UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Tesis para optar al Título de Magister en Informática

Mejora de la Calidad de los Prototipos Desarrollados en un Contexto Académico

Diego Vallespir Ligugnana
dvallesp@fing.edu.uy

Tutor de Tesis
Dr. Raúl Ruggia
ruggia@fing.edu.uy

Supervisor de Maestría
Dr. Álvaro Tasistro
tato@fing.edu.uy

Montevideo, Uruguay
Diciembre, 2006

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

El tribunal docente integrado por los abajo firmantes aprueba la Tesis de Investigación:

Mejora de la Calidad de los Prototipos Desarrollados en un Contexto Académico

Autor: Ing. Diego Vallespir

Tutor: Dr. Raúl Ruggia

Supervisor: Dr. Álvaro Tasistro

Carrera: Maestría en Informática - PEDECIBA

Calificación: Excelente

TRIBUNAL

Dra. Cristina Cornes _____

Dr. Juan V. Echagüe (Revisor) _____

Ing. Juan J. Cabezas _____

Montevideo, 22 de diciembre de 2006

Resumen

El desarrollo de software en las universidades se caracteriza por construir software que sirve como prueba de concepto o que es parte de un proyecto de investigación. Entendemos que este desarrollo de software no ha tenido la atención que merece desde el punto de vista de la Ingeniería de Software. Debido a que no se necesita obtener un *producto final*, la calidad de los productos desarrollados queda relegada. Por esto, los productos carecen de muchas de las propiedades de calidad que normalmente tienen los productos de software del mercado. Esto trae aparejados algunos problemas.

El objetivo general de esta tesis es reconocer y estudiar los distintos problemas que existen en el desarrollo académico y proponer una solución para los mismos.

El Grupo Concepción de Sistemas de Información (CSI) de nuestra Facultad ha detectado que los prototipos de software desarrollados en el grupo son de "mala" calidad. Entonces, el desarrollo de software en el grupo se toma como caso de estudio para esta investigación. Se estudian los problemas de desarrollo de software particulares del CSI y se propone una solución que se adapte al mismo.

Se realizan charlas y cuestionarios, de los cuales surge qué propiedades de calidad de los productos se deben mejorar para satisfacer las necesidades del grupo CSI. El objetivo específico de esta tesis es mejorar, en los productos que se desarrollen en el CSI, dichas propiedades de calidad. Para lograr esto, se debe contar con un marco de trabajo apropiado. Dicho marco debe contemplar todas las características particulares del desarrollo de software en el CSI. Además, no debe reducir la productividad de los desarrolladores ni aumentar la carga de trabajo de los investigadores del grupo.

Como ninguna de las propuestas más conocidas para la mejora de la calidad es aplicable de forma directa para el desarrollo académico de software, proponemos un Framework específico para este tipo de desarrollo, que llamaremos *Framework para la Mejora de la Calidad*.

El Framework está compuesto por un conjunto de Actividades que se agrupan en Niveles de Calidad y define un rol de Responsable de Calidad. Las Actividades deben ser realizadas por los desarrolladores de forma de alcanzar el Nivel de Calidad deseado. El Responsable de Calidad conoce el Framework y ayuda a los grupos de desarrollo en la aplicación del mismo.

Todas las Actividades que componen el Framework son actividades técnicas de construcción de software. No se incluye en el mismo ninguna actividad de gestión.

El Framework se pone en práctica en once proyectos que comienzan en abril de 2004.

Uno de los proyectos termina en diciembre de 2004 mientras que los restantes finalizan entre abril y junio de 2005.

Se realiza una encuesta a los desarrolladores y a los clientes de forma de conocer su opinión sobre la aplicación del Framework. El resultado es alentador; todos los desarrolladores y todos los clientes opinan que el Framework logró la mejora de la calidad de los productos desarrollados.

Creemos que el principal aporte de este trabajo es tener especificado y probado con éxito un Framework para el desarrollo académico de software. Este Framework respeta las características únicas de este tipo de desarrollo y, a la vez, consigue mejorar la calidad de los productos obtenidos.

Entendemos que la aplicación del Framework produjo la mejora de la calidad durante el desarrollo y también creemos que estos productos son mejores que productos de años anteriores desarrollados en el CSI. En particular, estos productos se desarrollan de forma más controlada, por lo que tienen menos defectos y estos son menos graves. También pensamos que los productos desarrollados son más fáciles de modificar, extender, reutilizar e integrar.

Palabras clave: Ingeniería de Software, Desarrollo Académico de Software, Ingeniería de Requerimientos, Verificación y Validación, Métricas de Diseño.

A mi esposa Mari,
a mis padres Nelly y Nadal,
a mi hermana Laura y a mi cuñado Pablo.

Agradecimientos

Agradezco a todas las personas que de una u otra manera estuvieron relacionadas con el proceso de escribir esta Tesis.

A Raúl Ruggia por ser mi Tutor, ya que sin su ayuda y su minuciosa lectura esta tesis no hubiera sido posible.

A Álvaro Tasistro por ser mi Supervisor durante mis estudios de maestría y entusiasmarme en todo momento. También le agradezco por revisar la tesis a pesar de que no era su tarea.

A todos los que leyeron la tesis en busca de mejoras y errores, incluso no conociendo de informática: mi esposa Mariana Poggi, mi hermana Laura Vallespir, Adriana Marotta, Jorge Triñanes y muy especialmente a mi padre Nadal Vallespir quien leyó la tesis más veces que yo. Todos ellos ayudaron en el refinamiento sucesivo de este texto.

A Regina Motz, Adriana Marotta, Raúl Ruggia, Federico Piedrabuena y Gustavo Vazquez por ser ser directores de los proyectos de grado del estudio de campo. A todos los muchachos que trabajaron como desarrolladores en los distintos proyectos: Ignacio Abel, Pablo Giampedraglia, Alan Kind, Sharon Benasus, Martin Cabrera, Sergio Perez, Veronica Giaudrone, Marcelo Guerra, Marcelo Vacaro, Maximiliano Panario, Fernanda Sorribas, Gustavo Signoreli, Javier Briosso, Patricia Gahn, Laura Gonzalez, Guillermo Roldos, Flavia Serra, Nicolas Doroskevich, Sebastian Moreira, Juan Carlos Campot, Jose Pedro Campot, Gaston Dodera, Rodolfo Amador, Fabricio Alvarez, Karina Brito, e Isabel Reolon . Sin ustedes tampoco hubiese sido posible.

A Dina Wonsever y Héctor Cancela por haber confiado y dejarme ser parte del grupo de Recursos Humanos y Gestión de Enseñanza del InCo aunque estuviera en la mitad de la tesis.

A Rodolfo Paiz por no “taparme” de tareas cuando estaba terminando la tesis.

A toda mi familia por el aguante!!!

Índice general

Índice de figuras

Índice de tablas

1 Introducción

El desarrollo de software en las universidades se caracteriza por construir software que sirve como prueba de concepto o que es parte de un proyecto de investigación. Entendemos que este desarrollo de software no ha tenido la atención que merece desde el punto de vista de la Ingeniería de Software. Debido a que no se necesita obtener un *producto final*, la calidad de los productos desarrollados queda relegada. Por esto, los productos carecen de muchas de las propiedades de calidad que normalmente tienen los productos de software del mercado. Esto trae aparejados algunos problemas.

En este trabajo se busca definir una propuesta para la mejora de la calidad en este tipo de desarrollo.

Contenido

1. Contexto y Motivación

Además del desarrollo industrial de productos de software existen otros tipos de desarrollo. Uno de estos es el desarrollo de prototipos de software en las universidades. A este desarrollo lo llamaremos "desarrollo académico de software".

Normalmente, en el desarrollo académico de software se construyen prototipos que son pruebas de concepto o que son parte de un proyecto de investigación. Como estos prototipos son sólo construidos para probar nuevas teorías o conceptos, no se consideran aspectos de calidad de software durante el desarrollo.

Este desarrollo crece cada vez más en el mundo. Además, ha aumentado la cantidad de prototipos, desarrollados en universidades, que luego son convertidos en productos que van a ser instalados y usados en ambientes de producción. A estos productos los llamaremos productos finales.

Varias de las características del desarrollo académico de software son diferentes a las del desarrollo en la industria:[1]

- La **cantidad de personas** involucradas en un proyecto de desarrollo es reducida.
- Normalmente, quien oficia de **cliente** es un investigador de la universidad donde se realiza el desarrollo. Este investigador es quien está interesado en el software para probar alguna teoría.
- Los **desarrolladores**[2] son estudiantes de grado, estudiantes de posgrado o investigadores. Esto indica que no son desarrolladores consolidados ni tienen una marcada experiencia en el área de desarrollo. También muestra que el equipo de desarrollo es altamente cambiante. Normalmente, los estudiantes que trabajan en un proyecto están en el mismo por un año aproximadamente.

También en nuestra Facultad se desarrolla software. En particular, el Grupo Concepción de Sistemas de Información (CSI) del Instituto de Computación (InCo) de la Facultad de Ingeniería de la Universidad de la República[3] desarrolla software para saber, mediante productos de software, si las propuestas de investigación del grupo realmente funcionan. Estos desarrollos se realizan, en su mayoría, en Proyectos de Grado y Tesis de Maestría.

[1] En esta tesis se usa "desarrollo de software en la industria" para referirse a la industria de software que desarrolla productos finales.

[2] Llamamos desarrolladores a las personas que llevan adelante la construcción del producto de software.

[3] Grupo Concepción de Sistemas de Información: http://www.fing.edu.uy/inco/grupos/csi/
Instituto de Computación: http://www.fing.edu.uy/inco/
Facultad de Ingeniería: http://www.fing.edu.uy/

2. Problema y Objetivos

Debido a que en el desarrollo académico de software no hay una preocupación por la calidad de los prototipos, estos carecen de muchas de las propiedades de calidad que tienen los productos finales de software. Esto trae aparejados algunos problemas.

Además, el desarrollo académico de software no ha tenido la atención que merece desde el punto de vista de la Ingeniería de Software, por lo que se desconocen exactamente los problemas existentes. El **objetivo general** de esta tesis es reconocer y estudiar los distintos problemas que existen en el desarrollo académico y proponer una solución para los mismos.

2.1. Contexto del Grupo CSI y Objetivo Específico

El CSI ha detectado que los prototipos de software desarrollados en el grupo son de "mala" calidad. Entonces, el desarrollo de software en el grupo se toma como caso de estudio para esta investigación. Se estudian los problemas de desarrollo de software particulares del CSI y se propone una solución que se adapte al mismo.

Se realizan cuestionarios, charlas e intercambio de correos electrónicos con investigadores del grupo CSI de manera de conocer la situación.[4] Se descubre que la "mala" calidad detectada responde a que los prototipos:

- No funcionan como se espera o directamente no funcionan.
- No se pueden volver a instalar luego de desinstalados.

Además, es común que estos problemas no se identifiquen hasta finalizado el desarrollo, empeorando todavía más la situación.

Esta situación implica para el grupo los siguientes problemas:

- No se tienen garantías sobre el funcionamiento real de las propuestas de investigación que se quieren probar con los prototipos desarrollados.
- No se logra que los prototipos puedan llegar a tener un impacto industrial.
- No se pueden presentar *demos* de los prototipos en sesiones de conferencias.
- No se pueden realizar grandes desarrollos donde distintos prototipos se deban extender, modificar, reutilizar y/o integrar con otros prototipos.

[4] El apéndice A contiene algunos de los comentarios recibidos.

Entonces, se debe mejorar la calidad de los prototipos. La solución a este problema debe contemplar el contexto en el cual se enmarcan los proyectos desarrollados dentro del grupo CSI. A continuación se enumeran las características que deben ser consideradas.

1. Los **prototipos** son sistemas de información o similares, aunque existen prototipos que difieren bastante de esta definición.

2. La **duración de los proyectos** es normalmente corta. La mayoría dura entre ocho y catorce meses.

3. La **cantidad de personas** involucradas en un proyecto es de tres o cuatro integrantes. De estas personas, una es cliente y dos o tres son desarrolladores.

4. El **cliente** es un investigador del grupo CSI que quiere un prototipo.

5. Los **desarrolladores** son estudiantes de grado de la carrera de Ingeniería en Computación, estudiantes del posgrado en Informática del PEDECIBA[5] o algún integrante del grupo CSI. Aparecen otros desarrolladores, pero en menor proporción; por ejemplo, personas contratadas por convenio. Normalmente, estas también son estudiantes. Esto nos indica que los desarrolladores no son desarrolladores consolidados ni tienen una marcada experiencia en el área de desarrollo.

6. El **alcance de los proyectos** es determinado, en gran medida, por los plazos de los mismos. Por ejemplo, los proyectos de grado tienen una duración fija, por lo que el alcance se planifica para el tiempo prefijado.

