas que agilitan el proceso de desarrollo. Sin embargo, el mayor cambio ha surgido a partir de replantear la manera en que el software es desarrollado, más que las herramientas con las que es creado. La Ingeniería de Software consiste en aplicar un enfoque sistemático y disciplinado al desarrollo de software donde usualmente se usan técnicas o procedimientos que llevan a un incremento en la confianza, productividad y fácil manutención del software.

El presente trabajo introduce la creación de un nuevo modelo de procesos específicamente orientado hacia el desarrollo de software libre. Se describen los problemas actuales con el desarrollo de software libre, las ventajas y desventajas de desarrollar bajo las características del estilo Bazar, las características del modelo nuevo MOCCA, los resultados obtenidos al desarrollar una aplicación colaborativa usando este modelo, y recomendaciones y posibles mejoras a este proceso.

CAPÍTULO I.

ANTECEDENTES Y JUSTIFICACIÓN

En este capítulo se presenta una introducción al software libre y a su entorno. No solo se habla del software libre en sí, sino también de su desarrollo y estudios realizados en torno al proceso de desarrollar aplicaciones FOSS.

Al hablar del entorno, se presentan ejemplos de proyectos FOSS relevantes alrededor del mundo, las ventajas que presentan a sus usuarios y motivos por los cuales decidieron que usar FOSS era la mejor opción.

Por último, se presenta uno de los principales problemas del desarrollo FOSS: la falta de un modelo estandarizado de procesos. Se discute por qué esto significa un problema grave para la comunidad de software libre y se propone una solución que lograría mitigar muchos problemas que se presentan a la hora de desarrollar FOSS.

Antecedentes

Se conoce como FOSS (Free Open Source Software) a cualquier programa o componente de software que ha sido publicado o distribuido bajo una licencia que brinda a los usuarios libertades como su uso para cualquier propósito, derecho a estudiarlo (lo cual implica tener acceso a su código fuente), derecho a modificarlo y derecho a redistribuirlo libremente con o sin modificaciones (1). Sin embargo, vale aclarar que el término hace referencia a dos corrientes ideológicas diferentes, pues quienes difunden el concepto de "Free Software" hacen énfasis en la libertad de los usuarios de computadoras como un derecho intrínseco, como una cuestión de valores; mientras que la corriente "Open Source" se enfoca en las ventajas prácticas que implica el desarrollo de software en comunidad con libre acceso al código fuente (2).

Algunos ejemplos de software creado por promotores de la corriente del Software Libre, son los proyectos GNU (3) (patrocinado por la Free Software Foundation) y Debian (4), desarrollados por comunidades de programadores voluntarios alrededor del mundo que creen y fomentan los preceptos de la libertad como un derecho de los usuarios de computadoras.

Por otra parte, la corriente “Open Source” es promovida principalmente por compañías que han obtenido excelentes resultados a través de su aplicación como modelo de negocio, donde la venta de licencias es reemplazada por la venta de servicios en torno al software, el mismo que a menudo se encuentra liberado bajo los términos de una licencia definida como libre. Compañías como Red Hat, Novel, MySQL, entre muchas otras obtienen ganancias brindando servicios de soporte y capacitación en relación a software libre.

El software libre en el mundo actual

Las tecnologías FOSS (Free Open Source Software) brindan a los países en vías de desarrollo una gran oportunidad para reducir la brecha digital. Sin embargo, cabe recalcar que no solo es útil para países en desarrollo sino también para países desarrollados. Algunos países, entre éstos Brasil, Francia, España y Alemania están aprovechando las ventajas del potencial del FOSS a través

de leyes que regulan el uso de este tipo de tecnologías en las instituciones públicas y en la educación. Destacable es el caso de la Junta de Extremadura en España, que ha desarrollado todo un sistema operativo libre, basado en GNU/Linux (5) para que sea usado en todas las instituciones públicas. España es considerada una potencia en el campo del software libre tanto en su uso como en su divulgación (6). Alemania, por otro parte, es el país con más uso de software libre a todo nivel (7). Francia (8) y Brasil (9) han empezado con programas de migración a tecnologías FOSS.

A causa del creciente interés, la investigación de diversos aspectos (técnicos, sociales y económicos) del FOSS se ha intensificado en los últimos años.

Como ejemplos claros de la efectividad de las tecnologías FOSS en la reducción de la brecha digital, podemos citar 3 casos específicos. En Filipinas, el gobierno decidió brindar ayuda a los colegios adquiriendo computadoras para mejorar el nivel de educación de éstas. Al hacer el análisis pertinente, tuvieron que escoger entre computadoras con el sistema operativo GNU/Linux o Microsoft Windows, el cual venía subsidiado a un costo de $20 por licencia e incluía capacitación sin costo adicional (10). Aún con este subsidio, GNU/Linux resultaba mucho más barato. Se compraron más computadoras de las que se tenía inicialmente pensado y actualmente están en planes de comprar más. Otro

ejemplo claro es el de Brasil, donde el gobierno, como parte de una campaña de alfabetización, está distribuyendo computadoras adquiridas a un costo de $100 equipadas con la distribución “Muriqui” de GNU/Linux, que fue especialmente diseñada para la educación de Brasil (11). Como dato curioso, la adquisición de las computadoras resultó más barata que la compra de libros por parte del gobierno. Por último, en Extremadura, España, una de las regiones más pobres del país, se decidió iniciar una campaña para brindar acceso al internet como un servicio público y estimular el desarrollo tecnológico de la región. Para este fin, debido al presupuesto con el que se contaba, se decidió usar FOSS para la elaboración de un sistema operativo, el cual se usa en los centros que proveen el servicio de internet. El resultado fue claro, se ahorraron cerca de 30 millones de euros gracias a la ejecución del proyecto (12).

El gobierno de Ecuador, siguiendo la tendencia mundial, ha establecido el uso de Software Libre como política de estado en las instituciones públicas vía decreto ejecutivo (13), con ello se espera que exista un considerable incremento en la investigación y desarrollo relacionados a este campo en nuestro país. Basado en dichas proyecciones, podemos anticipar que la definición de modelos y estrategias de desarrollo de FOSS son claves para la pronta y correcta adopción de estas tecnologías en una sociedad que no tiene ningún antecedente de aprovechamiento de las

mismas y más aún si quienes lo definen son estudiantes e investigadores del medio que conocen la realidad del país.

Situación actual del desarrollo de aplicaciones de Software Libre

A fin de presentar un análisis comprensible de la situación actual del desarrollo de software libre, vale la pena mencionar un par de aspectos históricos en relación a su origen.

La corriente del software libre data del año 1983 y su creador fue Richard Stallman, fundador de la FSF (Free Software Foundation) y del proyecto GNU (GNU's not Unix), el mismo que buscaba crear un reemplazo para el sistema operativo Unix, que brindara todas las libertades ya mencionadas a los usuarios de computadoras (14). Esto supuso también la redacción de una licencia que plasmara dichas libertades, la GPL (GNU Public License), actualmente en su tercera versión. Desde ese año hasta ahora, se han creado un sinnúmero de proyectos que de una u otra forma utilizan los conceptos definidos en la doctrina del software libre. Si bien la GPL no es la única licencia libre existente, es la más utilizada por los desarrolladores de FOSS en la actualidad.

Por otra parte, existen muchos sitios en internet que permiten a cualquier desarrollador publicar y administrar un proyecto de software libre aprovechando las ventajas del desarrollo colaborativo. Sitios como Savannah (15), SourceForge (16), BerliOS (17) o Tigris.org (18) brindan este tipo de servicios. Es importante recalcar que dichos sitios, están desarrollados a partir de proyectos de software libre que cualquier interesado puede descargar y utilizar para proveer el servicio en su comunidad. Proyectos de software libre de renombre y con grandes comunidades de usuarios y desarrolladores se han aprovechado de este tipo de herramientas para brindar a sus miembros de una infraestructura común de trabajo para cualquier extensión o proyecto derivado del software original. La siguiente tabla expone las estadísticas recabadas para los sitios de administración de proyectos más populares hoy en día a fin de ilustrar el impacto del desarrollo de software libre en los desarrolladores.

Sitio	Proyectos alojados	Usuarios registrados
SourceForge	180,405	1,878,499
BerliOS	5,413	38,619
Savannah	2,907	57,616

Tabla 1. Número de proyectos alojados y usuarios registrados en tres de los principales sitios de administración de proyectos FOSS (29 de junio del 2008) (15) (16) (17)

Otro hecho que prueba el gran impacto del software libre en el mundo actual, es su aceptación en la comunidad de usuarios. Muchos proyectos de software libre gozan de un excelente

posicionamiento, compitiendo sin problemas con sus análogos no libres.

Podemos citar el caso del servidor web Apache. Estadísticas para junio del 2008 estiman una cuota de mercado de 49.12% con más de 83 millones de sitios web usándolo, (19) siendo en la actualidad, el servidor web más difundido en la internet.

Es también destacable el caso del navegador libre Mozilla Firefox que para agosto del 2008 se ubica en segundo lugar con una cuota de mercado del 43.7%, versus la familia Internet Explorer que con todos sus integrantes lidera la lista con 50.6%. Sin embargo, esto fue muy diferente en sus inicios pues para enero del 2005, su cuota era del 16.6% versus Internet Explorer que llevaba gran ventaja con el 64.18% (20).

La evidencia expuesta hasta ahora confirma la influencia del software libre en el panorama tecnológico actual, sin embargo no se han expuesto las razones que motivan a los desarrolladores a trabajar en este tipo de proyectos. Dicha información es clave para entender cómo funcionan las comunidades de desarrolladores de software libre.

Existen diversas motivaciones al momento de emprender un proyecto de software libre pero la más común es simplemente la

existencia de una necesidad. Muchos proyectos nacieron a partir de un problema identificado por uno o varios desarrolladores, quienes lo resolvieron a través de un programa de computadora y en el proceso identificaron que muchas otras personas se beneficiarían con el progreso del mismo. En este caso, la existencia de esa necesidad o problema común es lo que motiva a los miembros de la comunidad a colaborar con el proyecto. El panorama explicado, puede estar sujeto a variantes. Es posible que los interesados en el progreso del proyecto no sean necesariamente desarrolladores independientes, sino empresas u organizaciones que incorporan desarrolladores asalariados a la comunidad. En otros escenarios, una compañía u organización (como una universidad o una institución pública) que se encontraba desarrollando un proyecto se da cuenta que puede reducir sus costos si logra encontrar otras compañías o grupos que se beneficien con el proyecto y por ende deseen colaborar con el mismo. A menudo, esas personas no se encuentran en la misma locación geográfica.

En este punto vale resaltar, que es posible que quienes formen parte de un proyecto de este tipo, no obtengan ningún beneficio económico directo del desarrollo. Si bien, este hecho puede ser un riesgo al momento de definir metas concretas, la evidencia sugiere que no es necesariamente un problema. Esto se debe a que desarrollar, no es la única forma de aportar a un proyecto

FOSS. La retroalimentación recibida en el transcurso de un proyecto proviene en su mayoría de los usuarios del software que reportan defectos. Nunca faltan quienes teniendo un conocimiento moderado de la estructura del software, pero no forman parte del grupo de desarrolladores del proyecto, aportan con parches de código o incluso documentación.

Como se puede apreciar, la forma en que se hace software libre sugiere fortalezas derivadas del desarrollo colaborativo con libre acceso al conocimiento, pero adolece de debilidades cuando tomamos en cuenta que quienes colaboran lo hacen de forma voluntaria (y consecuentemente no se tiene total control sobre ellos) y pueden estar ubicados en diferentes locaciones geográficas.

La siguiente sección describe brevemente el estado actual de la investigación en torno al software libre y al modelo bazar.

Estudios realizados en torno al desarrollo de aplicaciones de Software Libre

La mayor parte de los estudios realizados en torno al software libre se incluyen dentro de dos categorías bien diferenciadas: aquellos que miden el impacto del FOSS en un campo de aplicación en particular como la educación o un sector de la

industria y aquellos que estudian el desarrollo de un proyecto o grupo de proyectos a fin de encontrar aquellos aspectos destacables en su evolución y proceso de desarrollo. Lo que es común a ambas categorías, es su proveniencia, a menudo universidades e instituciones públicas que de una u otra manera se han beneficiado del FOSS. Es así, que para hacer una síntesis del estado de la investigación en este campo, es necesario categorizarla de acuerdo a lo expuesto.

La educación y la administración pública son tal vez los campos más influenciados por la corriente del software libre, siendo su impacto en el segundo, bastante reciente. Esto ha generado, la consiguiente publicación de diversos testimonios que revelan las estrategias aplicadas para su adopción y sus ventajas. Como ejemplo se puede citar la iniciativa de Ministerio de Ciencia y Tecnología de Venezuela (21) o la Unión Europea, a través de la Free Software Foundation Europe (22) . Dado que algunos países de la Unión Europea como Francia, Alemania y España (23, 24) han optado por el FOSS en la administración pública, esta región representa la fuente más rica de literatura en torno al impacto del FOSS en estos campos (25).

Sin embargo, la categoría en la que más estudios se han realizado, es la Ingeniería de Software Libre. El proceso de desarrollo FOSS contempla el libre acceso a la mayor parte de la información del proyecto, por lo que es relativamente sencillo para

los investigadores acceder a los datos. Estos estudios incluyen análisis de la evolución de propiedades intrínsecas del proyecto como tamaño, constitución y actividad (26). Proyectos como el kernel de Linux (27), GNOME (28), KDE, Debian, Red Hat (29), Apache (30), entre otros, han sido estudiados a profundidad, pues constan entre los proyectos FOSS más grandes y estables del panorama actual. El objetivo de estos trabajos es siempre el mismo: identificar aquellas condiciones y hechos que determinaron el éxito del proyecto. Otra línea de estudio, se ha enfocado en analizar las comunidades de FOSS desde un punto de vista social (30, 31, 32) con el objetivo de determinar quiénes son los desarrolladores de software libre y qué los motiva a participar en este tipo de proyectos. Otros estudios se han preocupado por facilitar el trabajo a otros estudiosos (33), a través de técnicas y opciones al momento de estudiar un proyecto FOSS.