7. Normalmente, el **lenguaje de desarrollo** usado en los proyectos es Java.

De las charlas y cuestionarios mencionados, surge qué propiedades de calidad de los productos se deben mejorar para satisfacer las necesidades del grupo CSI. Estas son:

- Cantidad de defectos y severidad de los mismos.
- Facilidad de instalación.
- Modificabilidad.
- Mantenibilidad.
- Capacidad de reutilización.
- Extensibilidad.
- Capacidad de integración con otros productos.

[5]Programa de Desarrollo de las Ciencias Básicas: http://www.rau.edu.uy/pedeciba/informat/

De estas charlas, surge también la necesidad de aumentar la productividad. Si bien este último punto no es un atributo del producto, es considerado parte del problema a resolver.

El **objetivo específico** de esta tesis es mejorar, en los productos que se desarrollen en el CSI, las propiedades de calidad que se mencionaron. Para lograr esto, se debe contar con un marco de trabajo apropiado. Dicho marco debe contemplar todas las características particulares del desarrollo de software en el CSI. Además, el mismo no debe reducir la productividad de los desarrolladores ni aumentar la carga de trabajo de los investigadores del grupo.

3. Calidad de Software

ESTA tesis tiene como tema central la calidad de los productos de software. Si bien el tema calidad de software abarca muchos tópicos, las particularidades del problema, descritas en la sección 2, acotan los mismos.

Se brindan a continuación dos definiciones de calidad de software, una de la IEEE y otra de ISO.

Calidad - IEEE 610.12[IEE90]. Es el grado en que un sistema, componente o proceso cumple con los requerimientos especificados y con las necesidades o expectativas del cliente o usuario.

Calidad - ISO 8402[ISO95]. Es la totalidad de las características de una entidad que influyen en su capacidad de satisfacer necesidades explícitas e implícitas.

No encontramos una respuesta directa para resolver el problema planteado. El desarrollo académico de software no es tenido en cuenta en las distintas soluciones encontradas a los problemas de calidad del software.

Sin embargo, existen modelos, procesos, métodos y actividades que se han propuesto para mejorar diversos problemas relacionados con la calidad de software. Normalmente, estas propuestas son pensadas y concebidas para cierto tipo de proyectos y/o productos de la industria. Entonces, corresponde analizarlas, aunque no sean una respuesta directa a nuestro problema, y estudiar si se pueden reutilizar ideas y/o partes de los mismos.

Se tienen en cuenta en este trabajo las propuestas: Capability Maturity Model [PWG+93, PCCW93], Capability Maturity Model Integration, Personal Software Process [Hum00, Hum95], ISO 9001:2000 [ISOa], RUP [IBM] y XP [BA04]. Estos, junto con el estudio de su posible uso en el grupo CSI, se presentan en la sección 1 del capítulo "Estado del Arte".

También se consideran especialmente relevantes para esta tesis las áreas de Ingeniería de Requerimientos (IR), Verificación y Validación (V&V), y Métricas y Medidas de Diseño. Estas se presentan en las secciones 2, 3 y 4 del capítulo "Estado del Arte". La discusión de la adaptabilidad y posible uso de estas áreas se realiza en las mismas secciones.

En la medida que dichos modelos no se pueden aplicar directamente a nuestro caso particular, se intenta reutilizar sus ideas. Esto se hace de las siguientes maneras:

- Tomando actividades que proponen los modelos y ajustándolas a las necesidades del desarrollo en el CSI.
- Persiguiendo los mismos propósitos y objetivos que definen los distintos componentes de los modelos, para conseguirlos en el desarrollo en el CSI.

Así, por ejemplo, de la Gestión de Requerimientos de CMM se toma su propósito: *"Establecer un entendimiento común entre el cliente y el proyecto de software..."*. Si bien nuestra propuesta no cumple con la KPA[6] Gestión de Requerimientos de CMM, ni pretende cumplirla, esta es considerada de forma global a través de su propósito. Trabajando de esta forma, se logra considerar los modelos y extraer partes utilizables para el desarrollo en el CSI.

4. Solución Propuesta y Realización

Como ninguna de las propuestas más conocidas para la mejora de la calidad es aplicable de forma directa para el desarrollo académico de software, proponemos un Framework específico para este tipo de desarrollo, que llamaremos *Framework para la Mejora de la Calidad*. En el capítulo 3 se presenta la propuesta completa.

La propuesta debe ser *liviana* y de fácil aplicación debido a las características del desarrollo académico de software y, en particular, a las características del desarrollo en el CSI. La parte central del Framework es un conjunto de actividades a realizar por los desarrolladores. Estas actividades tienden a mejorar la calidad del producto final.

Las actividades dentro de la ingeniería de software se pueden dividir en actividades de *construcción* y actividades de *apoyo* a la construcción. Las actividades de construcción son aquellas que pertenecen a las disciplinas de Requerimientos, Diseño, Implementación y Verificación. Estas disciplinas afectan directamente al producto que se quiere construir. Las actividades de apoyo pertenecen a disciplinas como Gestión de Proyectos, Gestión de la Calidad y Gestión del Cambio, entre otras. Estas afectan de forma indirecta al producto de software.

[6]KPA - Key Process Area. Este y otros conceptos de CMM se estudian en el capítulo "Estado del Arte".

Para el Framework se toman solamente actividades de construcción. Esta decisión se basa en que entendemos que para este tipo de desarrollo las actividades de apoyo aportan poco para los objetivos que se quieren lograr. Esto se debe a que las actividades de apoyo son mayormente actividades de gestión, y entendemos que estas no deben formar parte de un Framework para el desarrollo académico. Creemos que considerando solamente actividades de construcción, en el contexto del CSI, se puede mejorar la calidad de los productos, mantener un Framework sencillo y no perder productividad.

La selección de las actividades del Framework, dentro de las actividades de construcción, surge de un análisis de los problemas de calidad encontrados en el desarrollo dentro del CSI y del estudio de los modelos de calidad existentes. Considerando otra vez que las actividades de gestión no son apropiadas para este tipo de desarrollo, se dejan de lado todas aquellas actividades de construcción que también son de gestión.[7] Entonces, se busca un conjunto de actividades puramente técnicas que puedan solucionar los problemas mencionados, dejando libradas a los grupos de desarrollo las pocas actividades de gestión que puedan tener que realizar. Las actividades del Framework se clasifican en actividades de Especificaciones,[8] Verificación y Validación, y Métricas y Medidas.

Se propone como parte del Framework un rol de Responsable de Calidad. Este rol lo debe llevar adelante una persona que conozca ampliamente el Framework. El objetivo de este rol es colaborar con los clientes y los desarrolladores para que realicen las actividades de forma óptima.

El Framework define distintos Niveles de Calidad. Cada nivel tiene asociado un conjunto de actividades que deben realizar los desarrolladores para que el prototipo se encuentre en ese Nivel de Calidad. Los niveles van del uno al seis, y a medida que se sube de nivel se agregan actividades, manteniendo las del nivel anterior. La Figura 1.1 muestra los niveles de calidad con las actividades que se agregan a cada uno. Estas actividades se describen en las secciones 2, 3 y 4 del capítulo 3.

Cada nivel define un tipo de prototipo que se espera conseguir al trabajar en ese nivel. Es el cliente quien define en qué Nivel de Calidad debe trabajar el grupo de desarrollo. Esta decisión se basa en la calidad que debe tener el producto final. Trabajar en un nivel implica ejecutar con responsabilidad profesional y ética las actividades que este agrupa. La Figura 1.2 muestra la relación entre un proyecto y el Framework propuesto.

Como aplicación de la propuesta, se pone en práctica el Framework en once proyectos que comienzan en abril de 2004. Uno de los proyectos termina en diciembre de 2004 mientras que los restantes finalizan entre abril y junio de 2005. Todos los grupos de desarrollo son de dos o tres personas, y se usa Java como lenguaje de implementación.

Se realiza una encuesta a los desarrolladores y a los clientes de forma de conocer su opinión sobre la aplicación del Framework. El resultado es alentador; todos los desarrolladores y todos los clientes opinan que el Framework logró la mejora de la calidad de los productos desarrollados. La descripción completa de la aplicación del Framework en los

[7] Por ejemplo, según la definición brindada, las actividades de gestión de requerimientos son tanto de construcción como de gestión.

[8] Se usa Especificaciones para referirse a la IR junto con otro tipo de especificaciones distintas a la de requerimientos. Por ejemplo, la especificación del comportamiento de una componente de software.

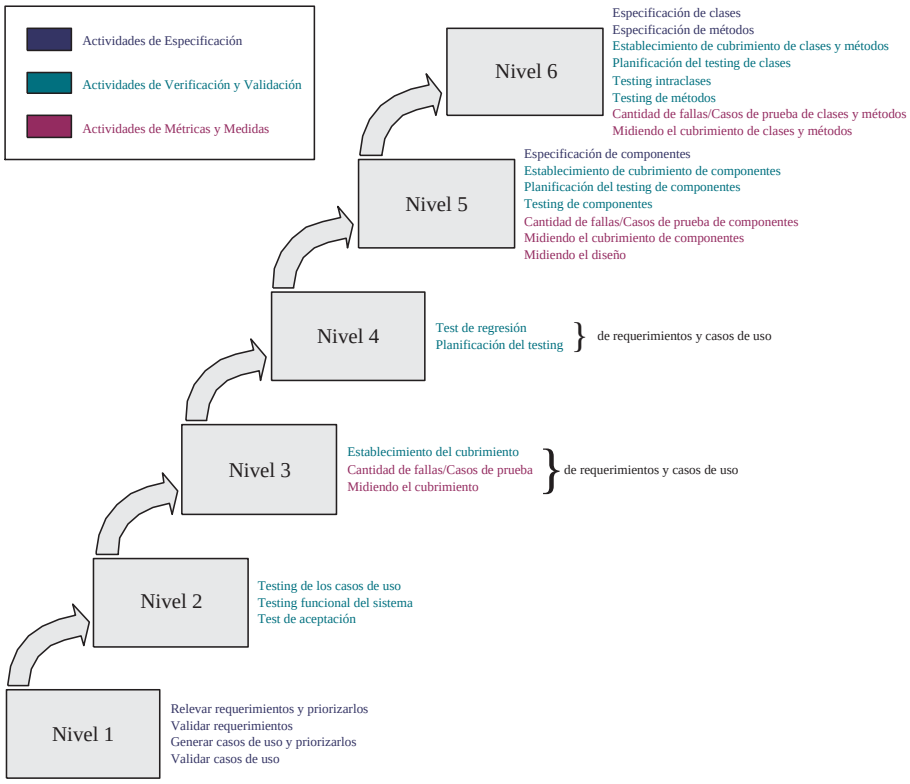


Figura 1.1: Niveles y Actividades del Framework

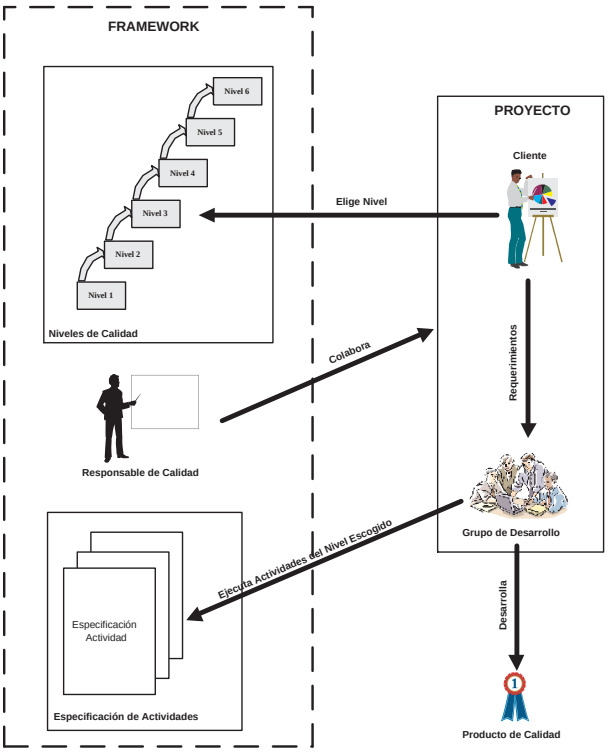


Figura 1.2: Framework y Proyecto

once proyectos de estudio se encuentra en el capítulo 4.

5. Aportes

Uno de los aportes de este trabajo es haber realizado un análisis de un problema poco analizado y para el cual no hay trabajos específicos: la mejora de la calidad en el desarrollo académico de software. También se ha estudiado el estado del arte en temas más generales y se han discutido distintas propuestas desde el punto de vista del desarrollo académico de software.