Planteamiento del problema

Dentro de los proyectos de desarrollo de software libre, no existe actualmente un proceso de desarrollo totalmente definido que se pueda seguir cuando se desea emprender un proyecto de esta naturaleza. Se tiene una idea general de ciertos aspectos y características que debe tener el proceso, pero en realidad cada

proyecto utiliza un modelo de desarrollo propio, generalmente no estandarizado ni reglamentado.

Debido a la falta de un modelo estandarizado, el desarrollo de software usando características del modelo Bazar no es visto, por parte de la industria del software, como una alternativa seria o viable. Por otra parte, las compañías a menudo apuntan a definir su propio modelo de procesos más o menos basado en los modelos tradicionales estudiados por la Ingeniería de Software, los cuales tienen una estructura definida y pasos a seguir claros y concisos. Dichos modelos caen dentro de lo que Raymond (2001) (34) denominó modelos estilo Catedral. En general, los modelos de desarrollo convencionales son predecibles y permiten a los líderes de proyectos estimar tiempos y costos. Sin embargo, estos modelos de procesos estilo Catedral no son aplicables al software libre; y es por ello que se torna imperioso el diseño de procesos de desarrollo que, siendo aplicables al software libre, permitan a los líderes de estos proyectos estimar de forma efectiva los tiempos y costos de desarrollo de este tipo de aplicaciones.

Propuesta de la solución y justificación de la tesis

Definir un modelo de desarrollo estilo Bazar, el mismo que es frecuentemente utilizado en los proyectos de FOSS, teniendo en consideración las diferentes variantes que se pueden encontrar de

este modelo y tratando de adoptar las ideas más relevantes y actuales.

Además de definir el modelo, es necesaria su evaluación a fin de comprobar su validez como solución al problema. Para evaluarlo, es imperativo simular un ambiente FOSS en el cual se trabaje de manera colaborativa como comunidad usando canales de comunicación como correos electrónicos y mensajería instantánea teniendo en cuenta que la mayoría de proyectos FOSS son desarrollados por personas que usualmente no comparten la misma locación geográfica.

Mediante la definición de un modelo estandarizado que pueda ser adoptado por los desarrolladores de aplicaciones FOSS, la comunidad de software libre ganaría tanto en organización como en reputación. El no tener un modelo de procesos estandarizado es una razón por la cual muchas personas no se arriesgan a adentrarse en este tipo de desarrollo. Sin pasos a seguir, ni guías que indiquen los elementos necesarios para empezar, mantener y culminar un proyecto FOSS, muchas personas optan por dejar a un lado su iniciativa de desarrollar usando software libre. Además de esto, las grandes empresas al ver que existe un modelo estandarizado que es controlado, pero a su vez no compromete la libertad inherente al estilo Bazar, se verán atraídas hacia este tipo de desarrollo.

CAPÍTULO II.

MARCO TEÓRICO

En este capítulo se presenta la teoría necesaria para poder entender la terminología usada a lo largo del documento. Además de la terminología, también se explican otros elementos importantes pertenecientes a la ingeniería de software.

Conceptos básicos de modelos de desarrollo de software

Los modelos de desarrollo de software fueron creados gracias al estudio que se lleva a cabo en la ingeniería de software. Por ende, lo primero que se necesita saber es qué es la ingeniería de software. Ésta se define como “una disciplina que comprende todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema, hasta el mantenimiento

de éste después de que se utiliza” (35). Es importante notar que no solo se refiere a los aspectos técnicos del desarrollo, sino que incluye actividades ajenas al mismo y que están más ligadas a la administración de un proyecto de software e inclusive actividades de soporte. En general, se podría decir que su objetivo es que las personas adopten un enfoque más sistemático y organizado en su trabajo para lograr crear software de calidad.

Una de las formas que la Ingeniería de Software plantea para lograr este enfoque sistematizado, son los procesos de desarrollo de software, que no son más que el conjunto de actividades asociadas para producir software. Dentro de todos los procesos de desarrollo, existen 4 actividades esenciales que se pueden encontrar en todos ellos (36):

- Especificación

  Funcionalidad y restricciones de operación del software.
- Desarrollo

  La producción del software de acuerdo a la especificación.
- Validación

  Asegurarse que se está desarrollando lo que el cliente requiere.
- Evolución

  El software debe evolucionar para cumplir con los requerimientos cambiantes del cliente.

Distintos procesos de desarrollo de software organizan estas actividades de diferentes formas, variando su duración, su nivel de detalle, e inclusive sus resultados.

Aunque los procesos ayudan mucho a la sistematización del desarrollo, era necesario algo más. En diferentes situaciones, se empezó a notar que ciertos enfoques eran más beneficiosos que otros, y que ciertos procesos servían de mejor manera. En base a esto, se crearon los modelos de procesos de desarrollo. Un modelo de procesos es una representación abstracta de un proceso de desarrollo de software. Cada modelo representa un proceso de desarrollo desde una perspectiva particular por lo que provee información parcial acerca de ese proceso. Estos modelos generales no son descripciones definitivas de los procesos de desarrollo, mas bien, son abstracciones útiles que se pueden utilizar para explicar diferentes enfoques para desarrollar software. Para sistemas muy grandes, no se utiliza solamente un modelo, se utilizan varios, donde diferentes componentes del sistema pueden ser desarrollados bajo un modelo de desarrollo diferente.

Modelos existentes

Actualmente, en la Ingeniería de Software existen muchos modelos de donde escoger para el desarrollo de un producto

cualquiera. Cada uno tiene sus características, ventajas y desventajas que lo diferencian de los otros. A continuación los más conocidos (37):

- Cascada

  La característica principal de este modelo es que es secuencial. Para avanzar a una etapa cualquiera, se tiene primero que completar la etapa anterior, simulando el flujo de una cascada. Las ventajas de este modelo es que es muy organizado y genera mucha documentación por cada etapa. Teniendo en cuenta esto, si los requerimientos iniciales son obtenidos de manera correcta, es casi seguro que el producto final sea del agrado de los clientes.

  La desventaja es que debido a que es secuencial, no es fácil retroceder, y si se encuentran errores en alguna etapa posterior, se hace muy complicado resolverlo. Además de eso, debido a que el producto se implementa en las fases finales, puede haber pasado tanto tiempo que los requerimientos iniciales pueden haber cambiado. Por ello se recomienda utilizarlo cuando exista certeza de que los requerimientos permanecerán más o menos constantes y los ejecutores del proyecto tengan bastante conocimiento acerca del dominio del problema.

- Espiral

La característica principal de este modelo es que funciona de manera iterativa. Es decir, funciona de tal manera que cuando termina la última etapa empieza otra vez la primera. En cada ciclo de la espiral, se toman decisiones acerca de que es necesario empezar en esa espiral en particular.

La ventaja de este modelo es el análisis de riesgos que se hace al inicio de cada ciclo. Gracias a este análisis, se puede decidir con mucha facilidad el paso a seguir en cada ciclo y además se puede integrar nuevos requerimientos o cambiar requerimientos previamente aceptados con mucha facilidad.

El problema que tiene este modelo es que el desarrollo del sistema toma mucho tiempo en comparación a otros modelos. Aparte de esto, el modelo es costoso debido a que el tiempo de desarrollo tiende a ser extenso.

- Prototipado (Prototipos Desechables)

  Este modelo es un poco diferente a los modelos anteriores. La idea principal es el de desarrollar un producto mediante la elaboración de prototipos del producto. También es un modelo iterativo, en el sentido que se desarrollan un sin número de prototipos uno después del otro. Se empieza a

trabajar por los requerimientos que son más propensos a errores y los más importantes, para darles más tiempo de prueba en los prototipos.

La ventaja es que se tiene una continua retroalimentación de parte de los usuarios finales, ya que evalúan el producto en base a los prototipos. En este proceso de retroalimentación se generarán sugerencias, aclaraciones, correcciones, etc. Gracias a esto, es muy probable que el producto final sea lo que el usuario necesita.

Una desventaja que tiene es que por desarrollar los prototipos lo más rápido posible, se pueden llegar a tomar malas decisiones en los aspectos técnicos como lenguaje de programación, diseño arquitectónico, entre otros.

- Incremental (Desarrollo Exploratorio)

  La idea principal es que el producto vaya evolucionando a medida que se lo va implementando. Se empiezan por los requerimientos más sencillos, y poco a poco se van agregando más y más funcionalidades a petición del cliente.

  La ventaja es que el producto es desarrollado con la retroalimentación provista por el usuario final, y por ende es

muy probable que el producto cumpla con sus expectativas.

La desventaja es que debido a que se necesita al usuario final para las pruebas, puede llegar a ser un modelo costoso y extenso. (38)

- Desarrollo Rápido de Aplicaciones (RAD)

  Este modelo tiene como principal característica el uso de aplicaciones ya existentes y su integración al producto en desarrollo. Los productos disponibles comercialmente (COTS, Comercial off the Shelves) son esenciales en el avance de este modelo. Una vez que se tienen los requerimientos iniciales, los programadores se encargan de buscar aplicaciones que cumplan esos requerimientos y que se puedan comprar. Luego integran dichos componentes para construir el sistema final.

  La ventaja es que el desarrollo se vuelve muy rápido en comparación a otros modelos. Esto se debe a la gran reutilización de código al usar COTS.

  Sin embargo, para lograr esta gran velocidad de desarrollo, se tienen que sacrificar algunos requerimientos, ya que encontrar una aplicación que tenga exactamente las

mismas funcionalidades que se requieren, es una tarea casi imposible.

Modelo Catedral

"Yo pensaba que el software de mayor envergadura (sistemas operativos y herramientas realmente grandes, tales como Emacs) requería construirse como las catedrales, es decir, que debía ser cuidadosamente elaborado por genios o pequeñas bandas de magos trabajando encerrados a piedra y lodo, sin liberar versiones beta antes de tiempo." (34)

La cita anterior, define claramente la principal característica del modelo de desarrollo de software estilo Catedral. Sin embargo, vale aclarar que cuando se hace referencia a este concepto, no se está hablando de un modelo concreto, sino de un estilo de desarrollo que engloba en sí a un grupo de modelos de procesos de la ingeniería de software.

El nombre Catedral fue adoptado debido a las características del desarrollo de software propietario, el cual usualmente toma un enfoque mucho más planeado y centralizado. La persona o personas encargadas de la administración del proyecto toman decisiones en torno a la dirección y ejecución del mismo. Usualmente sus responsabilidades incluyen:

- Definir objetivos
- Monitorear detalles
- Motivar a las personas
- Organizar
- Asignar recursos

Como se mencionó anteriormente, este estilo agrupa diferentes modelos y tiene etapas que se pueden o no encontrar en la gran mayoría de estos:

- Análisis de Requisitos:

  En esta etapa se extraen y analizan los requisitos que un producto de software debe cumplir. Si hay clientes, es necesario establecer una reunión con ellos para poder definir los requisitos. Aunque parezca una tarea trivial, es en realidad muy importante. Requiere pericia para poder reconocer requisitos incompletos, ambiguos o contradictorios. Si esta etapa no se lleva a cabo debidamente, existe una alta posibilidad que el producto final no cumpla las expectativas del cliente.

- Diseño

  En la etapa de diseño se debe determinar cómo debe funcionar el software sin la necesidad de entrar en detalles de implementación. Esto significa que se debe establecer qué hace el software, mas no el cómo.

- Implementación

  En esta etapa se traduce el diseño definido con anterioridad a código. Podría pensarse que esta etapa es la más larga de todo el modelo, sin embargo esto depende mucho de los requisitos que se hayan establecido y el lenguaje de programación escogido.

- Pruebas

  En esta etapa se verifica que el software haga lo que tiene que hacer, y que lo haga de manera correcta. El objetivo de esta etapa es librar al producto de errores. Estas pruebas deben ser llevadas a cabo por personas ajenas al desarrollo del producto si es posible.

- Mantenimiento

  Esta etapa implica entre otras cosas, enmendar errores que se descubran una vez que el producto se encuentre en producción o mejorarlo para dar soporte a más funcionalidades si el cliente lo requiere.

Modelo Bazar

“El estilo de desarrollo de Linus Torvalds ("libere rápido y a menudo, delegue todo lo que pueda, sea abierto hasta el punto de

la promiscuidad") me cayó de sorpresa. No se trataba de ninguna forma reverente de construir la catedral. Al contrario, la comunidad Linux se asemejaba más a un bullicioso bazar de Babel, colmado de individuos con propósitos y enfoques dispares (fielmente representados por los repositorios de archivos de Linux, que pueden aceptar aportaciones de quien sea), de donde surgiría un sistema estable y coherente únicamente a partir de una serie de artilugios." (34)

En contraposición a lo expuesto en la sección previa, la cita anterior resalta las principales características del modelo de desarrollo estilo Bazar, el mismo que no es un modelo en todo el sentido de la palabra. Sin embargo, mientras el concepto de Catedral engloba las características de varios modelos diferentes usados en el desarrollo de software propietario, el modelo Bazar es en sí una serie de lineamientos y características de los proyectos FOSS. Esto se debe a que en el desarrollo de software libre, no existen modelos a seguir mientras se desarrolla.

El modelo Bazar tiene una premisa básica sobre la cual todo el modelo gira.

"Libera a menudo, libera rápido"

Esta es la principal característica del desarrollo de software libre. Básicamente significa que el desarrollador o grupo de desarrollo debe liberar versiones de su aplicación lo más pronto posible. En el desarrollo propietario, no se libera una versión del software hasta que se tenga una aplicación que funcione de manera adecuada. En el modelo Bazar, no es necesario que la aplicación funcione de manera adecuada para ser liberada, es más, puede estar llena de defectos y apenas funcionar, pero lo importante es que pueda ser ejecutada. Si la aplicación puede ser ejecutada, puede ser probada, lo cual nos lleva a otro de los principios básicos del desarrollo Bazar

"Dados suficientes ojos, todo error es superficial"

Este principio, también llamado el Efecto Delphi, es uno de los pilares detrás de la alta estabilidad y calidad de los productos de software libre. Además, es solo aplicable en las comunidades de software libre debido a la gran cantidad de personas que las componen. En el modelo Catedral, usualmente se tiene un grupo reducido de gente dedicada a probar y encontrar defectos en las aplicaciones. Esto se debe a que no se pueden designar tantos recursos (especialmente económicos) a esta etapa del modelo. En cambio, en las comunidades de software libre, se tienen miles de personas dispuestas a probar una aplicación sin costo alguno. Aunque el hecho de no incurrir en gastos de pruebas es una

ventaja significativa, más importante es el hecho de tener miles de personas probando la aplicación, cada uno siguiendo un hilo de ejecución diferente, cada uno con un sistema diferente, diferentes configuraciones, etc. Esto garantiza con una muy alta seguridad que la aplicación va a ser probada bajo todos los escenarios que se podrían presentar en la vida real una vez que haya sido liberada.

Métricas

Hasta ahora, se han expuesto una serie de metodologías para desarrollar software, asumiendo que en el proceso todo marcha sin problemas. Lamentablemente, en un ambiente real de desarrollo, eso no ocurre. Debido a eso, la Ingeniería de Software define una serie de actividades paralelas al desarrollo del software que permiten llevar un control del proceso. Dichas actividades están encaminadas a recabar información que permita a los administradores del proyecto conocer el estado del proyecto en cualquier instante de tiempo, a fin de que puedan tomar las acciones necesarias para conducir exitosamente el proyecto.

Conceptos básicos

A fin de emprender un proceso paralelo de control, es necesario realizar mediciones. La medición del software se refiere a derivar

un valor numérico para algún atributo de un producto o proceso de software. Comparar estos valores con los de un proceso ideal, permite a los líderes o administradores de un proyecto sacar conclusiones sobre la salud del objeto medido (37). El mayor reto al momento de definir dichos indicadores o métricas, es encontrar aquellas que verdaderamente reflejen el estado del proyecto o producto.

Clasificación de las métricas

Al hablar de software, existen dos tipos principales de métricas o indicadores, dependiendo de la entidad cuyos atributos se desea cuantificar. Éstas son:

- Métricas del proceso
- Métricas del producto

Métricas del proceso

Las métricas del proceso permiten cuantificar atributos del proceso de desarrollo del software. Su objetivo es identificar condiciones problemáticas en la metodología de desarrollo y propender a una continua mejora de la misma. Un ejemplo de una métrica del proceso es la densidad de defectos durante las pruebas del sistema, que mide la cantidad de errores detectados en el software cuando éste se encuentra totalmente integrado. La

experiencia sugiere que esta métrica tiene una correlación alta con la densidad de defectos del producto, cuando éste ya se encuentra en el mercado (39).

Métricas del producto

Miden características intrínsecas del software. Son utilizadas para definir la calidad del software. En este punto vale la pena hacer dos aclaraciones. La primera tiene que ver con la relación entre las métricas del producto y las del proceso. El objetivo final del proyecto siempre es producir un software de calidad, por ello el proceso de toma de métricas en su totalidad está orientado a ese fin. Sin embargo, la experiencia sugiere que un proceso de calidad no necesariamente conduce a un software de calidad. La segunda aclaración es que las métricas del producto no necesariamente evalúan al software cuando éste se encuentra corriendo. Las métricas del producto también incluyen aquellas mediciones hechas en las representaciones del sistema como el diseño o la documentación.

Software Libre y métricas

Una de las principales razones por las cuales el desarrollo al estilo bazar no es considerado como una metodología viable al momento de emprender un proyecto de software, es la ausencia

de mecanismos de control de calidad. La razón es inherente a la naturaleza del desarrollo colaborativo y las motivaciones de quienes a menudo integran el equipo de desarrollo.

Sin embargo, el desarrollo bazar cuenta con una ventaja. Debido a que la información del proyecto es de acceso público, la adopción de hábitos de control del proceso pueden ser viables, si se cuenta con una comunidad lo suficientemente grande y comprometida. Además de ello, mucho del conocimiento que se posee sobre los proyectos de software libre más conocidos, provienen de estudios realizados por personas ajenas al proyecto.

Concluimos entonces que si bien las prácticas de control de calidad a través de métricas no son un hábito común en los proyectos FOSS, su adopción puede ser viable en muchos casos.

CAPÍTULO III.

MODELO DE DESARROLLO

En este capítulo se presenta el modelo propuesto - MOCCA - para el desarrollo de FOSS. Dentro de este capítulo se encontrarán todos los pasos necesarios para iniciar un proyecto de software libre, ya sea usando una aplicación existente como aplicación base, o iniciando una aplicación desde cero.

El modelo fue diseñado usando como base el estilo de desarrollo Bazar, el cual es el modelo usado en el desarrollo de FOSS. Sin embargo, el modelo Bazar no es un modelo per se; se lo podría considerar más que un modelo una serie de sugerencias o características que se siguen cuando se desarrolla de manera colaborativa con una comunidad de software libre.

Analizando la información recabada en la literatura relacionada con el desarrollo de FOSS, hemos identificado que todos los procesos de producción de este tipo de software constan de las siguientes etapas:

- Definición de aspectos iniciales
- Análisis y diseño de la solución
- Implementación
- Proceso de estabilización
- Liberación del producto

Sin embargo, es importante recalcar que las etapas mencionadas no se presentan necesariamente de manera secuencial. El siguiente diagrama muestra la secuencia de ejecución del modelo MOCCA en un ambiente típico de desarrollo FOSS.

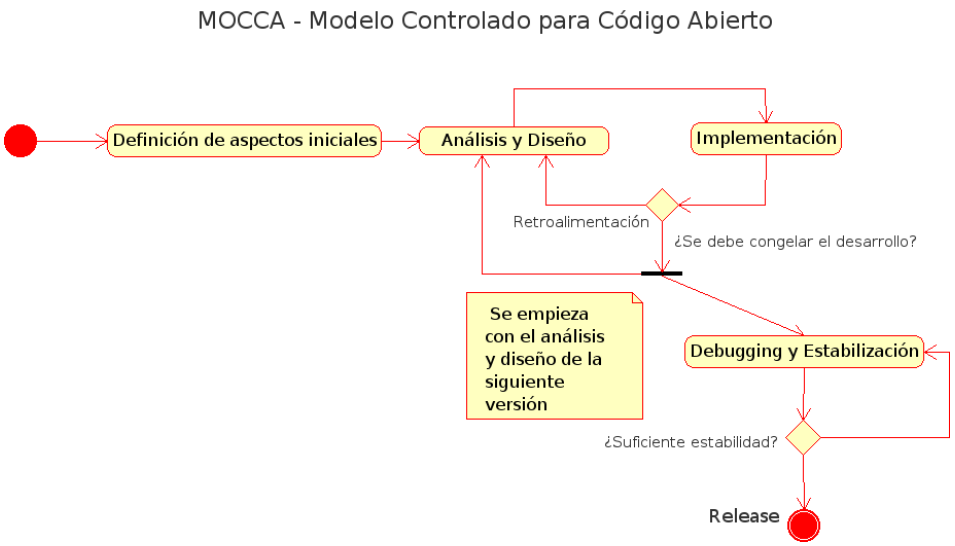

Gráfico 3.1. Modelo Controlado para Código Abierto (MOCCA)

Definición de aspectos iniciales

Define todas las actividades que se deben ejecutar para proveer a los desarrolladores del entorno apropiado para empezar. No debe entenderse como entorno a un espacio físico, sino todas las consideraciones técnicas y legales para poder emprender y trabajar en un proyecto de esta naturaleza. Algunas de ellas son:

- Definición de estructura y grupo de trabajo
- Definición de reglamentos y políticas
- Toma de requerimientos
- Definición del nombre y la licencia del producto
- Implantación de los aspectos técnicos.

Definición de estructura y grupo de trabajo

La definición de la estructura y grupo de trabajo es análoga a la creación de un organigrama dentro de una compañía. En ella se determina las responsabilidades y roles de cada uno de los miembros del grupo inicial de trabajo. La estructura particular de dicho organigrama depende exclusivamente de lo que decidan los miembros y no existen reglas generales para definirla. No obstante, un vistazo a diversos proyectos de software libre nos ha permitido identificar algunos roles comunes en estas organizaciones (40):

- Administradores de alto nivel

  Se encargan de la dirección general del proyecto.

- Administradores de releases

  Encargados del proceso de la liberación de una versión del software.

- Dueños de módulos

  Encargados de un módulo del proyecto, no necesariamente son quienes lo implementaron.

- Revisores

  Encargados de revisar y validar código.

- Committers

  Personas encargadas de subir los cambios al repositorio del proyecto.

- Contribuyentes

  Hacen aportaciones diversas al proyecto como documentación, reportes de bugs, entre otros.

Definición de reglamentos y políticas

Consiste en establecer las reglas a las que estarán sujetos todos los miembros de la comunidad. Al igual que en la definición de la estructura del grupo de trabajo, éstas dependen de lo decidido por quienes arrancan el proyecto. No obstante, existen algunos puntos que son comunes a todos los proyectos de este tipo. Algunos de ellos son (41):

- Privilegios de acceso al repositorio general
- Condiciones necesarias para formar parte de la comunidad

- Quiénes toman las decisiones y cómo lo hacen
- Manejo de situaciones especiales

Cabe recalcar, que dependiendo de los reglamentos que se formulen en esta etapa, la comunidad puede inclinarse por una tendencia informal u otra mucho más estricta.

Toma de requerimientos

Usualmente los proyectos de software libre no llevan a cabo una toma de requerimientos de manera explícita, sino que el proyecto nace a partir de la necesidad de uno o más programadores de solucionar algún problema encontrado, o alguna funcionalidad requerida que algún software existente no posea en ese momento. Una vez que el problema es identificado, es común que se busquen proyectos de software libre ya existentes que posean funcionalidad parecida a lo requerido, y que dentro de todos los proyectos encontrados, se escoja al que más se asemeje al que queremos implementar y usarlo como base, es decir, es muy raro que un proyecto empiece desde cero.

Definición del nombre y licencia del producto

Aunque parezca trivial, es muy importante. Con relación al nombre, vale la pena acotar que debe ser consistente con el

objetivo del producto y debe estar libre de posibles problemas legales y sociales como nombres de marcas registradas, o términos ofensivos. En cuanto a la licencia, es vital dado que determina, no solo el uso que le puedan dar al software sino además restricciones en cuanto a las herramientas o recursos que se podrán emplear en el transcurso del proyecto. Se recomienda, no redactar la licencia desde cero, sino tomar una ya existente, puesto que éstas han sido redactadas por personas que saben del tema y han sido aplicadas en múltiples escenarios. Otros aspectos que se deben considerar con relación a la licencia, es su compatibilidad con otras licencias como la GPL, la licencia libre más popular. En todo caso, la selección de la licencia depende ciento por ciento de los objetivos establecidos al inicio y no debe ser tomada a la ligera (41).

En relación a la definición de la licencia, debemos recalcar que su definición no necesariamente se realiza siempre al inicio del proyecto, pues ésta depende de sus objetivos, los mismos que pueden estar sujetos a cambios. Un claro ejemplo de esta situación es el proyecto Eclipse, el cual nació como una iniciativa no libre de IBM que se convirtió luego en un proyecto open source respaldado por un consorcio compuesto por múltiples compañías. Esta decisión surgió como consecuencia de un análisis costo-beneficio que reveló que sería mucho más económico liberar el

código del proyecto bajo los términos de una licencia open source (42).

Implantación de los aspectos técnicos

Consiste en implantar el entorno tecnológico en el que trabajarán los miembros del proyecto. Dada la naturaleza de este tipo de proyectos, se debe tener especial consideración en los medios de comunicación. Entre los puntos que hay que tomar en cuenta constan (41):

- Servidores donde se alojará el proyecto
- Sitio web
- Foros de la comunidad
- Listas de correo
- Sistema de control de versiones.
- Bug Tracker
- Chat (no imprescindible)
- Wiki (no imprescindible)
- Mecanismo de respaldo de información

Análisis y diseño de la solución

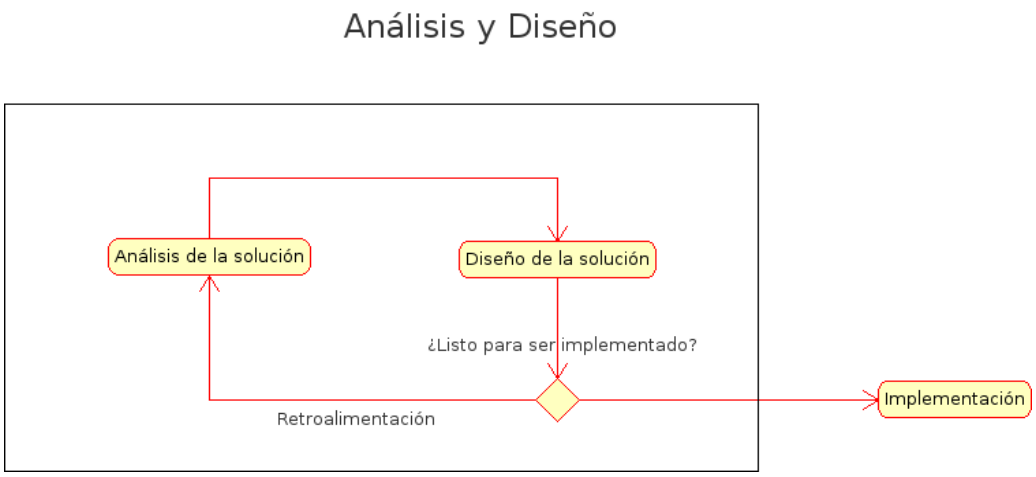

Gráfico 3.2. Etapa de análisis y diseño de MOCCA

No muy diferente al concepto existente en el desarrollo de software comercial o privativo. Sin embargo, en este tipo de proyectos, la primera vez que se pasa por esta etapa, ésta podría no estar sujeta a la naturaleza del desarrollo colaborativo (si por ejemplo quienes arrancan el proyecto deciden por su cuenta definir los requerimientos y el diseño inicial), lo cual dependerá de los reglamentos y políticas definidas en la etapa anterior. De haber decidido efectuar el diseño inicial de la aplicación de forma colaborativa, esta decisión originará un ciclo similar al de la fase de implementación, esto es: se propone una solución, se recibe retroalimentación, se analizan estas aportaciones y se genera una nueva solución tomando en cuenta los argumentos expuestos por los miembros encargados del diseño (34).