Creemos que el principal aporte de este trabajo es tener especificado y probado con éxito un Framework para el desarrollo académico de software. Este Framework respeta las características únicas de este tipo de desarrollo y, a la vez, consigue mejorar la calidad de los productos obtenidos.

Entendemos que la aplicación del Framework produjo la mejora de la calidad durante el desarrollo y también creemos que estos productos son mejores que productos de años anteriores en varias propiedades de calidad del producto de software. En particular, estos productos se desarrollan de forma más controlada, por lo que tienen menos defectos y estos son menos graves. También pensamos que los productos desarrollados en el Nivel 5 son claramente más fáciles de modificar, extender, reutilizar e integrar que prototipos de años anteriores. Además, el control que se tuvo en el desarrollo también permitió obtener productos de fácil instalación.

6. Estructura del Documento

Este documento está constituido por cuatro capítulos más. El capítulo *"Estado del Arte"* presenta el estado del arte en calidad de software y en las disciplinas Ingeniería de Requerimientos, Verificación y Validación, y Métricas y Medidas de Diseño.

En el capítulo *"Framework para la Mejora de la Calidad"* se presenta el Framework propuesto como forma de solucionar los problemas de calidad en el desarrollo del grupo CSI. En este capítulo, se tratan las actividades propuestas en las disciplinas consideradas: Especificaciones, Verificación y Validación, y Métricas y Medidas de Diseño. También se introducen los niveles de calidad del Framework y se hace un análisis del impacto en la

calidad, tanto de las actividades como de los niveles.

El estudio de campo realizado se presenta en el capítulo *"Estudio de Campo"*. En este se describen los distintos proyectos y productos realizados durante el estudio y se discuten los resultados obtenidos.

En el capítulo *"Conclusiones"* se presentan las conclusiones, los aportes y limitaciones de la propuesta, y el trabajo a futuro.

El apéndice A contiene algunos de los comentarios recibidos por los investigadores del grupo CSI respecto a la calidad que tienen los desarrollos en el momento de comenzar este trabajo.

2 Estado del Arte

Existen modelos y procesos que buscan la mejora de la calidad de los productos de software. Varios de estos modelos han sido usados en la industria con éxito. Algunos de estos se estudian y analizan desde la perspectiva del desarrollo en el grupo CSI.

Normalmente, estos modelos están pensados para grandes organizaciones, que desarrollan productos de gran porte y con gran cantidad de desarrolladores. Estas características son muy diferentes a las del desarrollo académico de software. Por este motivo, ninguno de los modelos estudiados puede ser usado directamente en el desarrollo en el CSI.

Este estado del arte busca seleccionar, de las tendencias actuales, aquellos modelos, procesos o actividades que se puedan usar o adaptar para mejorar la calidad del desarrollo académico de software.

Contenido

1. Calidad de Software

ESTA sección presenta modelos y procesos conocidos que buscan la mejora de la calidad en el desarrollo de software. Los modelos discutidos han sido usados con éxito en la industria.

Es de orden dar alguna definición de calidad de software antes de comenzar a discutir los modelos. Se brindan a continuación dos definiciones, una de la IEEE y otra de ISO.

Calidad - IEEE 610.12[IEE90]. Es el grado en que un sistema, componente o proceso cumple con los requerimientos especificados, y con las necesidades o expectativas del cliente o usuario.

Calidad - ISO 8402[ISO95]. Es la totalidad de las características de una entidad que influyen en su capacidad de satisfacer necesidades explícitas e implícitas.

Se consideran las siguientes categorías de modelos aplicables a la calidad de software y a su mejora: modelos de la calidad, modelos de madurez para la mejora del proceso, modelos de proceso y modelos de gestión de la calidad no específicos a software.

Los modelos de la calidad son, entre otros, ISO 9126[ISOb], modelo de la calidad de Bohem[BBK+78] y modelo de la calidad de McCall[MRW77]. Normalmente, estos modelos definen características o factores de la calidad de software y los relacionan con sub-características o sub-factores que son más directos de reconocer y/o medir. No consideramos relevante la discusión de estos modelos en esta tesis.

Los modelos de madurez para la mejora del proceso considerados son CMM y CMMI. Se consideran los modelos de proceso RUP, XP y PSP. Se considera a ISO 9001:2000 dentro de los modelos de gestión de la calidad no específicos a software.

El resto de esta sección se divide de la siguiente manera. En la primera subsección se presenta y discute CMM y CMMI. En la subsección 1.2 se trata PSP. Varios otros modelos son presentados brevemente en la subsección 1.3. La subsección 1.4 presenta algunas propuestas para la mejora de la calidad en un contexto académico. En la subsección 1.5 se concluye acerca de los modelos presentados y su posible aplicación en el desarrollo en el grupo CSI.

1.1. CMM y CMMI

Esta sección presenta de forma resumida las principales características de Capability Maturity Model (CMM). CMM es creado por el Software Engineering Institute (SEI) de Carnegie Mellon University.

La descripción de CMM que se presenta en esta sección se basa en los reportes técnicos del SEI *Key Practices of the Capability Maturity Model, Version 1.1* [PWG+93] y *Capability Maturity Model for Software, Version 1.1* [PCCW93].

CMM es creado para guiar a las organizaciones de software en la selección de estrategias de mejora de procesos. Esto se realiza mediante la determinación de la madurez del proceso de desarrollo y la identificación de los elementos más críticos, tanto de la calidad del software como de la mejora de procesos.

Las siguientes definiciones sirven para entender tanto a CMM como a CMMI:

Desempeño de un proceso. El desempeño en la ejecución de un proceso es una medida de los resultados reales conseguidos como efecto de su realización. El desempeño puede ser distinto cada vez que se realiza el proceso. Es deseable poder controlar y predecir el desempeño.

Capacidad de un proceso. Es el rango de resultados esperado que puede ser obtenido cuando se realiza el proceso. Permite predecir el desempeño de futuras ejecuciones.

Madurez de un proceso. Es el grado en el que un proceso está explícitamente documentado, gestionado, medido, controlado y continuamente mejorado.

La hipótesis que se maneja tanto en CMM como en CMMI es que un proceso maduro es un proceso con alta capacidad.

CMM está comprendido por cinco niveles de madurez. Cada nivel está compuesto por *Áreas Clave del Proceso* (Key Process Area). Para que una empresa alcance cierto nivel de madurez debe *cumplir* con las Áreas Clave de ese nivel y de todos los anteriores. En la Figura 2.1 se muestran los niveles, los tipos de procesos de software asociados al nivel y las Áreas Clave de ese nivel. Al subir de nivel se aumenta en la madurez del proceso. Considerando la hipótesis presentada se entiende que a mayor nivel mayor capacidad del proceso.

Cada Área Clave está presentada en el modelo de acuerdo a *Características Comunes* (Common Features), referidas a su institucionalización. Estas son: Compromiso en Realizar, Capacidad de Realizar, Actividades Realizadas, Medición y Análisis, y Verificación de Implementación. Cada Característica Común para un Área Clave se compone de *Prácticas Clave* (Key Practices). Las Prácticas Clave describen qué hay que hacer para alcanzar los objetivos de cada Área Clave, pero no describen el cómo.

Las Prácticas Clave de la característica común Actividades Realizadas describen qué debe ser implementado para establecer la capacidad del proceso. Las otras prácticas, distribuidas en las otras Características Comunes, forman la base para que la organización pueda institucionalizar las prácticas descritas en la característica común Actividades Realizadas.

En los siguientes cuadros se presentan las características y las Áreas Clave para cada uno de los niveles, y el propósito y los objetivos de cada Área Clave.

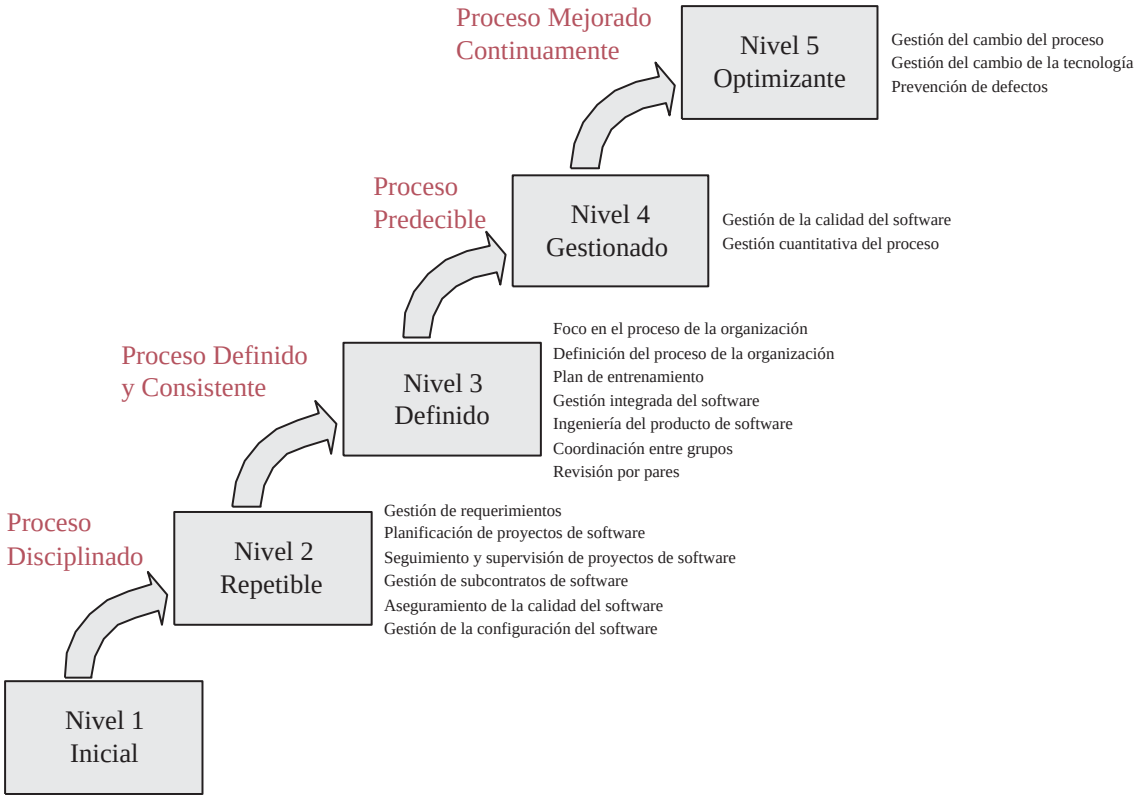


Figura 2.1: Niveles y Áreas Clave en CMM

Nivel 1 - Inicial

Características

El proceso de software se caracteriza por ser ad hoc y en ocasiones es caótico. Pocos procesos están definidos y el éxito depende de los esfuerzos individuales.

Nivel 2 - Repetible

Características

Están establecidos los procesos básicos de gestión para realizar el seguimiento de costos, la agenda de trabajo y la funcionalidad. Existe la disciplina del proceso necesaria para repetir éxitos de proyectos anteriores en aplicaciones similares.

Áreas Clave

Gestión de Requerimientos
Planificación de Proyectos de Software
Seguimiento y Supervisión de Proyectos de Software
Gestión de Subcontratos de Software
Aseguramiento de la Calidad de Software
Gestión de la Configuración de Software

Gestión de Requerimientos

Propósito

Establecer un entendimiento común entre el cliente y el proyecto de software de los requerimientos del cliente que van a ser tenidos en cuenta en el proyecto de software.

Objetivos

Objetivo 1	Tener controlados los requerimientos para que sean base para el uso de la ingeniería de software y la gerencia.
Objetivo 2	Mantener consistentes la planificación, los productos y las actividades con los requerimientos.

Planificación de Proyectos de Software

Propósito

Establecer planes razonables para realizar las actividades de ingeniería de software y para gestionar el proyecto.

Objetivos

Objetivo 1	Documentar las estimaciones del trabajo para que sean usadas en la planificación y en el seguimiento del proyecto de software.
Objetivo 2	Planificar y documentar las actividades y los compromisos del proyecto.
Objetivo 3	Los grupos y los individuos deben estar de acuerdo con sus compromisos en el proyecto.

Seguimiento y Supervisión de Proyectos de Software

Propósito

Proveer adecuada visibilidad del progreso real del proyecto para que la gerencia pueda tomar acciones efectivas cuando la ejecución del proyecto se desvía significativamente de los planes.

Objetivos

Objetivo 1	Comparar los resultados reales y la ejecución del proyecto contra los planes.
Objetivo 2	Tomar (y gestionar) acciones correctivas cuando los resultados reales y la ejecución del proyecto se desvían significativamente de los planes.
Objetivo 3	Los grupos y los individuos deben estar de acuerdo con los cambios en los compromisos que los afectan.

Gestión de Subcontratos de Software

Propósito

Seleccionar subcontratistas de software calificados y gestionarlos eficazmente.

Objetivos

Objetivo 1	Seleccionar subcontratistas de software calificados.
Objetivo 2	Acordar los compromisos de cada uno con el subcontratista.
Objetivo 3	Mantener comunicación constante con el subcontratista.
Objetivo 4	Comparar los resultados reales y el rendimiento del subcontratista contra sus compromisos.

Aseguramiento de la Calidad de Software	
	Propósito
	Proporcionar a la gerencia una apropiada visibilidad tanto del proceso que se está usando por el proyecto de software como de los productos que se están construyendo.
	Objetivos
Objetivo 1	Planificar las actividades de aseguramiento de la calidad de software.
Objetivo 2	Verificar objetivamente la conformidad de los productos de software y las actividades a los estándares, procedimientos y requerimientos aplicables.
Objetivo 3	Informar de los resultados y las actividades del aseguramiento de la calidad de software a los grupos e individuos que se vean afectados.
Objetivo 4	Proporcionar la información de incumplimientos que no pueden ser resueltos dentro del proyecto de software para que intervenga la alta gerencia.

Gestión de la Configuración de Software	
	Propósito
	Establecer y mantener la integridad de los productos del proyecto de software a través del ciclo de vida del proyecto.
	Objetivos
Objetivo 1	Planificar las actividades de gestión de la configuración.
Objetivo 2	Identificar, controlar y dejar disponibles los productos de software que estarán bajo configuración.
Objetivo 3	Controlar los cambios de los productos de software que estén bajo configuración.
Objetivo 4	Informar del estado y contenido de la linea base a los grupos e individuos que se vean afectados.

Nivel 3 - Definido

Características

El proceso de software para las actividades de gestión y de ingeniería está documentado, estandarizado e integrado en un proceso de software estándar de la organización. Para desarrollar y mantener software todos los proyectos usan y aprueban versiones adaptadas del proceso de software estándar de la organización.

Áreas Clave

Foco en el Proceso de la Organización
Definición del Proceso de la Organización
Programa de Entrenamiento
Gestión Integrada del Software
Ingeniería de Producto de Software
Coordinación entre Grupos
Revisión por Pares

Foco en el Proceso de la Organización

Propósito

Establecer una responsabilidad a nivel de la organización para las actividades del proceso de software que mejoran la capacidad del proceso.

Objetivos

Objetivo 1	Coordinar en toda la organización el desarrollo y la mejora del proceso de software.
Objetivo 2	Identificar las fortalezas y debilidades de los procesos utilizados respecto a un proceso estándar.
Objetivo 3	Planificar las actividades, a nivel de la organización, de desarrollo y mejora del proceso.

Definición del Proceso de la Organización	
	Propósito
	Desarrollar y mantener un conjunto de procesos de software utilizables que mejoran el rendimiento de los procesos a través de los proyectos y que provean una base para los beneficios acumulativos y de largo plazo para la organización.
	Objetivos
Objetivo 1	Desarrollar y mantener un proceso de software estándar para la organización.
Objetivo 2	Recolectar, revisar y dejar disponible la información relacionada con el uso en los distintos proyectos del proceso de software estándar de la organización.

Programa de Entrenamiento	
	Propósito
	Desarrollar las capacidades y el conocimiento de los individuos para que puedan desempeñar sus roles de forma efectiva y eficiente.
	Objetivos
Objetivo 1	Planificar las actividades de entrenamiento.
Objetivo 2	Proveer entrenamiento para desarrollar las capacidades y conocimiento necesarios para desempeñar roles en la gestión de software y roles técnicos.
Objetivo 3	Brindar a los individuos del grupo de ingeniería de software y los grupos de software relacionados el entrenamiento necesario para que puedan desempeñar sus roles.

Gestión Integrada del Software	
	Propósito
	Integrar las actividades de la ingeniería y la gestión de software en un proceso de software definido y coherente, este es una adaptación del conjunto de procesos estándar de software de la organización. Estos procesos son los descritos en Definición del Proceso de la Organización.
	Objetivos
Objetivo 1	Tener un proceso de software definido para el proyecto que es una versión adaptada del proceso estándar de la organización.
Objetivo 2	Planificar y gestionar el proyecto de acuerdo al proceso definido para el mismo.

Ingeniería de Producto de Software	
	Propósito
	Ejecutar un proceso de ingeniería bien definido que integra todas las actividades de ingeniería de software para producir de forma eficiente y efectiva productos de software correctos.
	Objetivos
Objetivo 1	Definir, integrar y ejecutar las actividades de ingeniería de software para producir el software.
Objetivo 2	Mantener la coherencia entre los productos de software.

Coordinación entre Grupos	
	Propósito
	Establecer los medios para que el grupo de ingeniería de software participe activamente con los otros grupos de ingeniería, por lo que el proyecto será más capaz de satisfacer las necesidades del cliente de forma efectiva y eficiente.
	Objetivos
Objetivo 1	Aprobar los requerimientos del cliente por todos los grupos afectados.
Objetivo 2	Acordar los compromisos entre los grupos de ingeniería por los grupos afectados.
Objetivo 3	Los grupos de ingeniería deben identificar, realizar el seguimiento y resolver los problemas entre los grupos.

Revisión por Pares

Propósito

Remover defectos de los productos de software de forma temprana y eficiente.

Objetivos

Objetivo 1	Planificar las revisiones por pares.
Objetivo 2	Identificar y remover los defectos en los productos de software.

Nivel 4 - Gestionado

Características

Se recolectan mediciones detalladas de la calidad del proceso y del producto. El proceso de software y los productos son entendidos y controlados de forma cuantitativa.

Áreas Clave

Gestión Cuantitativa del Proceso
Gestión de la Calidad del Software

Gestión Cuantitativa del Proceso

Propósito

Controlar cuantitativamente el desempeño del proceso en el proyecto de software. El desempeño de un proceso representa los resultados reales logrados al seguir el mismo.

Objetivos

Objetivo 1	Planificar las actividades de gestión cuantitativa del proceso.
Objetivo 2	Controlar de forma cuantitativa el desempeño del proceso de software definido para el proyecto.
Objetivo 3	Conocer, en términos cuantitativos, la capacidad del proceso de software estándar de la organización.

Gestión de la Calidad de Software	
	Propósito
	Desarrollar un conocimiento cuantitativo de la calidad de los productos de software y conseguir alcanzar determinados objetivos de calidad.
	Objetivos
Objetivo 1	Planificar las actividades de gestión de la calidad de software.
Objetivo 2	Definir y priorizar objetivos medibles para la calidad de los productos de software.
Objetivo 3	Cuantificar y gestionar el progreso real hacia el logro de los objetivos de calidad de los productos de software.

Nivel 5 - Optimizante	
	Características
	Es posible la mejora continua del proceso debido a la retroalimentación cuantitativa del proceso y a las pruebas piloto de ideas y tecnología innovadora.
	Áreas Clave
	Prevención de Defectos Gestión del Cambio de la Tecnología Gestión del Cambio del Proceso

Prevención de Defectos	
	Propósito
	Identificar las causas de los defectos y prevenir que estas ocurran.
	Objetivos
Objetivo 1	Planificar las actividades de prevención de defectos.
Objetivo 2	Buscar e identificar las causas comunes de defectos.
Objetivo 3	Priorizar y eliminar sistemáticamente las causas comunes de defectos.

Gestión del Cambio de la Tecnología	
	Propósito
	Identificar nuevas tecnologías (por ejemplo, herramientas, métodos y procesos) y llevarlas a la organización de una manera ordenada.
	Objetivos
Objetivo 1	Planificar la incorporación de cambios tecnológicos.
Objetivo 2	Evaluar las nuevas tecnologías para determinar el efecto sobre la calidad y la productividad.
Objetivo 3	Transferir a la práctica normal de la organización las nuevas tecnologías que sean apropiadas.

Gestión del Cambio del Proceso	
	Propósito
	Mejorar de forma continua el proceso de software usado en la organización con la intención de mejorar la calidad, aumentar la productividad y disminuir el tiempo de desarrollo.
	Objetivos
Objetivo 1	Planificar la mejora continua del proceso.
Objetivo 2	Lograr que toda la organización participe de las actividades de mejora del proceso de software.
Objetivo 3	Mejorar continuamente el proceso de software estándar de la organización y los procesos de software definidos en los proyectos.

Discusión

CMM está enfocado a organizaciones que realizan proyectos de gran escala en un contexto de contrato con el gobierno, y más específicamente con el Department of Defense (DoD) de los Estados Unidos de América.

Considerando lo mencionado en el párrafo anterior se entiende que CMM no se adapta a las necesidades de esta tesis. Por un lado el mismo está enfocado a las organizaciones, que en este caso podrían ser el CSI o el InCo. En este estudio se quiere tener el foco en los grupos de desarrollo y no en la organización. Por otro lado CMM está pensado para proyectos de gran escala y estos no son los proyectos desarrollados dentro del CSI.

Sin embargo, en CMM se sugiere realizar una interpretación razonable de las prácticas para cualquier contexto en que se quiera aplicar el modelo. Nosotros presentaremos una breve discusión a partir de los propósitos de cada Área Clave.

El propósito que se persigue en *Gestión de Requerimientos* es contemplado en el Framework propuesto. La sección 2 de este capítulo trata la disciplina Ingeniería de Requerimientos por lo que no se discute este punto aquí.

En *Planificación de Proyectos de Software* se pide establecer planes razonables para las actividades de ingeniería de software y de gestión de proyecto. Entendemos que por las características de los proyectos, y por experiencias pasadas, la gestión de los proyectos no ha sido un problema. En particular los proyectos no son casi gestionados. En el Framework se considera una planificación, no muy completa, de la Verificación. Sin embargo, no se planifican otras actividades de ingeniería por considerarlo innecesario dadas las características de los proyectos.

En *Seguimiento y Supervisión de Proyectos de Software* lo importante es proveer adecuada visibilidad del progreso real del proyecto para poder tomar acciones correctivas si es necesario. Al no contarse con planes, el progreso real no se puede comparar contra estos por lo que esta Área Clave pierde gran parte de sentido. Sin embargo, el seguimiento interesante que se da en los proyectos del CSI es con el cliente; con el investigador. Por esto el Framework propuesto sugiere entregas de prototipos, de forma continua, al cliente.

El Área Clave *Gestión de Subcontratos de Software* no aplica en este contexto debido a que directamente no existen los subcontratistas.

Para *Aseguramiento de la Calidad de Software* lo más importante es lograr tener una apropiada visibilidad del proceso de software y de los productos que se están construyendo respecto a la calidad de los mismos. Normalmente este trabajo lo hace un equipo independiente del equipo de desarrollo. En los proyectos del CSI no se tiene un equipo independiente de SQA. En el Framework se proponen actividades de aseguramiento de la calidad y de control de la calidad. Las mismas son, en su mayoría, actividades de verificación. Una discusión de la verificación y validación de software se trata en la sección 3 de este capítulo.

La *Gestión de la Configuración de Software* (SCM) es razonable para cualquier proyecto. De todas maneras no se dan sugerencias sobre este aspecto en el Framework presentado. Se entiende que esta actividad es realizada informalmente por los grupos de desarrollo y que no son necesarios ni procedimientos ni herramientas para SCM.

En *Foco en el Proceso de la Organización* se busca establecer una responsabilidad a nivel de toda la organización para mejorar la capacidad del proceso de software. Los aspectos que refieren a la organización no se consideran en el Framework debido a que el foco está puesto en los grupos de desarrollo. Una breve discusión sobre la organización se da en el capítulo 5 en la sección 3.

Definición del Proceso de la Organización no es considerado en el Framework por motivos similares a los mencionados en el Área Clave anterior.

En *Programa de Entrenamiento* se busca tener un entrenamiento para que los individuos puedan desarrollar sus roles de forma efectiva y eficiente. En el Framework se tiene un rol de Responsable de Calidad, este rol no es llevado adelante ni por el grupo de desarrollo ni por el cliente. Esta persona entrena tanto al cliente como al grupo de desarrollo en el Framework, permitiendo que puedan desarrollar sus roles. Esto se trata en el capítulo 3.

El Área Clave *Gestión Integrada del Software* busca integrar las actividades de ingeniería y las de gestión en un proceso que es una adaptación del proceso de la organización. Esto, por motivos que ya fueron mencionados acerca del proceso de la organización queda fuera del Framework.

En *Ingeniería de Productos de Software* se busca ejecutar un proceso de ingeniería bien definido para lograr producir productos correctos de forma efectiva y eficiente. En el Framework, las actividades son en su totalidad actividades de ingeniería. Si bien no se define un proceso se insta a los grupos de desarrollo a que definan el suyo propio. El propósito de esta Área Clave se ve sumamente relacionado con el objetivo planteado por el grupo CSI. Varias de las 10 Prácticas Clave de la característica común Actividades Realizadas de esta Área Clave son consideradas en el Framework.