Cabe recalcar que éste modelo, es válido tanto para el diseño arquitectónico como para el diseño detallado de cualquier componente del sistema.

Análisis de la Solución

Durante las siguientes iteraciones del ciclo entre esta etapa y la implementación, existirá de forma implícita un continuo análisis, recepción de sugerencias, reporte de defectos y aportaciones por parte de los miembros de la comunidad. En general, el análisis referido aquí, incluye de forma general las siguientes actividades:

- Recepción de nuevos requerimientos
- Recepción de retroalimentación
- Análisis de los aspectos preseleccionados.

Recepción de nuevos requerimientos

En esta etapa se definen nuevas funcionalidades que podrán ser agregadas al proyecto estableciendo cuáles son más importantes. Estos nuevos requerimientos pueden provenir de diversas fuentes como: usuarios, stakeholders o miembros del equipo de trabajo, a través de los medios de comunicación del proyecto (listas de correo, foros, etc). En el caso de que los requerimientos sean formulados por usuarios del software, puede ocurrir que un requerimiento no se considere apropiado o viable, en cuyo caso

es descartado. Dependiendo de la estructura organizacional del equipo, estas decisiones pueden ser tomadas unilateralmente por el o los administradores del proyecto o pueden ser sometidas a consenso.

Recepción de retroalimentación

En esta etapa, se recolecta toda la retroalimentación generada por la comunidad, sea ésta, defectos encontrados en el código, documentación o diseño, correcciones a defectos, parches, sugerencias a soluciones. Esta actividad, se realiza paralelamente a la recepción de requerimientos. La aceptación o rechazo de la retroalimentación recibida es tarea de los encargados del área pertinente a dicha aportación.

Análisis de los aspectos preseleccionados

En esta etapa se analizan todas las aportaciones recibidas por parte de la comunidad aprobadas previamente y se las prioriza. Dentro del análisis que se lleva a cabo se debe tomar en cuenta algunas consideraciones.

Para requerimientos:

- Tiempo y costo estimado de implementación.

- Cuan necesario es. Esto se puede medir en base a cuántos usuarios han manifestado su deseo de que se añada.
- Valor que añade al proyecto.

Para correcciones de defectos:

- Severidad del defecto
- Validez de la solución
- Eficiencia de la implementación

Una vez tomados en cuenta estos criterios, se debe decidir en qué punto en el transcurso del proyecto se van a implantar los cambios, ya sean estos correcciones o inclusión de nuevos requerimientos.

Diseño de la solución

Una vez que se han definido los cambios a realizar, se procede a diseñar cómo se los va a implementar. Las actividades de diseño son muy dependientes de la aplicación a desarrollar y de las normas y estándares de diseño y documentación acordados cuando se arrancó el proyecto.

Implementación

Toda la información aportada por los miembros de la comunidad es integrada al sistema, ya sea en forma de nuevas líneas de código, documentación, corrección de código existente, etc. Cabe recalcar que la recepción de sugerencias sigue siendo parte del quehacer de quienes integran el proyecto. El fin de los ciclos de análisis, diseño e implementación, está determinado por sí el producto posee un nivel de estabilidad apropiado, definición que depende exclusivamente de lo que se haya acordado en la primera etapa del proyecto. En muchos casos, ese nivel se alcanza cuando ha transcurrido un tiempo determinado establecido por los miembros del proyecto, sin embargo puede depender también de las reacciones de los usuarios o de otros factores. Una vez que se ha decidido terminar definitivamente con el ciclo ya mencionado, se procede a realizar el congelamiento del desarrollo. Congelar el desarrollo significa que no se aceptarán nuevos requerimientos para la versión en curso. Esto supone que las actividades que se lleven a cabo más adelante se enfocarán exclusivamente en la estabilización del software.

Debemos acotar, que el final de esta etapa, debe producir un código que sea capaz de ser ejecutado y por lo tanto validado por los miembros de la comunidad. Es importante resaltar además, que el resultado producido, puede contener defectos, salvo

aquellos que impidan su ejecución. Esto implica que el software debe funcionar, aunque no lo haga del todo bien.

Por la naturaleza del modelo de desarrollo bazar, la etapa de implementación debe ser lo más breve posible a fin de acelerar la validación y descubrimiento de defectos. El final de un ciclo de implementación, está definido por la actualización del contenido del repositorio del proyecto y de no ser el momento apropiado para congelar el desarrollo, genera un nuevo ciclo que parte nuevamente del análisis y diseño.

Proceso de estabilización

En esta etapa, el proceso de retroalimentación expuesto en las dos etapas anteriores, se enfoca exclusivamente en las funcionalidades que esta liberación tendrá. La comunidad se prepara para minimizar la cantidad de defectos que tendrá el producto que se libere (40). La forma más común de implementar el proceso de estabilización es a través de la liberación de versiones temporales del software, las mismas que estarán sujetas a un período intensivo de pruebas por parte de la comunidad. Esta etapa tiene una estructura cíclica en la que se realizan las pruebas, se reportan los defectos encontrados a través de los canales de comunicación, se corrigen dichos defectos y se vuelve a probar generando varios ciclos prueba-

corrección. Esta característica es considerada fundamental en el modelo de desarrollo bazar, pues, de contar con una comunidad lo suficiente grande, se puede garantizar que el software será probado en un sinnúmero de condiciones diferentes, lo cual garantiza la detección de defectos. Como esto implica la liberación de versiones, el esquema de versionamiento empleado es decisión de la comunidad. De igual forma, la duración de este período (finalización del ciclo prueba-corrección) depende de la definición de estabilidad que se haya acordado. Algunos criterios para decidir cuándo terminar con el período de estabilización son:

- **Tiempo**

  Se define un período de tiempo, al cabo del cual se termina con la etapa de estabilización. Dicho período de tiempo se puede definir de dos maneras:

  - Períodos de tiempo iguales para todos los procesos de estabilización
  - Períodos determinados previamente, en base a un análisis para estimar cuanto tomará estabilizar el producto.

  Una vez transcurrido este período de tiempo se procede con la liberación de la versión definitiva.

- **En base a métricas**

El proceso se detiene cuando se ha alcanzado un nivel de estabilidad previamente definido en base a un conjunto de métricas. Nuevamente, la definición de dicho nivel de estabilidad depende de la comunidad, sin embargo consideramos que se deben tomar en cuenta algunos aspectos al momento de seleccionar las métricas a utilizar y los valores aceptables (para esas métricas) que detendrán el proceso. Estos aspectos son generalmente requisitos no funcionales del producto como:

- Robustez
- Estabilidad
- Desempeño
- Futuros usuarios (lo cual tiene que ver con niveles de usabilidad)

Cabe recalcar que durante el período de estabilización, otra parte del equipo puede trabajar paralelamente en la que sería la siguiente versión del software.

Liberación del producto

Define las actividades que se deben ejecutar para liberar una versión del software. Se podría definir como el objetivo a corto plazo de cada ciclo. Algunas de las actividades más comunes en esta etapa son:

- Informar a los miembros de la comunidad acerca de la nueva liberación.
- Actualizar los canales de comunicación del proyecto como sitio web, wikis y cualquier fuente controlada por la comunidad que haga referencia al proyecto.

Medición de la salud del proceso

Paralelamente a la ejecución de las tareas que demanda el proyecto como tal, al igual que cualquier proceso de software, es necesario que éste sea medido a fin de identificar problemas que puedan afectar la ejecución del proyecto. En el caso de proyectos de FOSS, lo que se trata de evitar es que el proyecto termine, lo cual ocurre cuando todos los desarrolladores pierden el interés en seguir aportando con su tiempo y esfuerzo al mismo.

La toma de métricas es una responsabilidad de todo el equipo de desarrollo. Sin embargo, de acuerdo a la naturaleza de la métrica, pueden haber algunas que sean responsabilidad única de ciertos miembros del equipo. Estos tipos de roles, deben ser definidos al momento de establecer las normas y políticas del proyecto y el rol de un miembro puede variar con el tiempo.

Hemos planteado el siguiente conjunto de métricas para ayudar a identificar tendencias anómalas en el curso del proyecto. Las hemos dividido en métricas del proceso y métricas del producto.

Métricas del proceso

Permiten medir características del proceso de desarrollo.

- *Momentum* o frecuencia de liberaciones

  Su obtención debe estar a cargo de un miembro activo de la comunidad a partir de la primera liberación. Este valor permite establecer el nivel de actividad del proyecto y advertir sobre una posible desmotivación por parte de los integrantes.

- Número de correos enviados a la lista de correo por unidad de tiempo.

  Al igual que la métrica anterior, permite medir la actividad del proyecto de una forma más completa y sugerir desinterés de parte de sus integrantes. Este valor puede ser un buen complemento para la frecuencia de liberaciones pues, al existir diversas listas de correo, nos permite analizar más a detalle y por separado, la actividad de cada uno de los grupos que integran la comunidad, entendiéndose por grupos, a los desarrolladores, usuarios y colaboradores.

- Frecuencia de conflictos en el servidor de control de versiones

  Permite medir la eficiencia del modelo de desarrollo. Un valor elevado para esta métrica, puede sugerir una mala planificación de las responsabilidades definidas o debilidades en el diseño de la aplicación lo cual implica desperdicio de tiempo y esfuerzo. El responsable de esta métrica debería ser un miembro del núcleo (miembros más comprometidos) de la comunidad. Su toma está soportada, pues las herramientas de control de versiones casi siempre permiten conocer cada operación realizada a través de mensajes enviados a las listas de correo.

- Tasa de commits o aportaciones erróneas (debieron ser desechados posteriormente) por unidad de tiempo.

  Permite medir la relación existente entre el número total de commits efectuados en un período de tiempo versus el número de commits que debieron ser desechados posteriormente.

$$TCE = \frac{CE}{TC} * 100$$

  Donde:

  CE = Commits erróneos.

  TC = Total de commits en el período de tiempo establecido.

TCE = Tasa de commits erróneos (se expresa en porcentajes)

Complementaria a la métrica anterior; un valor exagerado puede considerarse un síntoma de mala planificación, debilidades en el diseño del software o falta de preparación de ciertos desarrolladores. En todo caso, es una medida de la eficiencia del proceso de desarrollo.

Cabe acotar que la métrica TC (Total de commits por unidad de tiempo) también puede ser utilizada para medir la actividad del proyecto en función del tiempo e identificar condiciones de desinterés o estancamiento del desarrollo.

- Tiempo de vida de un *bug* (defecto)

  Se define como el período que transcurre desde que se reporta un defecto, hasta que es liberada la versión del software que lo repara. Puede ser medido en unidades de tiempo o en número de liberaciones transcurridas desde que se reportó el bug hasta que fue resuelto. Adicionalmente, se puede clasificar a los defectos por severidad.

  Este valor es una medida de la velocidad de respuesta de la comunidad ante los reportes de problemas de los

usuarios, así como de la velocidad de evolución del producto, lo cual puede ser usado para medir la madurez y eficiencia del proceso de desarrollo.

Este es un ejemplo, de una métrica cuya responsabilidad recae en todos los desarrolladores. Puesto que la corrección de un error, es asignada a un desarrollador, es él quien debe encargarse de que esa información se registre, a menos que las herramientas de soporte lo hagan automáticamente. Cabe recalcar, sin embargo, que debe haber un encargado de manipular y presentar la información recabada por todos los miembros del equipo de desarrollo.

- Retraso en la entrega de liberaciones (Puede estar medida en unidades de tiempo)

  Esta métrica define el tiempo transcurrido entre la fecha fijada para la liberación de una versión del software, hasta que ésta es efectivamente realizada. Puede ser empleada como una medida de la efectividad a la hora de estimar el tiempo de estabilización y puede ser muy útil en las primeras iteraciones del ciclo, al permitir una mejora progresiva y paulatina de las estimaciones del tiempo. La responsabilidad de esta métrica puede asignarse a cualquier miembro comprometido del equipo.

Métricas del producto

Definen características intrínsecas del software que se está desarrollando. Deberían ser tomadas por cada versión liberada del software.

- Número de bugs reportados por *release* (liberación).

  Fácilmente soportada por las herramientas para reportes de defectos, debería ser responsabilidad de un solo miembro. Es una medida importante de la calidad del producto así como de la efectividad del período de estabilización. Un valor extremadamente elevado, puede indicar problemas en la estimación de tiempos, preparación de los desarrolladores, falencias en el diseño, entre otros factores.

- Popularidad

  La popularidad sirve para determinar la acogida del producto en el mercado. En el caso de un proyecto FOSS, podría definirse en base al número de descargas en un período de tiempo. Puede ser útil aunque imprecisa, si terceras personas deciden distribuir copias del software. Cabe acotar que es muy fácil de determinar con las herramientas de soporte.

Como se puede apreciar, ciertas métricas definen eventos que se suscitan por unidad de tiempo. Dicha unidad depende de lo acordado al inicio por el equipo de trabajo. Por otra parte, debe existir un análisis periódico de los datos recabados por parte de los responsables del proyecto durante cada iteración del ciclo de desarrollo. Dicho análisis debe servir para tomar decisiones que permitan que el proyecto continúe con mejores indicadores de eficiencia, madurez y salud que contribuyan a cumplir con los objetivos y metas propuestas.

CAPÍTULO IV.

SOFTWARE A DESARROLLAR

OPENASEL

En este capítulo se describe de forma breve la aplicación en la que se trabajó para evaluar el modelo de desarrollo, la misma que fue bautizada como openASEL y es una plataforma para administración de seminarios en línea, también conocidos como *webinars*.

Antecedentes

Con el avance de la tecnología en el campo de la redes de datos, las organizaciones cuentan cada día con tecnologías cada vez más avanzadas a nivel de hardware y software además de

notables mejoras en sus enlaces de datos. Protocolos como IP versión 6 y tecnologías como Internet 2 fueron diseñadas para aprovechar esta tendencia. Sus aplicaciones son innumerables: rápido acceso a la información, capacidades para realizar conferencias en línea, entre otras. Sin duda estas tecnologías han revolucionado campos como la educación, donde su principal aplicación es la educación a distancia. Muchas universidades se han percatado de este hecho, invirtiendo grandes sumas de dinero en acceso a las nuevas redes de Internet avanzado a fin de mejorar la calidad de la educación.