El Área Clave *Coordinación entre Grupos* considera los medios para que el grupo de ingeniería de software interactúe con otros grupos de ingeniería. Esta área no se considera en el Framework debido a que existe un único grupo de ingeniería y este es el de desarrollo.

Remover defectos de forma temprana es el propósito del Área Clave *Revisión por Pares*. En este caso no se puede hacer una discusión sólo basándose en el propósito. Es intención del Framework remover defectos de forma temprana. De todas formas no se define ninguna revisión por pares por lo que esta Área Clave no es contemplada en el Framework.

Las Áreas Clave restantes tratan del control cuantitativo del proceso y los productos (*Gestión Cuantitativa del Proceso* y *Gestión de la Calidad de Software*), de la prevención de defectos (*Prevención de Defectos*) y de la gestión del cambio de la tecnología y del proceso buscando la mejora del mismo (*Gestión del Cambio de la Tecnología* y *Gestión del Cambio del Proceso*). Ninguna de estas es considerada en el Framework. Se entiende que para una primera versión del Framework y sin tener en claro la definición de la organización y sus responsabilidades (ver sección 3 en el capítulo 5), es excesivo considerar estas Áreas Clave.

Si bien se ha mencionado que algunas Áreas Clave son consideradas por el Framework hay que entender que lo que se hizo fue una discusión basada en los propósitos de las mismas. Se ha realizado entonces una consideración muy *liviana* de las Áreas. Sin embargo, se ha intentado tener una interpretación de las Áreas que se adapte a las necesidades de desarrollo de software dentro del grupo CSI.

Una consideración más profunda es a nivel de las Prácticas Clave. Estas dan una idea más acabada de lo que se busca en cada Área Clave y cómo *implementar* su propósito y objetivos. El Framework propuesto está muy lejos de la implementación de todas las Prácticas Clave de las Áreas que se han tenido en cuenta. Las Prácticas, como ya se mencionó, responden a organizaciones que realizan proyectos de gran escala en un contexto de contrato con el gobierno. El desarrollo en el CSI es muy diferente a este caso. Entendemos que la adaptación de las Prácticas, para este caso en particular, no es una buena idea. Por esto, únicamente se considera el estudio de los propósitos de cada Área y se busca satisfacer aquellos que parecen razonables para el desarrollo académico de software.

Si bien Capability Maturity Model Integration[1] (CMMI) tiene diferencias con CMM estas no influyen en nuestro trabajo. Se deja de lado una presentación y una discusión de CMMI por entender que no aporta a este trabajo ya que sería repetir mucho de lo planteado sobre CMM.

1.2. PSP

Esta sección presenta en forma resumida las principales características de Personal Software Process (PSP). PSP es creado, al igual que CMM, por el SEI.

La descripción de PSP aquí contenida es extraída del reporte técnico del SEI *The Personal Software Process* [Hum00] de Humphrey. Una presentación completa de PSP se puede encontrar en [Hum95].[2]

Personal Software Process (PSP) se crea para extender el proceso de mejora, planteado en CMM, a las personas que hacen el trabajo de desarrollo de software; los ingenieros. El principio de PSP es que para producir software de calidad todo ingeniero que trabaja en el desarrollo debe realizar un trabajo de alta calidad. Para esto los desarrolladores se deben sentir responsables por la calidad de sus productos.

PSP es un proceso para un individuo. Este abarca desde la asignación de un problema (requerimientos) hasta la prueba unitaria del programa.[3] En la Figura 2.2 se muestra el proceso y sus fases.

El proceso comienza con los requerimientos y se ejecutan distintas fases donde la primera es la planificación. PSP provee scripts (instrucciones) que sirven de guía para las fases. El tiempo consumido en cada fase y la cantidad y tipos de defectos encontrados se registran en los logs. Durante la fase de postmortem se resumen los datos de los logs, se mide el tamaño del programa que se desarrolló y se entran estos datos en el Sumario del Plan.

La fase de planificación tiene como entrada los requerimientos. En esta fase se realiza un diseño conceptual, se estima el tamaño del producto, se estiman los recursos y se arma la agenda de desarrollo. Luego de desarrollado el producto se guardan, en la base de datos de históricos, el tamaño real del producto y el tiempo usado en el desarrollo. Estos sirven para estimar el tamaño del producto y los recursos necesarios en futuros desarrollos. El proceso de planificación se muestra en la Figura 2.3.

A partir de los requerimientos se produce un diseño conceptual. Este es una primera aproximación a la solución del problema. Después, durante la fase de diseño el ingeniero examina alternativas de diseño y produce un diseño completo del producto. Este sirve para realizar las estimaciones que se necesitan para armar el cronograma.

En PSP se realiza la estimación del tamaño y los recursos usando el método PROxy Based Estimating (PROBE). Primero se determinan los objetos que se necesitan para

[1] CMMI: http://www.sei.cmu.edu/cmmi/cmmi.html/

[2] Este libro no fue leído para realizar esta tesis.

[3] Aquí por programa se entiende: pequeño programa, módulo, clase o un conjunto pequeño de clases.

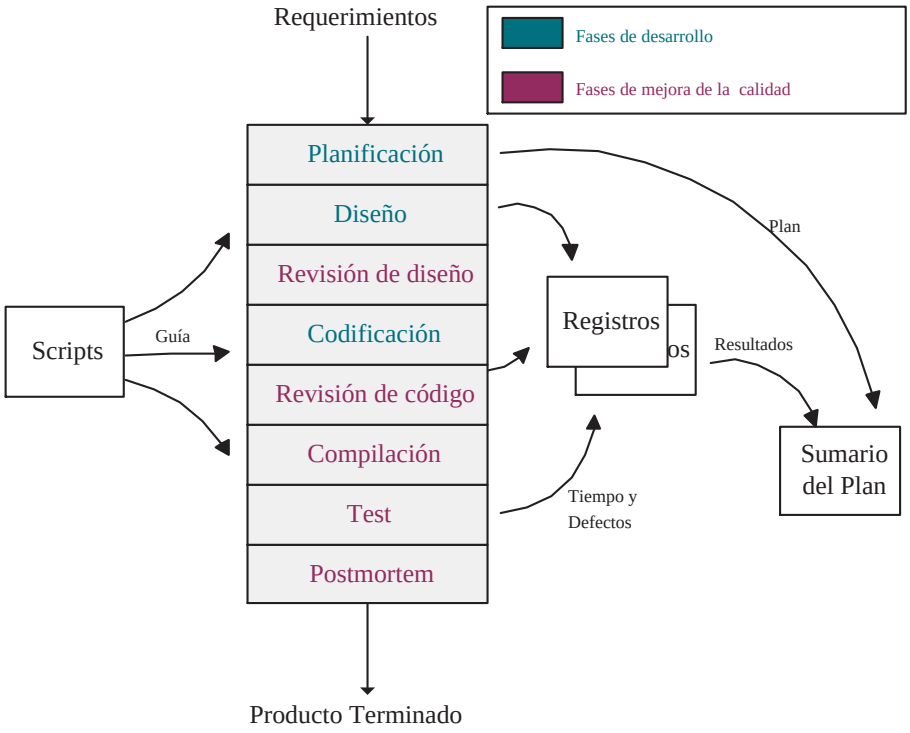


Figura 2.2: Flujo del Proceso PSP

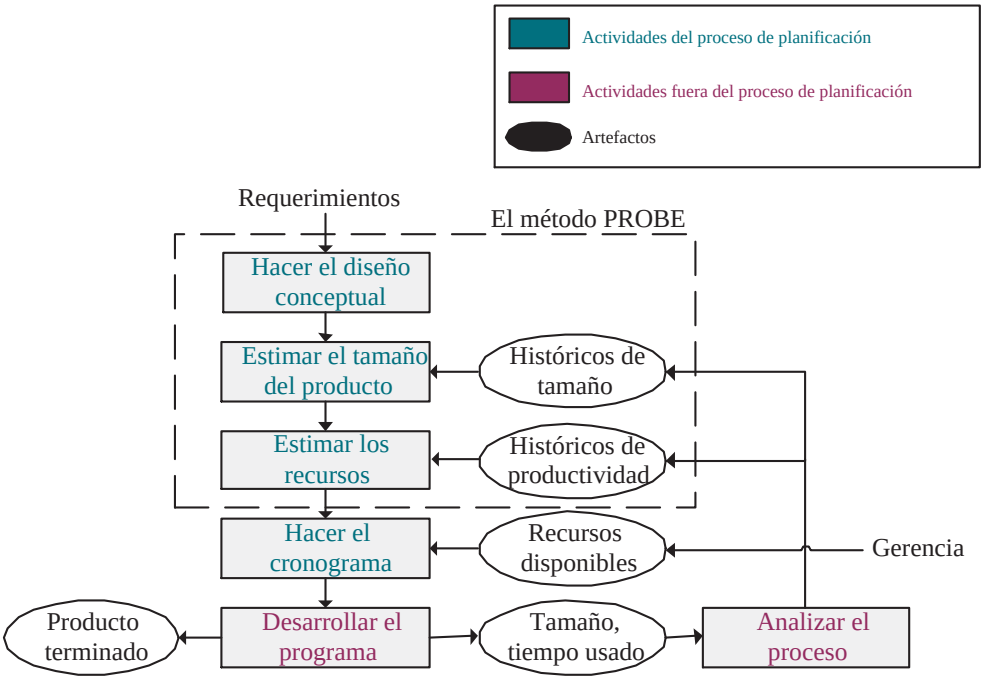


Figura 2.3: Proceso de Planificación del Proyecto

construir el producto descrito en el diseño conceptual. Luego, basándose en los datos históricos de objetos similares y usando regresión lineal se estima el tamaño del producto final. Con esta estimación, los datos históricos de productividad y nuevamente usando regresión lineal se calcula el tiempo que se necesita en cada fase del proceso. Al final de este proceso se tiene una estimación del tamaño total del producto, del tiempo total de desarrollo y del tiempo requerido en cada fase.

A partir de los estimativos del tiempo requerido para cada fase se realiza el cronograma de desarrollo. También es necesario conocer cuánto tiempo se puede dedicar al proyecto por día o por semana.

En PSP existe una continua recolección de datos por parte del desarrollador. Estos datos se guardan para realizar mejores estimaciones y para monitorizar el trabajo.

Se debe guardar, en el registro de tiempos, el tiempo que se usa en cada fase del proceso. Si hay interrupciones estas se deben registrar. Las mismas pueden ser de cualquier tipo, por ejemplo, llamadas por teléfono, almuerzo y cortes breves.

También se registran las mediciones del tamaño de los productos que se desarrollan. Estas y los registros de tiempos sirven para realizar las estimaciones. PSP usa las líneas de código como medida del tamaño de los productos. De todas maneras, se puede usar cualquier medida que pueda ser obtenida de forma automática, y que tenga una correlación razonable entre el tiempo de desarrollo y el tamaño del producto.

El foco principal de calidad en PSP son los defectos. Para poder administrarlos se deben recolectar los defectos que se inyectan, las fases en las cuales son inyectados, las fases en las cuales son encontrados y corregidos, y cuánto tiempo lleva corregirlos. Estos datos se toman en las fases de revisión, compilación y testing.

PSP tiene un conjunto de medidas de calidad que ayuda a los ingenieros a examinar la calidad de sus productos. Mediante los resultados de estas medidas se pueden tabular los tiempos a usar en cada fase del proceso, y los defectos que se deben encontrar y que se pueden inyectar en cada una. Algunos de estos datos son generales pero todos deben ser ajustados al individuo particular que usa PSP.

PSP busca remover los defectos de forma temprana y prevenir la introducción de los mismos como forma de lograr productos de calidad.

Para remover tempranamente los defectos se proponen las fases revisión de diseño y revisión de código. Conociendo los defectos en los que incurre la persona esta puede tener *checklists* que la ayuden a revisar el diseño y el código. De esta forma se llega a la fase de test con menos defectos.

Para prevenir la inyección de defectos PSP propone dos cosas. Primero, que la persona conozca los defectos en los que normalmente incurre. Al conocer estos defectos se puede evitar cometer los mismos errores. Segundo, realizar el diseño de forma completa; se recomienda tener un método y una notación. Esto produce mejores diseños y menos defectos en el mismo. Además, debido a que el diseño es completo, se obtiene un menor tiempo en la etapa de codificación.

También esto genera menos defectos ya que durante el diseño se introducen menos defectos por hora que en la codificación.

El SEI tiene un curso para introducir PSP en la universidad y un curso para la industria. Tanto el curso universitario como el de la industria hacen pasar al ingeniero por siete versiones de PSP. Estas se muestran en la Figura 2.4. Cada versión agrega nuevos elementos y la última versión, PSP 3, es la completa. El curso universitario dura un semestre. El curso para la industria tiene una carga de 150 horas. Estas normalmente se distribuyen en 14 días de trabajo.

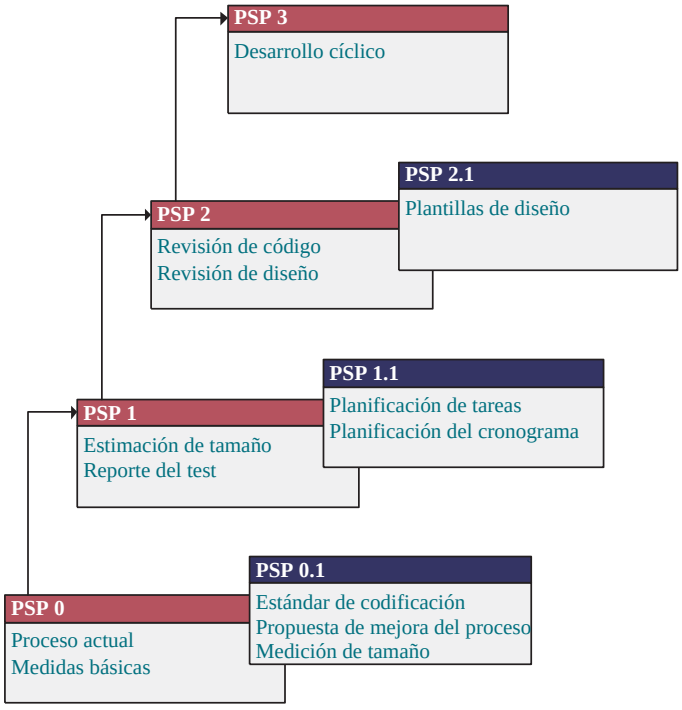


Figura 2.4: Evolución del Proceso PSP

Como resultado se tiene que durante el curso de PSP los ingenieros mejoran las estimaciones, disminuyen los defectos por líneas de código y no disminuyen la productividad. Sin embargo, sí existe una disminución en la productividad mientras los ingenieros aprenden PSP y están en las versiones 0.1, 1 y 1.1.

Discusión

PSP ha dado buenos resultados en la academia y en la industria [FHK+97]. Sin embargo, los resultados se dan luego de aprendido PSP, tanto sea con el curso universitario como con el curso para la industria.

PSP presenta una dificultad, que entendemos como insalvable en este momento, para ser aplicado dentro del CSI. El proceso ha demostrado dar buenos resultados en lo que refiere a calidad de software, sin embargo, esto se da con una persona ya formada en PSP. La Facultad de Ingeniería, en este momento, no cuenta con un curso en PSP por lo que los desarrolladores comienzan el proyecto sin esta formación. Por diferentes motivos es imposible formar a los desarrolladores durante la ejecución del proyecto. Además, realizar

esto supone una perdida en productividad, al menos mientras los desarrolladores transitan los niveles 0.1, 1 y 1.1. En suma, se debe descartar tener desarrolladores usando PSP en los proyectos del CSI.

De todas formas parece interesante discutir acerca de las generalidades de PSP que llevan a obtener productos de calidad. Estas son: planificar el trabajo, diseñar detalladamente, revisar el diseño, revisar el código y realizar pruebas unitarias.

Como ya se mencionó en la discusión de CMM, no parece que vaya a mejorar la calidad de los productos del CSI si se realiza una planificación de las tareas. Entendemos entonces, que la planificación del trabajo individual no es necesaria en el contexto del CSI.

Las tareas de diseño detallado son sugeridas en el Framework y se pide que se realicen mediciones del mismo. Las mediciones de diseño se tratan en la sección 4 de este capítulo. No se propone en el Framework realizar, cada vez que se diseña, una revisión de lo diseñado. Sin embargo, se establece que cuando las mediciones tomadas del diseño caen fuera del rango establecido sea realizada una revisión del mismo. A partir de la revisión se establece si es necesario o no mejorar el diseño.

Se entiende que las revisiones de código son una actividad fundamental en la búsqueda de la calidad de software. Lamentablemente, los desarrolladores que se tienen que considerar en este trabajo no están acostumbrados a realizar esta actividad. Se cree, que por más que esta actividad figure en el Framework la misma no será realizada. Por este motivo y también por intentar mantener un Framework sencillo esta actividad no forma parte del mismo.

Las pruebas unitarias son una actividad dentro del Framework propuesto. Se considera que esta puede mejorar sustancialmente la calidad de los productos entregados y que los desarrolladores están dispuestos a realizarla.

En resumen, PSP no puede ser usado como tal en los proyectos del CSI. Sin embargo, las actividades que consideramos más importantes y aplicables a los proyectos del CSI están adaptadas y son parte del Framework.

1.3. Otros Modelos

En esta sección se mencionan brevemente tres modelos estudiados para la construcción del Framework.

ISO 9001:2000 [ISOa] presenta un modelo de un sistema de gestión de la calidad. El énfasis de este modelo está puesto en la satisfacción del cliente y la mejora continua. Al igual que CMM está pensado para organizaciones y no para pequeños grupos de desarrollo.

Rational Unified Process [IBM] (RUP) es un proceso *pesado* de desarrollo de software. Este está compuesto de actividades en distintas disciplinas de la ingeniería de software. RUP está pensado para proyectos de gran porte.

Extreme Programming [BA04] (XP) es un proceso *liviano* de desarrollo de software. XP tiene particularidades que lo diferencian de otros procesos de desarrollo, entre ellas

se encuentran la programación por pares y que el cliente tiene que estar en el lugar de desarrollo.

Discusión

ISO 9001:2000 no fue considerado para esta tesis ya que creemos que las actividades que se podrían adaptar o aprovechar del mismo están incluidas en CMM y estas ya son tenidas en cuenta.

Para nuestro trabajo se estudian las distintas actividades de las disciplinas de Análisis, Diseño y Verificación de RUP. Estas no son discutidas aquí ya que las mismas se presentan en el estado del arte de la Ingeniería de Requerimientos, Métricas y Medidas de Diseño, y Verificación y Validación. Algunas de estas actividades son adaptadas para su uso en el Framework.

Con XP se realiza el mismo tipo de estudio que el que se lleva a cabo con RUP. Muchas de las prácticas de este proceso no se adaptan a las características de los proyectos del CSI. Sin embargo, algunas se pueden considerar, por ejemplo, la programación por pares. De todas maneras, estas prácticas no se incluyen en el Framework sino que se dejan libradas a cada grupo de desarrollo.

1.4. Calidad en un Contexto Académico

Distintos trabajos se han propuesto para mejorar la calidad de los productos de desarrollo en un ambiente académico. Sin embargo, todos los estudios encontrados responden a enseñar la ingeniería de software y los procesos de desarrollo buscando productos de cierta calidad como resultado. En definitiva, el foco está en dos cosas: enseñanza de la ingeniería de software y calidad de productos. Es importante hacer esta distinción ya que nuestro trabajo se dedica solamente a la mejora de la calidad en el desarrollo académico de software.

En la Universidad de Sydney, en el curso *Proyecto de desarrollo de software*, grupos de entre 5 o 6 personas desarrollan un producto de software. El proceso que deben seguir los estudiantes es similar a XP. Los clientes del curso son externos a la universidad. Se usan varias herramientas de apoyo al desarrollo de software durante el curso [Ude06].

Hay otros cursos de características similares en Alemania [BFD04, SBD98].

En nuestra Universidad se dicta el curso *Proyecto de Ingeniería de Software*, que aplica un proceso similar al RUP para introducir de manera práctica la ingeniería de software [Tri04]. En este curso los grupos de desarrollo son de entre 10 y 12 integrantes. Existen numerosos cursos en universidades de distintos países con las mismas características.

Discusión

Los cursos mencionados normalmente tienen más personas desarrollando que los grupos de desarrollo de CSI. Además, el objetivo principal de estos cursos es que los estu-

diantes aprendan distintas prácticas de la ingeniería de software.

Entendemos que estos cursos son positivos para mejorar las capacidades de las personas que desarrollan. En particular, el curso dictado en nuestra facultad aporta de forma indirecta en la calidad del desarrollo dentro del grupo CSI. Creemos que algunas de las actividades planteadas en el Framework serían difíciles de entender, y sobre todo de aplicar, si los desarrolladores no han tomado un curso de este estilo o participado en desarrollos en los cuales se desarrollan actividades de calidad.

1.5. Discusión General de la Calidad de Software

Se presentaron en esta sección distintos modelos que buscan, entre otras cosas, mejorar la calidad del software. Estos modelos están pensados para grandes organizaciones, que desarrollan productos de gran porte y con gran cantidad de desarrolladores. Estas características son muy diferentes a las del desarrollo dentro del grupo CSI. Por este motivo, los modelos no son aplicables de forma completa.

Los modelos que tienen las características mencionadas y que se presentaron en esta sección son: CMM, CMMI, ISO 9001:2000, RUP Y XP. Estos modelos se recorren buscando partes que puedan ser ajustadas para su uso en el desarrollo en el CSI. De esta manera se consiguen alcanzar varios de los objetivos de estos modelos.

Por otro lado, el modelo PSP presenta características distintas a los otros modelos. PSP ha dado resultados en la academia, sin embargo, los resultados se dan luego de aprendido PSP.

Lamentablemente, al día de hoy, no hay cursos de PSP en la Facultad de Ingeniería. Este hecho provoca que el mismo no se pueda aplicar en los desarrollos del CSI.

Sin embargo, se puede realizar un estudio similar al realizado con los otros modelos. Actividades como el diseño detallado, la revisión del diseño y la realización de pruebas unitarias, que son actividades de PSP, también están en el Framework.

En resumen, ninguno de los modelos estudiados puede ser usado directamente en el desarrollo en el CSI. Debido a esto se construye el Framework. Sin embargo, en esta sección se vieron adaptaciones de los modelos que forman parte del Framework.

2. Ingeniería de Requerimientos

En esta sección se presenta el estado del arte en el área Ingeniería de Requerimientos (IR). Esta sección está dividida en distintos tópicos que creemos de importancia mencionar en un estudio del estado del arte de la IR.

La IR es una área muy amplia dentro de la ingeniería de software. La idea de este estado del arte es mencionar la gran mayoría de los tópicos más importantes de la IR para la industria. Luego, para cada uno de los tópicos se realiza una discusión donde evaluamos la importancia relativa del tópico presentado para el desarrollo académico de software. Por último, cuando creemos que corresponde, se menciona qué se incluye en el Framework para mejora de la calidad. Comenzamos, como primer tópico, con distintas definiciones de la IR.

2.1. Definición de la Ingeniería de Requerimientos

En [Hof00] se dice que la IR es tanto el proceso de especificar los requerimientos mediante el estudio de las necesidades de los *stakeholders* como el proceso de analizar sistemáticamente y refinar esas especificaciones.

La IR debe plasmar los objetivos (*goals*) que explican por qué se necesita el sistema, las funcionalidades que el software debe llevar a cabo para lograr esos objetivos y las restricciones que restringen la forma en la cual el software va a ser diseñado e implementado. Estos objetivos, funciones y restricciones tienen que estar mapeados a especificaciones precisas del comportamiento del software; su evolución a través del tiempo y a través de familias de software debe ser considerado también [Zav97].

En términos generales, la Ingeniería de Requerimientos de Sistemas de Software es el proceso de descubrir ese propósito *[el propósito para el cual es previsto el sistema]*, mediante la identificación de *stakeholders* y sus necesidades, y documentar estas en una forma que es adecuada para el análisis, la comunicación y posterior implementación [NE00].

2.2. Definición e Importancia de Stakeholders

Existen varias definiciones de *stakeholder* en la literatura, a continuación se brindan algunas de ellas. Un relevamiento de definiciones de *stakeholder* se presenta en [SFG99].

Definimos stakeholder como esos participantes *[en el proceso de desarrollo]* junto con otros individuos, grupos u organizaciones cuyas acciones pueden influir o ser influenciadas por el desarrollo y uso del sistema tanto sea directa como indirectamente [PW97].

Un *stakeholder* en una organización es (por definición) cualquier grupo o individuo que puede afectar o es afectado por la realización de los objetivos de la organización [Fre84].

La retroalimentación brindada por los *stakeholders* juega un rol decisivo desde el comienzo hasta el final de un proyecto exitoso de IR. Los equipos más exitosos siempre incluyen clientes y usuarios en el proceso de IR y mantienen una buena relación con los *stakeholders*. Tienen una constante colaboración con los *stakeholders* para asegurar que los requerimientos son interpretados correctamente, para tratar con los requerimientos cambiantes, y para evitar que se rompa la comunicación [HL01].