Análisis de la aplicación

Sin embargo, el acceso a este tipo de redes no es suficiente para explotar sus ventajas. Hace falta la infraestructura de software que permita lograrlo, la misma que por desgracia es aún escasa si nos enfocamos exclusivamente en las necesidades de organizaciones educativas como universidades.

Con el fin de contribuir en la solución de este problema, el software a desarrollar para la evaluación del modelo, es una aplicación pensada para que las universidades puedan administrar sus seminarios o clases a distancia, permitiendo a los administradores, profesores y estudiantes obtener la información sobre las clases planificadas a través de una interfaz web. La

herramienta tendría soporte para videoconferencia, mensajería instantánea, recursos compartidos (archivos) y presentaciones con pizarra compartida durante la realización de la clase.

Herramientas de soporte para desarrollar la aplicación

Una de las prácticas más recomendadas al momento de emprender un proyecto FOSS, es nunca empezar a desarrollarlo desde cero. Es preferible buscar alguna herramienta que implemente en parte lo que se necesita y trabajar sobre ella. Esto tiene dos implicaciones: se puede ahorrar mucho trabajo si se encuentra que alguien ha resuelto parte del problema y a la vez existe la posibilidad de poder contribuir con ideas a otro proyecto.

Por la razón expuesta, nos pusimos en búsqueda de herramientas libres que permitieran realizar videoconferencias. Las evaluamos y al final escogimos la plataforma AccessGrid 3.1 (43), la misma que soporta todas las funcionalidades de videoconferencia que necesitamos. Se encuentra escrita en el lenguaje de programación Python y está compuesta de una serie de herramientas de software que permiten instalar con relativa facilidad una infraestructura para videoconferencias siempre y cuando se cuente con el hardware requerido.

El siguiente paso, fue definir cómo desarrollar la interfaz web de administración. A fin de que toda la solución estuviera desarrollada con herramientas compatibles, se escogió para este efecto la plataforma para desarrollo web Django (44), la misma que se encuentra escrita en Python y se enfoca principalmente en el patrón de diseño MVC (Model-View-Controller) a fin de garantizar un diseño conciso y de fácil mantenimiento.

Diseño de la solución

Con los requisitos definidos y las herramientas escogidas, procedimos a elaborar un diseño medianamente detallado de lo que se implementaría. Se procedió a dividir el problema en componentes y éstos a su vez en módulos de menor tamaño que nos permitieran repartir convenientemente el trabajo entre los miembros.

Arquitectura de la aplicación

El siguiente diagrama expone la arquitectura general de openASEL.

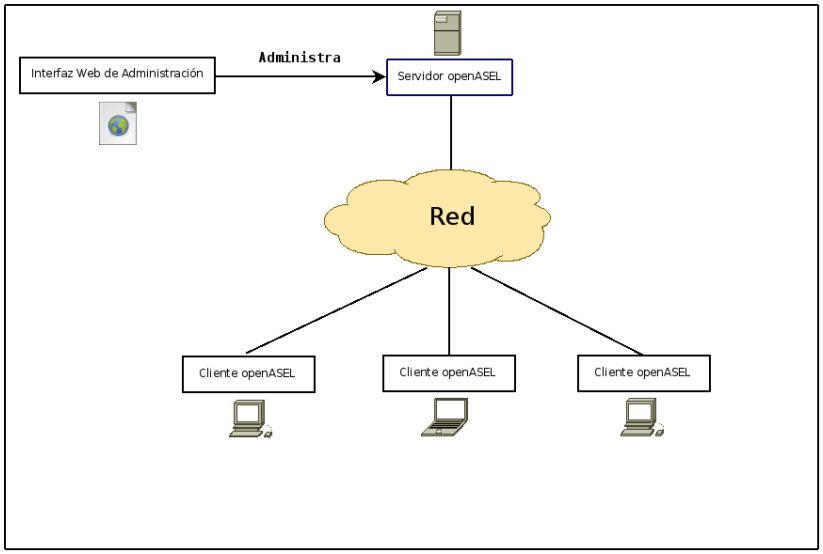

Gráfico 4.1. Arquitectura de OpenASEL

Como se aprecia en la figura anterior, la plataforma openASEL está compuesta de tres elementos que se describirán a continuación:

Interfaz Web de Administración

Es el componente que permite a los usuarios del sistema consultar la información sobre la planificación de clases virtuales. Permitirá entre otras cosas:

- Tareas de administración relacionadas a los usuarios y al servidor de videoconferencias utilizado.
- Planificación de seminarios en línea.
- Consulta de seminarios planificados.
- Invitación de personas a seminarios previamente planificados.
- Realización de un seminario.

Una aclaración sobre el último requerimiento. Cuando un seminario está a punto de empezar, es deber de los miembros iniciar sesión en la interfaz web de administración e indicar que desean ingresar al salón virtual. En ese momento, se realizará la carga de la aplicación cliente, cuya descripción es expuesta más adelante en esta sección.

Servidor openASEL

Es el componente encargado de administrar todos los recursos requeridos durante la realización de una videoconferencia y es administrado por el componente web descrito anteriormente. Se encarga principalmente de mantener sincronizados a todos los clientes y por ende es quien almacena la información de estado del seminario.

Debido a que la plataforma AccessGrid brinda las funcionalidades mencionadas, decidimos escoger la herramienta VenueServer, que es un servidor de videoconferencias en sí. Dicha herramienta, al igual que la mayor parte de los componentes de AccessGrid, intercambia información a través de servicios web. Por ende la interfaz web de administración se comunica con el servidor openASEL a través del protocolo SOAP que corre sobre HTTP. De esto se puede concluir, que estos componentes están diseñados para correr en diferentes computadores.

Cliente openASEL

Es la aplicación que corre en la computadora del usuario y le permite interactuar en un seminario virtual. La plataforma AccessGrid posee una herramienta llamada VenueClient. Es un cliente de escritorio que permite entre otras muchas cosas interactuar en un salón virtual. Uno de los problemas de AccessGrid es la interdependencia entre los componentes del sistema. Aunque un usuario desee utilizar solamente el cliente, deberá instalar todos los componentes. Por esa razón y a fin de que todo se hiciera a través de la web, se decidió implementar un nuevo cliente usando el lenguaje de programación Java y la tecnología Java Web Start Applications la cual permite que los usuarios se descarguen la aplicación desde su navegador al momento de ejecutarla por lo que no tendrán necesidad de instalar ninguna dependencia adicional, excepto Java.

CAPÍTULO V.

APLICACIÓN DEL MODELO

En el capítulo anterior se describió la aplicación desarrollada. En este capítulo se describe el proceso mediante el cual se aplicó el modelo MOCCA para desarrollar OpenASEL. Se mencionan las métricas escogidas para controlar la salud del proceso, las herramientas que tuvimos que usar para poder llevar a cabo la medición de manera más adecuada y la comunidad que tuvimos que formar para simular un ambiente de desarrollo colaborativo.

Ambiente de desarrollo

Para poder evaluar la validez del modelo MOCCA, era imperativo desarrollar la aplicación en un ambiente colaborativo, como la gran mayoría de los proyectos de software libre. El propósito

original fue contactar a 2 universidades dentro del Ecuador, para desarrollar OpenASEL de manera conjunta con programadores separados geográficamente y cuyos medios de comunicación sean estrictamente los establecidos por el modelo. Debido al desinterés por parte de las universidades, se decidió formar la comunidad con programadores de la misma universidad con la ayuda del grupo KOKOA (Comunidad de Software Libre de la ESPOL) (45). La comunidad estuvo compuesta por 6 desarrolladores.

Definición de la licencia del producto

Como se explicó en el capítulo III, la definición de la licencia del producto, es una tarea que no debe ser tomada a la ligera y debe ser consistente con los objetivos del proyecto. Uno de los objetivos específicos planteados por esta tesis, fue brindar a las universidades con acceso a una red de Internet avanzado, una plataforma completamente libre para administrar seminarios en línea. Debido a que los derechos de propiedad de intelectual del producto no pertenecen exclusivamente a los autores de este documento, nos limitamos a sugerir que openASEL sea distribuido bajo los términos de la licencia GPL en su versión 2 a fin de maximizar la protección de los derechos de los usuarios, en este caso las universidades, así como facilitar la formación de una comunidad de desarrollo que puede contar con miembros de otras universidades, bajo la dirección de desarrolladores de ESPOL.

Finalmente, vale la pena acotar que la licencia utilizada por AccessGrid (AGTPL) es compatible con la licencia propuesta.

Herramientas de soporte para la toma de métricas

Existen un sinnúmero de herramientas FOSS que permiten soportar el desarrollo de software de forma colaborativa. Se escogió un subconjunto de las mismas y se las evaluó a fin de escoger las más apropiadas para el proyecto. Cabe recalcar que el modelo no recomienda en ningún momento el uso de una herramienta en particular, por lo que la viabilidad de las mismas depende exclusivamente del proyecto de software donde va a ser utilizada. La siguiente tabla expone las categorías de herramientas de soporte existentes, las herramientas evaluadas y las finalmente escogidas.

Categoría	**Herramientas evaluadas**	**Herramienta escogida**
Sistema Operativo para el servidor	Ubuntu 7.10 Debian Etch CentOS 5	Ubuntu 7.10 con Xfce Desktop
Publicación y manejo de contenidos	Joomla Drupal Plone	Drupal
Administración	Trac	GForge

general del proyecto	GForge	
Control de versiones	CVS Subversion GNATS Bitkeeper	Subversion
Bug Tracking	GForge Bugzilla JIRA* Agile Edge*	GForge
Lista de correos	Mailman	Mailman
Foros de discusión	DrupalBB GForge	GForge
Servidor de correos	Exim Sendmail Qmail Postfix	Postfix
Wiki del proyecto	MediaWiki Wikiwig GForge Wiki Plugin	GForge Wiki Plugin
Obtención de estadísticas en torno a la actividad del proyecto.	Munin GForge SVNStat	GForge SVNStat

Tabla 2. Herramientas de soporte existentes, evaluadas y escogidas

En el caso del sistema operativo escogido para el servidor, se optó por la distribución Ubuntu Server 7.10 por su facilidad de uso y configuración. Debido a que la utilización de la plataforma AccessGrid 3.x (sobre la que está basada la aplicación a desarrollar para evaluar el modelo) requiere la instalación de un entorno gráfico, se optó por Xfce Desktop debido a su robustez y bajo consumo de recursos de hardware.

La selección de Drupal (46) como gestor de contenidos para el sitio web del proyecto, se originó debido al interés de quienes escriben esta tesis, en aprender a utilizarlo, dada su gran difusión y conocida facilidad de uso.

En relación al control de versiones, las falencias del que hasta hace poco fuera el estándar de facto (CVS), nos llevaron a optar por Subversion (47).

La plataforma GForge (48), permite integrar en una sola distribución, todas aquellas herramientas que soportan el desarrollo de software de forma colaborativa, lo cual incluye:

- Soporte para múltiples proyectos
- Listas de correo
- Foros de discusión (vía integración con Mailman)
- Wikis (vía extensiones)
- Manejo de documentos

- Bug tracking
- Integración con los sistemas de control de versiones más populares (CVS y Subversion)

Finalmente, GForge permite obtener estadísticas sobre la actividad de los proyectos alojados, excluyendo a las listas de correo y al repositorio. Por esa razón, se debió implementar dos programas adicionales que permitieran calcular las métricas relacionadas a la actividad en la lista de correos y que complementaran los datos ya conocidos sobre los foros de discusión. Con relación al repositorio, SVNStat (49) ofrece excelentes reportes sobre la actividad en un repositorio Subversion.

Descripción de métricas a emplear

Si bien el modelo define un conjunto de métricas útiles para revelar la salud del proceso en cualquier instante de tiempo, las condiciones y unidades bajo las cuales se deben tomar dichas mediciones dependen exclusivamente del proyecto. Es así que previo al inicio del desarrollo, se debió particularizar lo estipulado en el modelo. La siguiente tabla muestra las métricas seleccionadas para esta instancia del modelo y las unidades escogidas.

Métrica	Unidades
Momentum o frecuencia de liberaciones	Liberaciones por mes
Número de correos enviados a la lista por unidad de tiempo	Total de correos en el período de análisis
	Promedio de correos por día
	Total de mensajes de foro en el período
	Promedio de mensajes de foro por día
Frecuencia de conflictos en el servidor de control de versiones	Conflictos por mes
Tasa de *commits* o aportaciones erróneas por unidad de tiempo.	Tasa de commits erróneos por mes
Tiempo de vida de un *bug*	Promedio del tiempo de vida en días laborables de los bugs resueltos en el período
	Promedio del tiempo de vida en número de liberaciones de los bugs resueltos en el período
Retraso en la entrega de liberaciones	Días laborables entre la fecha programada y la fecha real de la liberación
Número de bugs reportados por *release*	Número de bugs reportados y aceptados entre dos liberaciones
Popularidad	Número diario de descargas

Tabla 3. Lista de las métricas y sus unidades de medición definidas para esta instancia del modelo.

Como dato adicional, se debe recalcar que las métricas expuestas corresponden a métricas del proceso y del producto. Como se expondrá más adelante, algunas de ellas nunca pudieron ser tomadas debido a que no se pudo culminar en su totalidad el software propuesto. Aquellas que fueron regularmente evaluadas, corresponden al grupo que mide la actividad del proceso. Es destacable el hecho de que en un proceso estilo Bazar, es poco común tomar métricas del producto debido a un sinnúmero de factores inherentes al proceso, como las condiciones en las que los defectos son corregidos y el hecho de que habiendo una

comunidad muy grande de personas interesadas en el proyecto, cualquier defecto en el mismo será eventualmente corregido.

Finalmente, los indicadores relacionados a la liberación de versiones del software (Frecuencia de releases y Popularidad) se establecieron como responsabilidad de los líderes, si bien nunca pudieron ser medidos.

Procedimiento de toma de métricas

Los reportes de datos fueron elaborados periódicamente a intervalos de tiempo que oscilaron entre 5 a 15 días, dependiendo si hubo algún evento importante en dicho período, como la entrega de una tarea compleja dentro del desarrollo. Para ello, se emplearon las herramientas de soporte definidas en la sección “Herramientas de soporte para la toma de métricas” en este capítulo, aunque hubo que implementar un par de programas adicionales para obtener más granularidad en la información relacionada a los correos y mensajes de foros enviados por día.