Discusión

A nuestro entender, y al menos considerando la realidad que se da en la Facultad de Ingeniería de la UdelaR, no vemos como necesaria la búsqueda exhaustiva de los stakeholders en el desarrollo académico de software. Normalmente basta con conocer al usuario del sistema a construir, donde a veces es uno mismo (ejemplo: tesis de maestría o doctorado donde hay una parte de desarrollo de software).

Es común en esta clase de desarrollo que los usuarios sean pocos y que se sepa apenas comienza el proyecto quienes son estos.

Debido a las razones presentadas es que no se tiene en el Framework para la Mejora de la Calidad ninguna técnica para identificar a los *stakeholders* ni se presentan técnicas para solucionar los conflictos en los objetivos de distintos *stakeholders*.

En [HL01], se menciona la importancia de incluir clientes y usuarios durante todo el proceso de IR y mantener una buena relación con ellos durante todo el proceso. Esto fue considerado y se sugirió a cada grupo de desarrollo que utilizaran, en lo posible, un desarrollo iterativo e incremental y que siempre mostraran resultados a los usuarios.

2.3. Documento de Especificación de Requerimientos

Acerca del documento de especificación de requerimientos presentamos las siguientes citas: la especificación, el resultado primario de la IR, es una concisa declaración de los requerimientos que el software debe satisfacer [IEE98]. Idealmente, una especificación permite a los *stakeholders* aprender rápidamente sobre el sistema y a los desarrolladores entender exactamente qué quieren los *stakeholders* [HL01].

Discusión

El documento de especificación de requerimientos es básico y fundamental en el Framework que proponemos.

Debido al porte de los proyectos no consideramos que sea necesario tener establecidas plantillas estándar para el documento de requerimientos.

2.4. Importancia de la Ingeniería de Requerimientos

La importancia de la Ingeniería de Requerimientos es presentada por varios autores, aquí mostramos algunas de estas opiniones. Algunas de las citas refieren a los motivos de los fracasos en la industria de software. Siendo tantos fracasos debidos a la pobre IR, se entienden los mismos como una muestra de la importancia del área dentro del desarrollo de software.

Entre los beneficios de tener *buenos requerimientos* se encuentran los siguientes [DT97]:

- Acuerdo entre los desarrolladores, clientes y usuarios en el trabajo a ser realizado y en los criterios de aceptación para el sistema a ser entregado.
- Base para la estimación de recursos (costo, cantidad y habilidades del personal, equipamiento y tiempo).
- Mejora la usabilidad, mantenibilidad y otros atributos de calidad del sistema.
- Logro de objetivos con mínimos recursos (menor re-trabajo, menos omisiones y malentendidos).

En el reporte CHAOS: A Recipe for Success [Gro99] se plantea que subestimar la complejidad de un proyecto e ignorar los cambios en los requerimientos son razones básicas de por qué un proyecto falla. La estimación de un proyecto normalmente parte de una definición de requerimientos, que es producida por actividades de la IR. El cambio en los requerimientos se ataca mediante la *gestión de cambios de los requerimientos*, actividad propia de la IR. De esta manera se concluye que la IR tiene un impacto importante en la concreción de un proyecto de software. Esta conclusión se ve respaldada en [HL01] donde se dice que la causa más grande por la que falla un proyecto de software es contar con requerimientos deficientes . Relacionado con esta afirmación se encuentra el estudio de Caper Jones [Jon91], que habiendo estudiado varios cientos de organizaciones, descubre que más del 75 % de las mismas tienen una deficiente Ingeniería de Requerimientos.

La primera medida del éxito de un sistema de software es el grado en que cumple con el propósito para el cual fue previsto [NE00].

Con frecuencia es posible estimar los costos de un proyecto, el calendario y la factibilidad técnica a partir de especificaciones precisas de los requerimientos [NE00].

Cuando los requerimientos están mal definidos y los procesos de la IR son *ad hoc*, el resultado final es casi siempre un producto poco satisfactorio o un proyecto cancelado [VCBC04].

Los productos de un *buen análisis de requerimientos* no sólo incluyen la definición del sistema sino una adecuada documentación de las funciones, performance, interfaces internas y externas y atributos de calidad del sistema bajo desarrollo, así como también restricciones en el diseño del sistema y en el proceso de desarrollo [DT97].

Boehm estimó que la corrección tardía de los errores en los requerimientos puede llegar a costar 200 veces más que la corrección durante la IR [Boe81].

Una encuesta sobre 8000 proyectos llevados a cabo por 350 compañías de los Estados Unidos de América reveló que un tercio de los mismos nunca fue completado y la mitad tuvo solamente un éxito parcial, esto es, con parte de las funcionalidades, con costos más altos de los previstos, o con atrasos significativos [Gro95]. Al consultar los motivos de tales fracasos, los gerentes ejecutivos identificaron como la mayor fuente de problemas a los malos requerimientos. Más específicamente, la falta de participación de los usuarios (13 %), requerimientos incompletos (12 %), requerimientos que cambiaron (11 %), expectativas no realistas (6 %) y objetivos poco claros (5 %).

Una encuesta llevada a cabo en 3800 organizaciones de 17 países de la Unión Europea concluyó que la mayoría de los problemas del software son en el área de especificación de requerimientos (>50 %) y en la administración de los requerimientos (50 %) [Ins96].

Discusión

Si bien en nuestro Framework no le restamos importancia a la IR entendemos que es difícil fallar en la obtención de los mismos en los proyectos académicos. Esto se debe a lo mencionado anteriormente: *los stakeholder son pocos*. Además los usuarios, a veces clientes, mantienen un constante trabajo con los desarrolladores por lo que es fácil para estos últimos entender qué es lo que tienen que hacer.

2.5. ¿Por Qué es Compleja la Ingeniería de Requerimientos?

La parte más difícil del desarrollo de sistemas de sofware es decidir exactamente qué construir. Por lo tanto, la función más importante que realiza el desarrollador de software para un cliente es la extracción y refinamiento interactivo de los requerimientos del producto [Bro87].

Basado en la definición dada por Zave [Zav97] de la IR, que se presentó en el primer tópico de esta sección, Lamsweerde [vL00] explora por qué la IR es tan compleja y presenta los siguientes puntos:

- El alcance (de la IR) es bastante extenso, ya que varía desde un mundo de organizaciones humanas o leyes físicas a artefactos técnicos que deben ser integrados en él; desde objetivos de alto nivel hasta recetas operacionales; y desde lo informal a lo formal. El sistema objetivo no es simplemente una pieza de software sino que incluye el ambiente que lo va a rodear; este, compuesto por humanos, dispositivos, y/u otro software. El sistema completo debe ser considerado bajo muchas facetas, por ejemplo, socio-económicas, físicas, técnicas y operacionales entre otras.
- Hay múltiples asuntos que deben ser considerados además de los de la funcionalidad, por ejemplo, seguridad, seguridad de la persona, usabilidad, flexibilidad, rendimiento, robustez, interoperabilidad, costo, mantenibilidad. Estos tópicos no-funcionales están normalmente en conflicto.
- Hay muchas partes involucradas en el proceso de la IR, con cada una teniendo diferente origen, procedencia, formación, habilidades, conocimiento, preocupaciones, y percepciones; estas partes son: clientes, usuarios, expertos del dominio, ingenieros de requerimientos, desarrolladores de software o personas de mantenimiento del sistema. Muy a menudo estas partes tienen puntos de vista encontrados.
- Las especificaciones de requerimientos pueden sufrir una gran variedad de deficiencias. Algunas de estas son errores que pueden tener efectos desastrosos en los pasos subsecuentes del desarrollo y en la calidad del producto de software resultante.

- La IR abarca múltiples actividades entrelazadas. Algunas de ellas son: análisis del dominio, obtención de los requerimientos, negociación y acuerdo sobre los requerimientos, especificación de los requerimientos, documentación de las decisiones tomadas durante el proceso de IR y evolución de los requerimientos.

Hay un número de dificultades inherentes a este proceso *[proceso de IR]*. Los *stakeholders* (incluyendo clientes, usuarios y desarrolladores) pueden ser numerosos y estar distribuidos en distintas regiones o lugares de la Tierra. Sus objetivos pueden variar y estar en conflicto, dependiendo de su perspectiva del entorno en el cual trabajan y las tareas que desean llevar a cabo. Sus objetivos pueden no ser explícitos o pueden ser difíciles de articular, e inevitablemente, el logro de esos objetivos puede estar restringido por una variedad de factores fuera de su control [NE00].

Discusión

Las investigaciones que se citan en la sección anterior sobre la complejidad de la IR, son claramente aplicables a proyectos de la industria. Por otro lado, entendemos que estas complejidades no son aplicables al desarrollo académico de software en general y en particular al desarrollo en el CSI.

2.6. El Proceso de Ingeniería de Requerimientos

Distintos autores presentan distintos, pero similares, procesos para la ingeniería de requerimientos.

Sommerville [Som04] presenta dos procesos de IR. Ambos incluyen los mismos sub-procesos, uno es un proceso en cascada y el otro es un proceso en espiral. Los sub-procesos presentados son: "estudio de factibilidad", "obtención y análisis de requerimientos", "especificación de requerimientos" y "validación de requerimientos". Al primer sub-proceso le concierne la evaluación de si el sistema es útil para el negocio. El segundo se encarga de descubrir los requerimientos. El tercero convierte los requerimientos obtenidos en una forma estándar. El último chequea que los requerimientos realmente definan el sistema que el cliente quiere.

La IR debe incluir cuatro actividades separadas pero relacionadas: obtención, modelado, validación, y verificación. Típicamente, primero obtenemos los requerimientos desde cualquier fuente que esté disponible(expertos, repositorios o el software actual), luego los modelamos para especificar una solución. El modelado describe una solución en el contexto de un dominio de aplicación usando una notación informal, semi-formal o formal. Después se valida y verifica la especificación. Esto da retroalimentación a los *stakeholders* en la interpretación de sus requerimientos, pudiendo corregir malentendidos lo más pronto posible [HL01].

Discusión

No es de interés para este trabajo presentar un proceso para la IR. Entendemos que las fases más importantes para los proyectos académicos son el correcto modelado de los requerimientos y la validación con el usuario de forma iterativa. El modelado es de suma importancia ya que es usado en el momento de validar con el usuario.

2.7. Identificación de Stakeholders

Existen muy pocos artículos que discutan cómo identificar los *stakeholders* para un sistema. A continuación se presenta un artículo que trata el tema.

Sharp, Finkelstein y Galal [SFG99], afirman que las definiciones de *stakeholder* que existen en la literatura no ayudan a resolver el problema práctico de identificar un conjunto de *stakeholders* relevantes para un proyecto dado. Para resolver este problema proponen un enfoque para identificar *stakeholders* durante la IR. Primero definen las interacciones entre los mismos. Interacciones entre los *stakeholders* incluyen: intercambio de información, productos, o instrucciones, o proveer tareas de soporte. Es necesario identificar y guardar la información sobre los *stakeholders* y la naturaleza de sus relaciones e interacciones. Las dimensiones de importancia son: relaciones entre los *stakeholders*, la relación que tiene cada *stakeholder* con el sistema y la prioridad que se le da a cada punto de vista de un *stakeholder*. Esta información es necesaria para administrar, interpretar, balancear y procesar las entradas de los *stakeholders*. El método para identificar *stakeholders* propone cuatro tipos de *stakeholder*, stakeholder "baseline", stakeholder "proveedor", stakeholder "cliente" y stakeholder "satélite". Se parte de la identificación de los baseline y a partir de ellos se identifican los clientes, los proveedores y los satélites. El método va generando una red de stakeholders. Para identificar esta red se propone seguir estos cinco pasos:

1. Identificar todos los roles específicos dentro del grupo de stakeholders baseline.
2. Identificar los stakeholders proveedores para cada rol baseline.
3. Identificar los stakeholders clientes para cada rol baseline.
4. Identificar los stakeholders satélites para cada rol baseline.
5. Repetir los pasos 1 a 4 para cada grupo de stakeholders identificado en los pasos 2 a 4.

Se definen y explican cuatro grupos dentro del grupo skateholders baseline, estos son: usuarios, desarrolladores, legisladores (tales como: agencias de gobierno, sindicatos, abogados, auditores de aseguramiento de la calidad) y personal implicado en la toma de decisiones. Los skateholders proveedores proveen información o tareas de soporte a los baseline. Los stakeholders clientes procesan o inspeccionan los productos de los baseline. Los skateholders satélites interactúan con los baseline en una forma variada, como por

ejemplo: comunicación y búsqueda de información. Este enfoque de identificación de *stakeholders* se basa en las interacciones entre los mismos más que en las relaciones entre estos y el sistema, esto se debe a que la primera relación es más fácil de seguir. Para los *stakeholders* proveedores, clientes y satélites, los autores definen roles que facilitan su identificación. El beneficio más grande de este enfoque, según los autores, es que se pueden capturar todos los *stakeholders* que son relevantes y no incluir aquellos que no lo son. Uno de los defectos más grandes de este enfoque, según los autores, es que se corre peligro en gastar mucho tiempo en identificar los roles y las relaciones entre los mismos. Como trabajo a futuro se propone validar el enfoque propuesto con un proyecto real [SFG99].