Por último, es importante mencionar que todas las métricas que incluían unidades con períodos de tiempo en días, emplearon tan solo días laborables, por lo que se omitieron fines de semana y feriados.

Los líderes del proyecto fueron los encargados de dirigir el proceso de toma de las métricas, incluyendo la elaboración de los reportes.

Para los indicadores relacionados con la actividad en los canales de comunicación y defectos reportados (Número de correos y mensajes de foro enviados por unidad de tiempo y Número de *bugs* reportados por *release*), el uso de las herramientas de soporte, permitió que los datos fueran automáticamente recabados cada vez que se requería la elaboración de un reporte.

En el caso de las métricas relacionadas a conflictos o errores con el sistema de control de versiones, la colaboración de parte de todo el equipo fue vital, pues no hay forma de automatizar las mediciones. Fue responsabilidad de cada desarrollador reportar a los líderes, sobre la ocurrencia de un conflicto o aportación errónea.

CAPÍTULO VI.

ANÁLISIS Y PRESENTACIÓN DE LOS RESULTADOS

El principal objetivo del modelo es garantizar en cualquier momento información certera sobre la salud del proceso a fin de tomar las medidas del caso para encaminarlo al éxito. Con esa premisa y con el hecho de no haber llegado a una versión estable del software objetivo, nos enfocaremos en analizar la actividad del proceso en base a los datos recabados y tomando en cuenta los factores externos a los cuales estuvieron sujetos los desarrolladores y que sin duda afectaron notablemente los resultados.

Pruebas Realizadas

Como fue explicado en los capítulos 4 y 5, se decidió que la mejor prueba a la que podía ser sometido el modelo sería el desarrollo de una aplicación compleja, que involucre a varios programadores y contenga de ser posible varios módulos. Durante el periodo de implementación de OpenASEL, se tomaron diversas métricas relacionadas a la salud y eficiencia del proceso, sobre todo de sus canales de comunicación.

Análisis de Resultados

Los indicadores más precisos de la actividad de los miembros de un proyecto de desarrollo FOSS son los canales de comunicación del mismo, para nuestra instancia, la lista de correos y el foro. Proyectos de mayor tamaño optan adicionalmente por utilizar canales IRC u otros mecanismos de mensajería instantánea para comunicarse, no obstante estos medios presentan mucha dificultad al tratar de cuantificar la actividad.

El gráfico 6.1 muestra la actividad de la lista de correos en el período de desarrollo del proyecto.

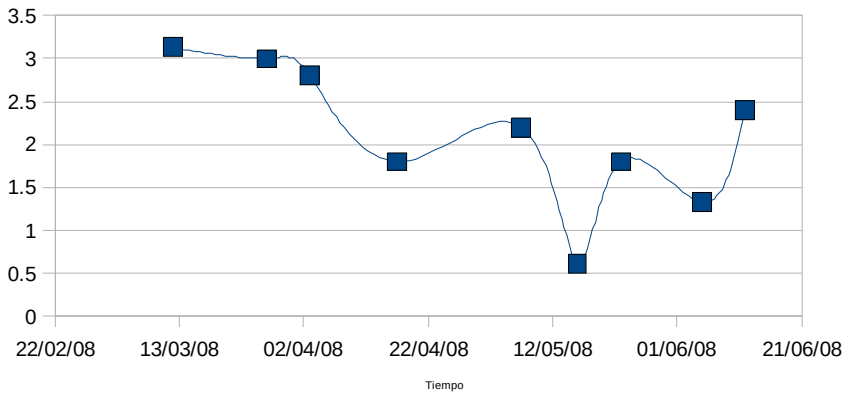

Gráfico 6.1. Promedio diario de correos enviados y aceptados en la lista de correos (Se consideraron solamente días laborables)

En primer lugar, es importante acotar que la lista de correos fue empleada como el principal medio de comunicación del proyecto, por lo que es el canal más fidedigno para evaluar la actividad del mismo. Un vistazo al gráfico, nos permite notar una tendencia a la baja en la actividad de la lista de correos durante el proceso, con un repunte al final del período de análisis. Ese comportamiento es predecible, pues al inicio del proceso se necesitó una intensiva comunicación para familiarizarse con el proceso de desarrollo. Una vez terminada esta etapa, cada grupo de trabajo se enfocó en sus actividades con lo cual la actividad decreció aparentemente, para luego incrementarse al final del período, cuando se estableció que se debía realizar una liberación parcial del software con la entrega de reporte. Este último hecho revela una característica de esta métrica: un descenso en su valor no necesariamente implica un decrecimiento real de la actividad del proyecto ni en su salud. Para emitir juicios sobre la real actividad, es imprescindible tener información sobre el contexto en el que fueron tomados los datos. Para proyectos de mayor trayectoria con múltiples liberaciones es de esperarse que esta métrica presente repuntes en las fechas cercanas a eventos destacados como la publicación de una nueva versión del software.

A diferencia de la lista de correos, la actividad en el foro de discusión fue una métrica poco contundente para nuestra instancia del modelo. El gráfico 6.2, muestra actividad en las primeras fases del desarrollo con un descenso sostenido y valores nulos en ciertos casos.

El foro de discusión fue utilizado como medio para discutir temas técnicos muy puntuales que tuvieran acceso público al considerarlos útiles para los futuros usuarios de la aplicación. Cuando no había temas puntuales de discusión, simplemente no se lo utilizaba. En proyectos FOSS con grandes comunidades de usuarios, la actividad en los foros es intensa, pues son los miembros ajenos al desarrollo quienes solicitan ayuda a través de este tipo de canales. Para estos casos, dicha métrica puede ser mucho más útil.

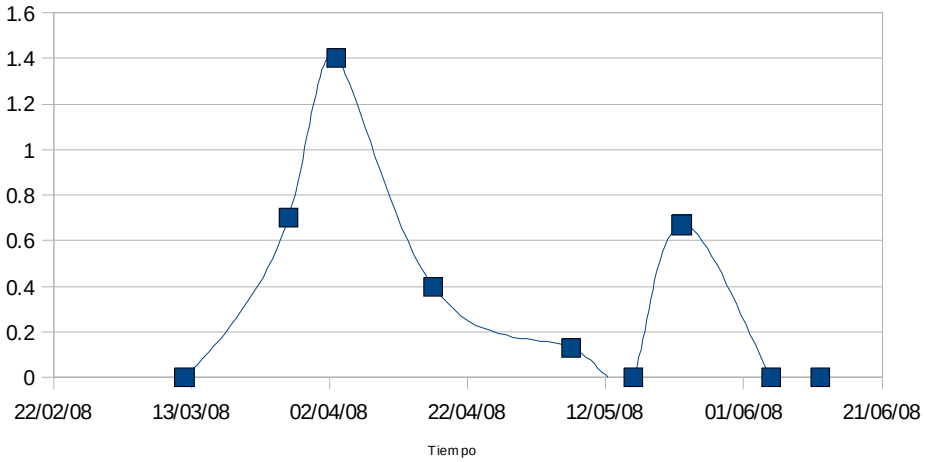

Gráfico 6.2. Promedio diario de mensajes de foro (Se consideraron solamente días laborables)

Además de analizar la actividad global del proyecto, hemos considerado necesario desglosarla por usuario. La razón es destacar una característica que es muy común en los procesos estilo Bazar: la mayor actividad en los canales de comunicación la concentran unos pocos desarrolladores, en la mayoría de los proyectos, los líderes de los mismos.

El gráfico 6.3 muestra la distribución de la actividad total en la lista de correos por usuario, al cabo de la última revisión al momento de escribir este reporte.

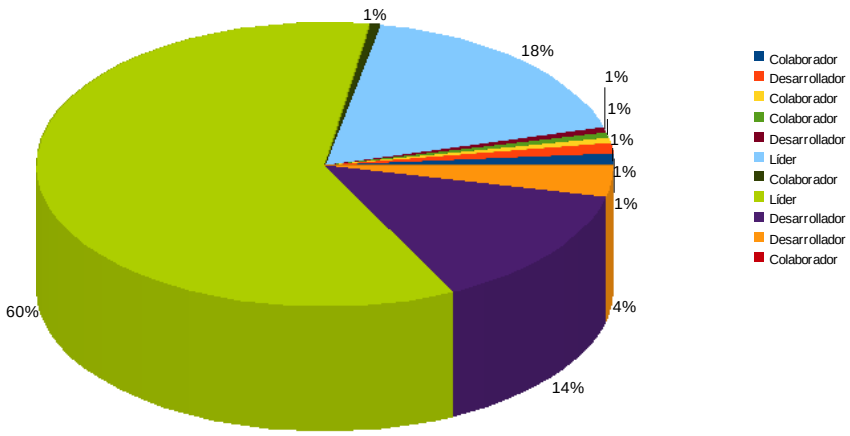

Gráfico 6.3. Distribución de la actividad en la lista de correos por miembro del proyecto (29 de febrero - 16 de junio)

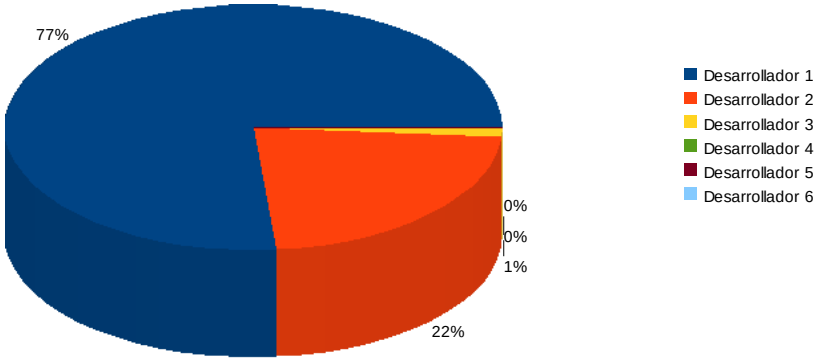

Gráfico 6.4. Distribución de las aportaciones de código realizadas en el repositorio por desarrollador (hasta el 18 de junio)

El gráfico 6.4 muestra la distribución de aportaciones al repositorio del proyecto por usuario, basados en todos los cambios realizados en los archivos del repositorio, la cual expone la misma tendencia que en el caso de la lista de correos, con el detalle adicional de que hay desarrolladores que parecen haber aportado muy poco. La razón es que el desarrollo se dividió en grupos de trabajo, cada uno con un líder. Todos trabajaron en equipo, pero ocurrió que en ocasiones era el líder quien al final subía los cambios al repositorio. Sin duda esas decisiones afectan al análisis cuantitativo, pues es difícil estimar la real aportación de cada desarrollador al código del proyecto.

Los valores de las líneas de código por desarrollador exponen una tendencia más marcada aún, a causa de este factor.

Finalmente, cada vez que se mida la actividad de los desarrolladores usando el repositorio, es imperativo considerar también la calidad de las mismas. Allí aparecen métricas como la tasa de conflictos y aportaciones erróneas, expuestas más adelante; no obstante en el camino identificamos que puede ser útil conocer el aporte en líneas de código de cada desarrollador para complementarlo con el número total de aportes en el servidor de control de versiones. La tabla 3 expone esta información al cabo de la última revisión del repositorio, la cual como dijimos está seriamente afectada por el argumento expuesto.

Autor	Cambios	Líneas de código	Líneas de código por cambio
Desarrollador 1	4958 (76.5%)	330118 (94.1%)	66.5
Desarrollador 2	1447 (22.3%)	20768 (5.9%)	14.3
Desarrollador 3	67 (1.0%)	58 (0.0%)	0.8
Desarrollador 4	1 (0.0%)	0 (0.0%)	0
Desarrollador 5	7 (0.1%)	0 (0.0%)	0

Tabla 4. Distribución de las aportaciones en el repositorio por cambios realizados en la estructura del repositorio y número de líneas de código subidas por desarrollador

Como se ha observado en los resultados mostrados hasta aquí, unos pocos usuarios concentran la mayor actividad en los canales de comunicación y en el repositorio; sin embargo esto no refleja una mala salud en el proyecto, ya que esta situación es muy común en los proyectos FOSS (34). Un problema que se identificó, fue que los desarrolladores que no eran líderes del

proyecto no acostumbraban a indagar suficiente sobre las tareas que tenían que hacer y eran los líderes quienes debían asegurarse de que todo estuviera claro. Al inicio, también afectó el hecho de que el ambiente fuera simulado y los desarrolladores pudieran tener contacto de forma relativamente fácil, evitando el uso de los canales de comunicación cuando era realmente más rápido hacerlo por este medio. Esto fue corregido con el paso del tiempo gracias a los informes realizados periódicamente. Nuevamente, la emisión de juicios sobre la salud del proceso queda sujeta a información de contexto.

Adicionalmente a la actividad en los canales de comunicación, el modelo también busca evaluar problemas al momento de realizar aportaciones al proyecto. Como casi todos los proyectos FOSS usan sistemas de control de versiones para alojar el código, se decidió elaborar métricas en torno a la calidad de las aportaciones en los repositorios. Esta idea la condensan las métricas de conflictos y tasa de aportaciones erróneas por unidad de tiempo. La métrica, tasa de conflictos por unidad de tiempo, nunca dejó de tener valores iguales a cero. El gráfico 6.5 muestra los valores de la segunda métrica en el período de análisis.

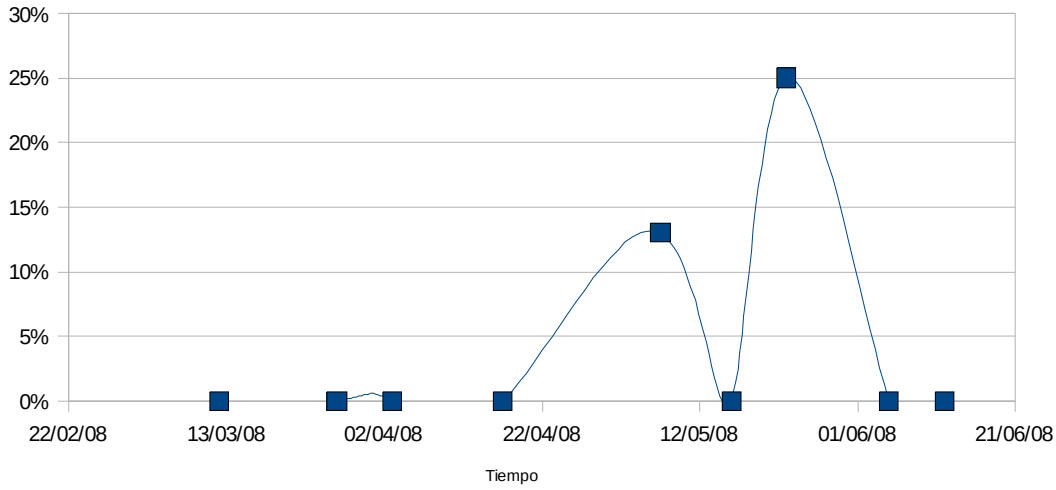

Gráfico 6.5. Valores de la métrica “Tasa de aportaciones erróneas por unidad de tiempo” (29 de febrero - 16 de junio)

¿Son estos datos, señales de que todo fue perfecto en el proyecto al momento de organizarse para escribir código? La respuesta para esta instancia es que no lo sabemos, pues los resultados dependen mucho de un factor clave que no ha sido tomado en cuenta hasta ahora: el tamaño de la comunidad de desarrolladores. Nuestra comunidad simulada contaba con 6 desarrolladores, con lo cual la probabilidad de generar conflictos por trabajar en una misma sección del proyecto es muy baja. Por otra parte, todos los desarrolladores carecían de experiencia en desarrollo FOSS, factor que si bien trató de ser contrarrestado con las jornadas de capacitación, no evitó que cometieran un par de

errores al momento de interactuar con el repositorio y las herramientas de gestión.

En relación al resto de métricas, la presencia de datos fue escasa por factores imputables a la instancia. Con un período de desarrollo de 4 meses basado en una herramienta que nadie conocía, era predecible. El desarrollo no pudo finalizar y por tanto no se obtuvo una versión estable del software, solo un par de versiones pre-alpha de acuerdo a la decisión de la comunidad. Esto se debió a un sinnúmero de factores externos dentro del cual excluimos la apatía de parte de los desarrolladores. El principal fue la complejidad de la herramienta base, en este caso, la plataforma AccessGrid. Aquí vale hacer una aclaración importante, pues recibimos mucha ayuda de parte de algunos miembros de esta comunidad, al igual que el ofrecimiento de una base de código que ellos habían implementado y que permitiría a nuestro proyecto interactuar con facilidad con AccessGrid. Implementar dicho módulo habría significado algunas semanas adicionales de investigación, diseño e implementación que nuestras restricciones de tiempo nos impedían asumir. Lamentablemente, la ayuda nunca llegó con lo cual fue imposible generar una versión completamente estable y funcional de la aplicación. Se procedió a trabajar con las partes en las que se podía avanzar sin ayuda externa y generar dos liberaciones de prueba que permitieran detectar defectos e inferir nuevos

requisitos. Sin embargo, esto es insuficiente para establecer conclusiones en torno al resto de métricas.

CONCLUSIONES

Con todo lo expuesto hasta ahora, es fácil establecer diversas conclusiones:

Una de las debilidades de los procesos estilo Bazar es que quienes colaboran, en muchos casos, lo hacen de forma voluntaria, con lo cual es imposible obligar a alguien a hacer algo. Esto se corroboró en el proyecto con la falta de colaboración de las universidades ecuatorianas y de la comunidad desarrolladora de AccessGrid en momentos en los que su aporte era vital para la consecución de nuestros objetivos. La solución a este tipo de inconvenientes es la elaboración de planes de riesgo que permitan definir un plan B ante cualquier inconveniente. Esta es una tarea que no se encuentra para nada reñida con la esencia del desarrollo Bazar por lo que debe ser considerada por quienes apliquen este modelo.

La evidencia cuantitativa recolectada durante el proceso de control de calidad es importante, pero insuficiente si no se conoce la información contextual de cuando fue recabada. Descensos en las métricas que miden la actividad del proyecto no implican necesariamente un aumento en la apatía de los miembros.

No todas las métricas definidas en el modelo sirven para todas las instancias del mismo. Comunidades con pocos desarrolladores pueden no requerir de métricas como la tasa de conflictos, así como comunidades muy jóvenes pueden no necesitar de las métricas relacionadas a la actividad en la liberación de versiones del software. Para estos casos, puede ser útil recurrir a otras métricas que si bien no han sido definidas explícitamente en el modelo, pueden resultar muy útiles y son derivables de las ya definidas. Un ejemplo de esta propuesta es el caso de la métrica *Retraso en la entrega de liberaciones*, que en nuestro proceso fue escasamente medida, pero que pudo haber sido extendida para usarla a nivel de tareas. Si no se puede medir la exactitud al estimar plazos para liberaciones del software, bien valdría hacerlo para las tareas y subtareas que se deben ejecutar para lograrlo, con lo cual habríamos obtenido una medida muy objetiva de nuestra habilidad de organización y estimación de plazos. Otro ejemplo podría ser extender la métrica *Tiempo de vida de un bug* a todos los ítems que se pueden reportar en un bug tracker como parches y solicitudes de nuevos requerimientos.

El modelo expone una serie de prácticas recomendadas para garantizar pleno conocimiento del estado del proyecto en cualquier instante de tiempo; sin embargo no ofrece ninguna recomendación de cómo solucionar un caso de apatía de parte de los miembros, pues dicha parte es muy subjetiva y completamente dependiente del proyecto a emprender y de las condiciones bajo las cuales se arranca. Contar con una comunidad motivada (aunque sea pequeña) de personas interesadas con canales efectivos de difusión de la idea, son buenos signos de progreso; por otra parte si no se cuenta con esto, la tarea de lograr que el proyecto continúe y gane adeptos se vuelve muy difícil.

También es importante aclarar que la falta de presupuesto no permitió que sigamos adelante más tiempo con el proyecto. En un ambiente de desarrollo de software libre, los usuarios se ven unidos y motivados por la necesidad de solucionar un problema común para todos. En nuestro caso, tuvimos que simular este ambiente mediante la inclusión de desarrolladores que no necesariamente compartían la necesidad de solucionar un problema, y tuvimos que incentivarlos de manera económica para garantizar su activa participación en el proyecto. Debido a esto, una vez que se agotaron los recursos, se tuvo que detener el proceso de desarrollo.

Para finalizar, se puede afirmar que el modelo propuesto es válido en gran parte, mas no en su totalidad. Prueba de esto es el hecho que no logramos conseguir nuestra meta inicial, liberar un prototipo estable. Aunque se ha mencionado anteriormente que esto se debió en gran parte a motivos externos al modelo y a problemas inherentes al desarrollo voluntario de comunidades de código abierto, no es razón válida para decir que el modelo es libre de toda culpa. Esto significa que, aunque el modelo es un gran paso hacia la estandarización del proceso de desarrollo de software libre, existen leves mejoras que pueden ser aplicadas para perfeccionarlo.

ANEXOS

Anexo A: Estado del proyecto openASEL

Este apartado tiene como objetivo describir el estado del proyecto openASEL al finalizar el proceso de desarrollo descrito en la parte principal de esta tesis a fin de facilitar cualquier iniciativa de continuarlo por parte de cualquier persona o grupo de desarrollo. Para ello, hemos dividido esta sección en dos partes:

- Cronograma de trabajo y esfuerzo invertido
- Estado de la aplicación

Cronograma de trabajo y esfuerzo invertido

A fin de comprender a cabalidad cómo se ejecutó este proyecto, es necesario revisar las actividades efectuadas por etapas. El proyecto tuvo como objetivo definir y evaluar un modelo desarrollo de software al estilo bazar, por ende, analizaremos el cronograma de trabajo y el esfuerzo invertido en las fases de definición de la metodología de desarrollo y aplicación a través del proyecto openASEL.

La etapa de definición del modelo, fue llevada a cabo exclusivamente por quienes escriben esta tesis y por nuestra directora. El anexo B, incluye un diagrama de Gantt con todas las tareas efectuadas durante esta fase. Nótese que previo al inicio del desarrollo del software que serviría para evaluar y validar el modelo, se realizaron unas jornadas de capacitación, a fin de entrenar a los futuros desarrolladores en el modelo y en las

herramientas a utilizarse en el desarrollo. La tabla 5 expone los eventos llevados a cabo para cumplir este cometido.

Evento	Tipo	Duración
MOCCA – Modelo Controlado para código abierto	Exposición	2 horas
OpenASEL – Administración de Seminarios en Línea (Arquitectura y requerimientos)	Exposición	2 horas
Herramientas de soporte para desarrollo FOSS	Taller práctico	6 horas
Introducción a Python	Taller práctico	6 horas

Tabla 5. Exposiciones realizadas en las jornadas de capacitación previas a la aplicación del modelo. Los documentos de soporte empleados pueden ser encontrados en (50)

En la segunda etapa del proyecto se procedió a aplicar el modelo definido en el proceso de desarrollo del software openASEL. En esta ocasión contamos con un equipo fijo de 7 personas: 2 líderes del proyecto (quienes escriben este documento), 4 desarrolladores contratados a 20 horas semanales por un período de 4 meses y la directora de tesis. Adicionalmente, contamos con la ayuda ocasional de 2 miembros claves del equipo de desarrollo del proyecto AccessGrid que brindaron su ayuda a través de la lista de correos del proyecto.

Finalmente, la tabla 6 muestra los valores de las métricas utilizadas para estimar el esfuerzo invertido en la fase de aplicación del modelo (desarrollo de openASEL). Los valores han sido tomados a partir de la última revisión del repositorio. Dichas cifras pueden resultar útiles, dado que revelan la productividad alcanzada por el equipo de desarrollo en las condiciones establecidas por el modelo de procesos y pueden servir como referente para evaluar el impacto de cualquier mejora o variante introducida al modelo original. De igual forma, han sido contrastadas con los resultados arrojados por el modelo de estimación de costos, COCOMO.

Métrica	Valor
Esfuerzo invertido	
Número de desarrolladores	6
Tiempo de desarrollo (meses)	4
Horas diarias*	4
Esfuerzo (personas-mes)*	12
Esfuerzo estimado por COCOMO	
Líneas físicas exclusivas de código escrito	5707
Esfuerzo (personas-mes)**	14.94

Tabla 6. Información sobre el esfuerzo invertido en la fase de evaluación del modelo (desarrollo del software openASEL.

* Los desarrolladores fueron contratados a medio tiempo

** Se utilizó el modelo básico COCOMO: $E = 2.4 \quad KSLOC^{1.05}$

Estado de la aplicación

Debido a restricciones de tiempo y recursos, openASEL no pudo terminar en una versión estable, sin embargo se liberaron dos versiones algo prematuras (51) (prealphas 1 y 2) con el objetivo de recibir la mayor

retroalimentación posible por parte de la comunidad universitaria y despertar el interés en otros estudiantes. Como se pudo apreciar en el capítulo V, la plataforma openASEL fue concebida para la realización de seminarios o clases en línea (webinars) a través de la implementación de 3 componentes:

- Servidor openASEL
- Cliente openASEL
- Interfaz web de administración

El servidor openASEL no es más que una instancia del componente VenueServer, incluido en la distribución de la plataforma AccessGrid. Su función es gestionar los recursos requeridos para la realización de las videoconferencias. El servidor dedicado para la ejecución del proyecto (52) fue instalado con todo el software necesario para funcionar como un servidor openASEL.

La interfaz web de administración es el componente que permite configurar el comportamiento del servidor openASEL y a la vez brinda a los usuarios de un sitio común a través del cual acceder a los seminarios, obtener información sobre su planificación y registrarse en los mismos. La tabla 7 es un recuento de las funcionalidades soportadas por la interfaz web de openASEL en su versión prealpha 2.

Funcionalidad	Soportada	Observaciones
Planificación de seminarios	Sí	
Consulta de seminarios planificados.	Sí	

Invitación de personas a seminarios previamente planificados.	Sí	Debe ser probada.
Realización de conferencia	No	Falta la integración completa con el cliente de videoconferencia.
Gestión de usuarios	Sí	

Tabla 7. Recuento de las funcionalidades soportadas por la interfaz web de openASEL al cabo del prealpha 2.

El cliente openASEL es una aplicación que corre en la computadora del usuario y le permite interactuar en un seminario virtual. Para acceder a ella, debe iniciar sesión en la interfaz web de administración cuando el seminario ha empezado. Esto se logra a través de una Java Web Start Application. Las funcionalidades soportadas por el cliente openASEL al cabo de la liberación efectuada son las siguientes:

- Transmisión / Recepción de video
- Transmisión / Recepción de audio
- Mensajería instantánea

Cabe recalcar que estas funcionalidades son las más importantes dentro del contexto de una aplicación de videoconferencia y por ende el cliente podría ser usado para conferencias sencillas.

Actualmente, el cliente openASEL tiene la capacidad de transmitir audio y video entre 6 clientes distintos, sin embargo, es escalable con suma facilidad. El único problema que tiene actualmente la transmisión de

audio y video es que las direcciones IP y los puertos por los cuales se transmiten y se reciben los datos son estáticos. Si se desea ejecutar la aplicación y conectarse con otros clientes, se necesita saber de antemano las direcciones IP de las computadoras y cambiarlas en el código del proyecto para poder llevarse a cabo la transmisión. Esto se debe a que el módulo encargado de unir la lógica del componente de la interfaz web con el cliente no pudo ser finalizado e integrado con el resto del proyecto. Sin este componente cuyo propósito era comunicarle al cliente las IP's de las computadoras que se estaban conectado al seminario, los usuarios que iban a ser parte del seminario y otros datos importantes, la información tuvo que permanecer estática.

En relación a la funcionalidad de mensajería instantánea, vale la pena aclarar que la plataforma AccessGrid no la implementa. Sin embargo, ofrece una interfaz que le permite al cliente de videoconferencias interactuar con un servidor XMPP, permitiendo el envío de mensajes instantáneos. Por esa razón el cliente openASEL incluye un ligero cliente XMPP el mismo que fue implementado haciendo uso de la librería Smack (53), utilizada en la implementación de Openfire (54), el servidor Jabber escogido durante el desarrollo del cliente openASEL.

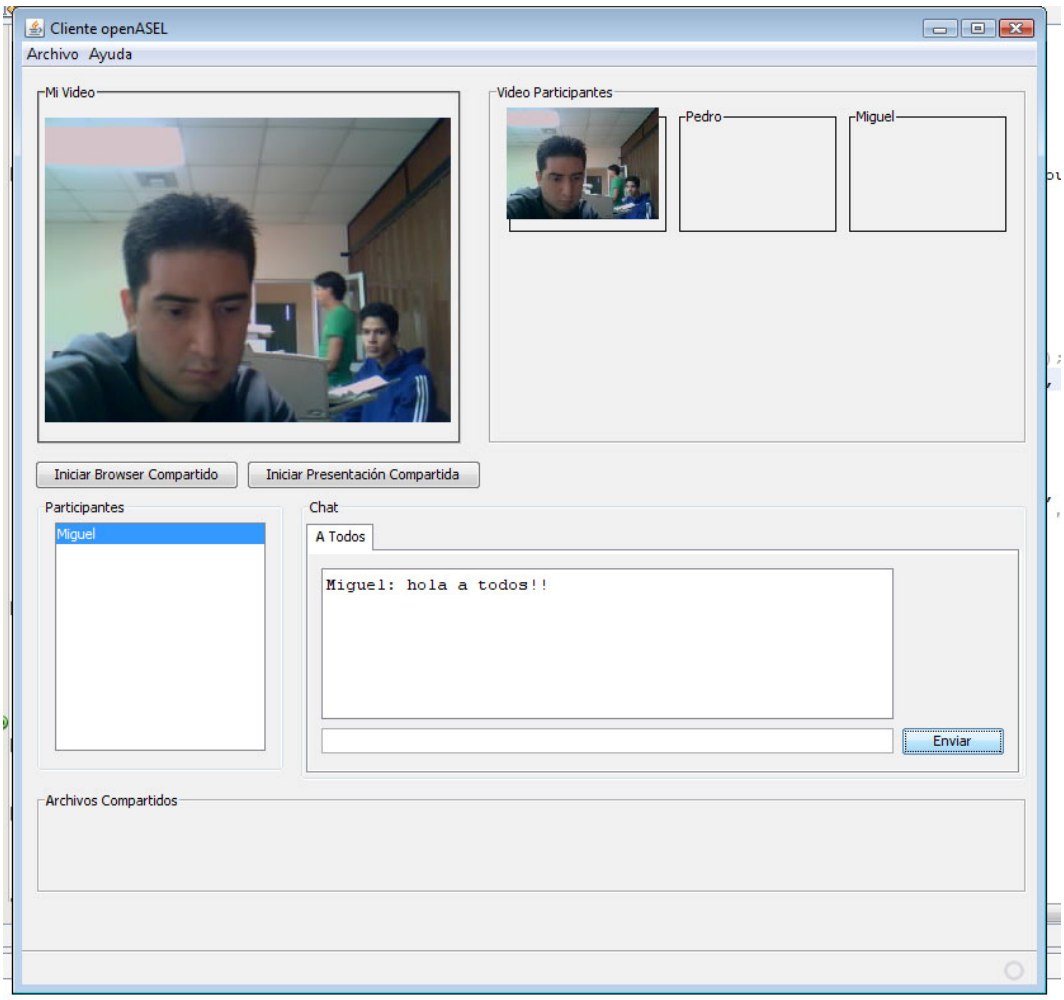

Gráfico A.1 Captura de pantalla de la aplicación cliente de openASEL

Anexo B: Diagrama de Gantt del proceso de definición y elaboración de MOCCA

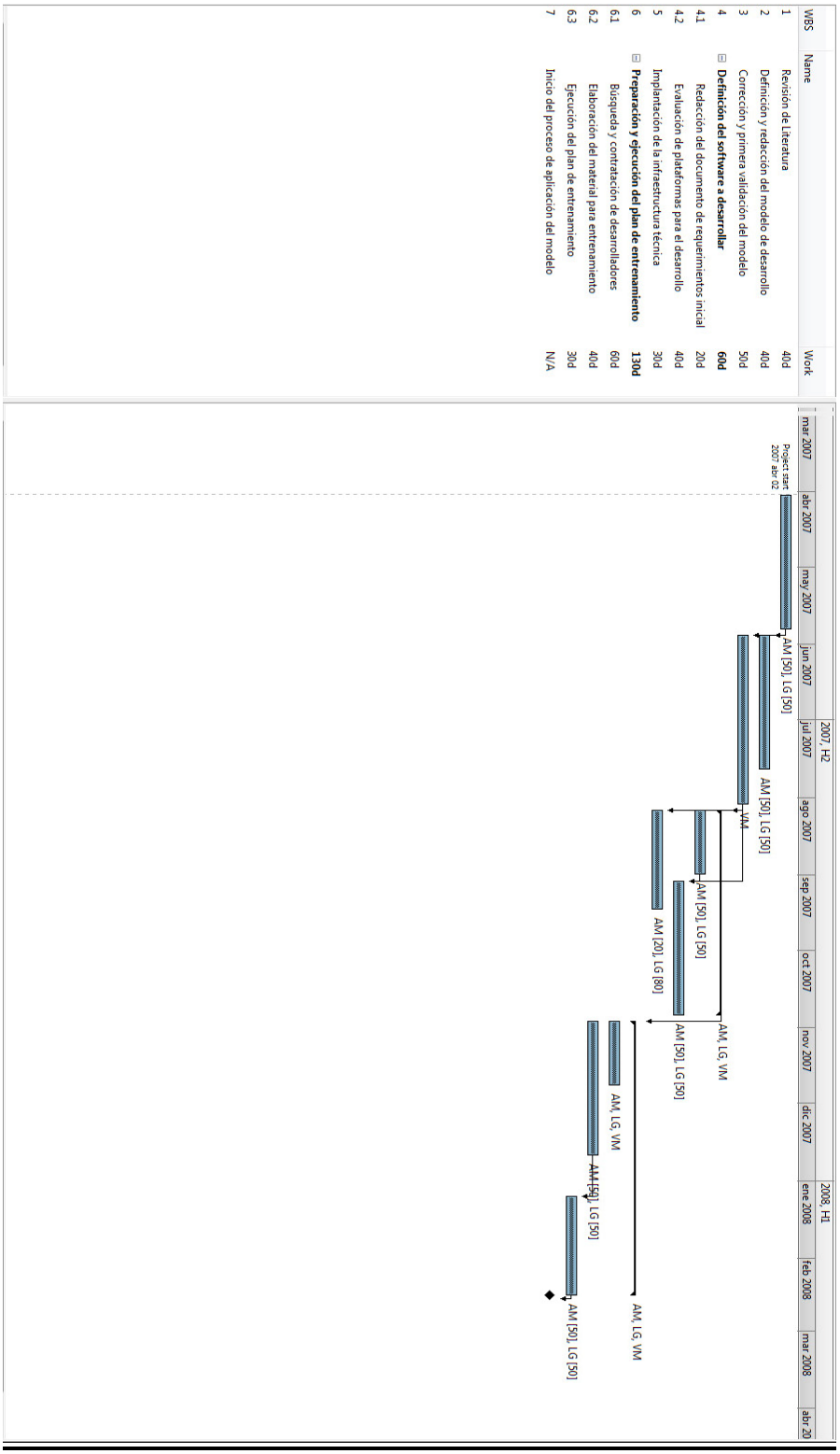

Gráfico B.1 Diagrama de Gantt del proceso de definición y elaboración de MOCCA

REFERENCIAS

(1) La Definición de Software Libre. Recuperado Septiembre 13, 2008 del sitio de, http://www.gnu.org/philosophy/free-sw.es.html.

(2) The Open Source Initiative. Recuperado Septiembre 13, 2008 de http://www.opensource.org/.

(3) http://www.gnu.org/home.es.html

(4) http://www.debian.org/

(5) Web oficial del proyecto gnuLinEx. Recuperado Junio 18, 2008 de http://www.linex.org/joomlaex/

(6) España, una potencia del software libre. (2004, Abril 16). Recuperado Junio 18, 2008 de http://www.laflecha.net/canales/softlibre/200404161/

(7) Alemania es el país con mayor uso de Software Libre. (2007, Junio 25). Recuperado Junio, 2008 de http://www.mastermagazine.info/articulo/11901.php

(8) French National Assembly switches to Linux. (2006, Noviembre 26). Recuperado Junio, 2008 de http://www.pcadvisor.co.uk/news/index.cfm?newsid=7687

(9) Brasil y la ofensiva Linux. (2005, Junio 3). Recuperado Junio, 2008 de http://news.bbc.co.uk/hi/spanish/business/newsid_4606000/4606701.stm

(10) 23,000 Linux PCs forge education revolution in Philippines. Linux still cheaper than heavily-subsidized Microsoft products (2008, Enero 28). Recuperado Junio, 2008 de http://www.computerworld.com.au/index.php/id;1163450117;pp;2

(11) Plan de alfabetización con Software Libre en Brasil (2008, Mayo 26). Recuperado Septiembre 2008 de http://www.mastermagazine.info/articulo/12841.php

(12) FLOSS deployment in Extremadura, Spain (2003). Recuperado Junio 19, 2008, de http://ec.europa.eu/idabc/en/document/1637/470

(13) El Presidente Correa autoriza la utilización del Software Libre en la Administración Pública Central (2008, Abril 11). Recuperado Junio 19, 2008 de http://www.presidencia.gov.ec/noticias.asp?noid=13318

(14) The GNU Project. Recuperado Septiembre 13, 2008 de http://www.gnu.org/gnu/thegnuproject.html

(15) http://savannah.gnu.org/

(16) http://sourceforge.net

(17) http://www.berlios.de/

(18) http://tigris.org

(19) June 2008 Web Server Survey. Recuperado Junio 29, 2008 de http://news.netcraft.com/archives/2008/06/22/june_2008_web_server_survey.html

(20) Browser Statistics. Recuperado Septiembre 13, 2008 de http://www.w3schools.com/browsers/browsers_stats.asp

(21) Ministerio de Ciencia y Tecnología (2004). Software libre: Uso y desarrollo en la administración pública venezolana: Lineamientos de políticas. Venezuela.

(22) Free Software in Europe: European perspectives and work of the FSF Europe. Recuperado Septiembre 13, 2008 de http://www.fsfeurope.org/documents/eur5greve.en.html.

(23) Roca M. y Castells M. (2006) El software libre en Catalunya y en España. Internet Interdisciplinary Institute, España.

(24) Bermejo M., et.al (2006). O Software libre nas entidades de Galiza (Setembro 2006): Estudo sobre a situación actual do Software Libre nas entidades de Galiza, PEME, Empresas informáticas, Concellos e Centros de ensino: informe técnico CESGA. Fundación Centro Tecnológico de Supercomputación, Galicia.

(25) Ghosh R.A (2006) Study on the: Economic impact of open source software on innovation and the competitiveness of the Information and Communication Technologies (ICT) sector in the EU. UNU-MERIT, the Netherlands.

(26) Robles, G., González-Barahona J., Centeno-González J., Matellán-Olivera V. and Rodero-Merino L. (2003). Studying the evolution of libre software projects using publicly available data. Proceedings of the 3rd Workshop on Open Source Software Engineering held at the 25th International Conference on Software Engineering, http://opensource.ucc.ie/icse2003/3rd-ws-on-oss-engineering.pdf (28. 01. 2004).

(27) Wheeler, D. (2000). Estimating Linux's Size, http://www.dwheeler.com/sloc (28. 01. 2004)

(28) German D. M. (2002). The evolution of the GNOME Project. In Proceedings of the 2nd Workshop on Open Source Software Engineering.

(29) González-Barahona, J., et. al (2003): Anatomy of two GNU/Linux distributions, pending publication in: Free/Open Source Software Development, edited by Stefan Koch, Idea Inc, Vienna.

(30) Robles G., Scheider H., Tretkowski I., and Weber N (2001). WIDI – Who Is Doing It? A research on Libre Software developers, http://widi.berlios.de/paper/study.pdf (28. 01. 2004).

(31) Levy, S. (2001). Hackers: Heroes of the computer Revolution, New York, Penguin Books.

(32) Madey, G., Freeh V., and Tynan R. (2002). The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory, http://www.nd.edu/~oss/papers/amcis_oss.pdf (28. 01. 2004)

(33) Park, R. E. et. al (1992). Software Size Measurement: A Framework for counting Source Statements, Technical Report CMU/SEI-92-TR-20, http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.020.html (28. 01. 2004).

(34) Raymond, Eric S. (2001) The Cathedral & the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly.

(35) Sommerville I. (2002). Socio Technical Systems. In *Software Engineering* (6th edition). Pearson Education, Mexico. pp. 33

(36) Sommerville I. (2002). Introduction to Software Engineering. In *Software Engineering* (6th edition). Pearson Education, Mexico. pp. 8

(37) Braude Eric J. (2003). Ingeniería de Software: Una perspectiva orientada a objetos. Grupo Alfaomega, México.

(38) Sommerville I. (2002). Socio Technical Systems. In *Software Engineering* (6th edition). Pearson Education, Mexico. pp. 46

(39) Kan Stephen (2003). Metrics and Models in Software Quality Engineering (2nd edition). Pearson Education.

(40) Koch Stefan. Free/Open Source Software Development.

(41) Fogel Karl (2005). Producing Open Source Software: How to Run a Successful Free Software Project.

(42) http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F

(43) http://www.accessgrid.org/

(44) http://www.djangoproject.com/

(45) http://kokoa.espol.edu.ec

(46) http://drupal.org

(47) http://subversion.tigris.org/

(48) http://gforge.org

(49) http://sourceforge.net/projects/svnstat

(50) https://proyectossw.espol.edu.ec/projects/openasel/

(51) https://proyectossw.espol.edu.ec/OpenaselWeb/

(52) https://proyectossw.espol.edu.ec

(53) http://www.igniterealtime.org/projects/smack/index.jsp

(54) http://www.igniterealtime.org/projects/openfire/index.jsp