Discusión

No entendemos como necesario presentar formas de identificación de *stakeholders* en nuestro Framework. Entendemos que los *stakeholders* se identifican fácilmente en la mayoría de los proyectos académicos.

2.8. Técnicas para la Obtención de Requerimientos

Uno de los objetivos más importantes de la obtención de requerimientos es encontrar qué problemas se necesitan resolver, y a partir de estos, encontrar las fronteras del sistema. Identificar y ponerse de acuerdo en las fronteras del sistema, afecta todos los esfuerzos siguientes de la obtención de requerimientos. La identificación de *stakeholders*, clases de usuarios, objetivos, tareas, escenarios y casos de uso, todos dependen de cómo se eligen las fronteras del sistema [NE00].

En [NE00], los autores identifican distintas técnicas para la obtención de requerimientos:

- Las *técnicas tradicionales* incluyen una extensa clase de técnicas de obtención de datos. Estas incluyen el uso de cuestionarios y encuestas, entrevistas, y análisis de documentación existente.
- Las *técnicas de obtención grupales* buscan fomentar el acuerdo entre *stakeholders* mientras se explotan dinámicas de grupo para obtener un mejor entendimiento de las necesidades. Estas incluyen: tormentas de ideas, grupos de foco, así como también talleres RAD/JAD.
- El *prototipado* ha sido utilizado para la obtención de requerimientos cuando hay un alto grado de incertidumbre sobre los requerimientos o cuando es necesaria la retroalimentación temprana de los *stakeholders*.
- Las *técnicas guiadas por el modelo* proveen un modelo específico del tipo de información a ser obtenida, y se usa ese modelo como técnica de obtención de requerimientos. Estas incluyen a métodos basados en *goals* como KAOS e I*, y métodos basados en escenarios como CREWS.

- Las *técnicas cognitivas* incluyen una serie de técnicas originariamente desarrolladas para adquisición de conocimiento en sistemas basados en el conocimiento. Estas técnicas incluyen: análisis de protocolo, escalera y ordenamiento de tarjetas.

- Las *técnicas contextuales* surgen en los años 90 como una alternativa a las técnicas tradicionales y a las técnicas cognitivas. Estas incluyen técnicas etnográficas tal como observación de participantes. También incluyen a la etnometodología y al análisis de conversaciones.

Una pregunta básica en IR es cómo descubrir qué es lo que los usuarios realmente quieren [GL93]. En el artículo *Techniques for requirements elicitation* [GL93], Goguen y Linde inspeccionan varios métodos para la obtención de requerimientos: introspección, entrevistas, cuestionarios, conversación, interacción, análisis de protocolos y análisis de conversaciones. Los autores afirman que la obtención de requerimientos no puede ser resuelta en una forma puramente tecnológica, porque el contexto social es mucho más crucial que en las fases de especificación, diseño e implementación. Dividen las entrevistas, como forma de obtener los requerimientos, en: entrevistas con cuestionario, se sigue un cuestionario en la realización de la entrevista donde el mismo tiene un conjunto de respuestas posibles para cada pregunta, entrevistas abiertas, en este tipo de entrevistas las respuestas no están atadas a un conjunto de respuestas posibles, grupos para enfocar y grupos de desarrollo de aplicaciones, son tipos de entrevistas grupales (un tipo conocido es *Joint Application Development Group*). El análisis de protocolo se basa en que una persona realice una tarea y que en voz alta vaya explicando sus procesos mentales para poder realizar la misma. Los autores analizan los pros y contras de cada técnica y proponen su uso de forma complementaria, dejando de lado el análisis de protocolo, que lo ven como una técnica que no es útil dentro del análisis de requerimientos.

En [SSV98], se introduce al lector en el uso del enfoque *multi-perspective requirements engineering* (PREview). En el mismo se describe el proceso de PREview, el cual ha sido designado para permitir la obtención de requerimientos de forma incremental.

En [HPW98], se propone el uso de medios de comunicación (video, discursos, dibujos, etc.), para la obtención y validación de los requerimientos. Se propone para esto capturar el uso del sistema actual e interrelacionar estas observaciones con las definiciones de los *goals*. Los autores dicen que su enfoque no puede ser igualmente bien aplicado para cualquier tipo de proyecto. Es apropiado para proyectos en los cuales la funcionalidad del sistema viejo puede ser observada y en el cual su funcionalidad va a ser provista, en gran parte, por el sistema nuevo. Proyectos innovadores, en los cuales no hay un sistema precursor, se van a beneficiar menos con este enfoque. El enfoque propuesto ha sido validado con una pequeña aplicación de prueba realizada en una empresa llamada ADITEC.

Discusión

Las técnicas que sugerimos para la obtención de requerimientos en el desarrollo académico de software son entrevistas con los usuarios y prototipación. Como este emprendimiento surge en el grupo Concepción de Sistemas de Información (CSI), y debido al tipo de desa-

rrollo que se realiza dentro del grupo, se ve como una técnica interesante de obtención de requerimientos el uso de escenarios.

También existen casos donde el proyecto es una composición e integración de soluciones existentes, para estos casos también es importante especificar los requerimientos y es conveniente partir de especificaciones de las soluciones que se van a utilizar. En caso de no existir las especificaciones se deberá usar ingeniería reversa para tener el documento de especificación del nuevo proyecto.

Creemos que el enfoque propuesto en [HPW98], donde se usan videos, discursos, dibujos, entre otros, para relevar los requerimientos, es muy complejo de llevar a cabo y prepararlo y analizar los resultados consume mucho tiempo. Como punto importante queremos resaltar que este enfoque fue validado en casos reales.

2.9. Uso de Escenarios en la Ingeniería de Requerimientos

En el artículo [Gli00], se presenta la potencialidad que tienen los escenarios para mejorar la calidad de los requerimientos, así como también una forma de representar sistemáticamente los escenarios individuales y representar la estructura y las relaciones de un conjunto de escenarios. Para esto último usan una estructura basada en diagramas de estado. A continuación se listan las ventajas clave del uso de los escenarios en la IR según el artículo mencionado:

- *Considerar el punto de vista del usuario.* Un escenario siempre ve el sistema desde el punto de vista de uno de sus usuarios. Esta es una ventaja fundamental en el momento de validar la adecuación de los requerimientos.
- *Especificaciones parciales.* Los escenarios son una forma natural de escribir especificaciones parciales. Cada escenario captura una secuencia de interacciones entre un usuario y el sistema, representando una transacción del sistema o una función del sistema desde la perspectiva del usuario. Una fortaleza de los escenarios está en el hecho de que proveen una descomposición del sistema en funciones desde una perspectiva del usuario y que cada una de estas funciones puede ser tratada separadamente.
- *Fácil de entender.* Los escenarios simplifican tanto la obtención como la validación de requerimientos, debido a que la noción de interacción entre usuario y sistema resulta ser una forma natural de entender y discutir requerimientos con los usuarios y con los ingenieros de requerimientos.
- Pequeños ciclos de retroalimentación. Los pequeños ciclos de retroalimentación entre los usuarios y los ingenieros de requerimientos son logrados debido a la combinación de la habilidad de tratar cada función de usuario de forma separada en un escenario y la forma orientada al usuario de representar los requerimientos en los mismos.
- *Base para el testing del sistema.* Las secuencias de interacción capturadas en los escenarios son una base ideal para definir el test del sistema.

En [AEG+98], se presenta un extenso estudio del uso de escenarios en la industria. En particular se estudian doce proyectos seleccionados, nueve de Alemania y tres de Suiza. Los hallazgos de esta investigación son muchos y variados y se dividen en: "contenidos y presentación de los escenarios", "objetivos del uso de escenarios", "relación entre los escenarios y el proceso de desarrollo" y "beneficios, problemas, necesidades y planes futuros relacionados con el uso de escenarios". La encuesta revela que el enfoque de escenarios es flexible y aplicable en términos generales. Se usan para múltiples propósitos en distintas fases del desarrollo de los proyectos, por ejemplo, durante las primeras fases de la obtención y especificación de requerimientos, así como también durante la fase de diseño y con propósitos de verificar y validar. Los datos recolectados prueban que los escenarios son aplicables en proyectos de tamaño variado y de diferentes dominios de aplicación. Nueve de doce de los proyectos volverían a usar escenarios en proyectos nuevos y ninguno de los proyectos reportó un fracaso debido al uso de escenarios. Por lo tanto uno podría decir que los escenarios son empleados con éxito en la práctica.

En [WPJH98], se realiza un estudio de 15 proyectos de software, en cuatro países europeos. Los resultados obtenidos son similares a aquellos presentados en [AEG+98].

Discusión

Varios motivos y estudios de casos reales se presentan en la sección anterior sobre el uso de escenarios en la IR. A nuestro entender el uso de escenarios en los casos que existe mucha interacción entre un usuario (o actor) es casi indispensable y da buenos frutos. Los mismos aportan en la comunicación entre los usuarios y los desarrolladores, permiten validar los requerimientos y son una base para el posterior testing del sistema.

En el Framework se establecen actividades para crear los casos de uso y para validar los mismos.

2.10. Ingeniería de Requerimientos Orientada a *Goals*

La IR orientada a *goals* ha sido muy estudiada en los últimos diez años. Muchos artículos se han escrito sobre el tema. La idea principal de este enfoque es considerar los *goals* (metas) de la organización que necesita el sistema. Normalmente asociamos la pregunta ¿qué se debe construir? con el análisis de requerimientos, este enfoque cambia la pregunta a: ¿por qué se necesita el sistema? antes de hacerse la pregunta anterior. Antón, en [Ant96], presenta un panorama de las técnicas orientadas a *goals* y describe estrategias para el análisis de los *goals* y la evolución de los mismos. Los *goals* son útiles para organizar y justificar los requerimientos de software. Antón define los siguientes conceptos en el artículo mencionado:

- Los *Goals* son objetivos de alto nivel del negocio, organización o sistema. Capturan las razones de por qué un sistema se necesita y guía las decisiones a varios niveles dentro de la empresa.

- Un *requerimiento* especifica cómo un *goal* debe ser logrado por el sistema propuesto.
- *Operacionalización* es el proceso de definir un *goal* con suficiente detalle de modo que sus *sub-goals* tengan una definición operativa.
- Los *agentes* son las entidades o procesos que buscan lograr *goals* dentro de una organización o sistema, basados en la responsabilidad implícita que ellos deben asumir para el logro de ciertos *goals*.
- Las *restricciones* son requerimientos que deben ser alcanzados para completar *goals*. Una restricción pone una condición sobre el logro de un *goal*.
- La *descomposición* de los *goals* es el proceso de subdividir un conjunto de *goals* en un subgrupo lógico, por lo que los requerimientos del sistema pueden ser entendidos, definidos y especificados más fácilmente.
- Los *obstáculos* de los *goals* son comportamientos u otros *goals* que previenen o bloquean el logro de un *goal* dado. Identificar los obstáculos de los *goals* nos permite considerar las formas posibles en que un *goal* puede fallar.

En el artículo se discute el método Goal-Based Requirements Analysis Method (GBRAM). El mismo sirve para identificar, elaborar, refinar y organizar *goals* para especificaciones de requerimientos.

Discusión

No nos parece necesario realizar un análisis de los *goals*, o partir de estos para obtener los requerimientos dentro del desarrollo académico de software.

2.11. Etnografía y Aspectos Sociales de la Ingeniería de Requerimientos

La etnografía es popular en la comunidad de IR. Esta es una rama de la antropología que provee descripciones científicas de las sociedades humanas. Numerosos estudios se realizaron en distintos dominios, incluyendo: control de tráfico aéreo [BHR+92] y la banca [BRH97].

En un artículo [VS99], Viller y Sommerville describen tres métodos diferentes de uso de la etnografía en la IR. Estos tres métodos han sido utilizados en aplicaciones reales. Estando cierto tiempo junto con los trabajadores, los etnógrafos desarrollan un profundo entendimiento del trabajo. Entonces, los etnógrafos pueden proveer a los desarrolladores con un detallado conocimiento del trabajo describiendo la forma en la cual es llevado a cabo, siendo este presentado en el lenguaje y terminología de los usuarios. Los autores mantienen que la etnografía tiene mucho para ofrecer como técnica para la IR. Sin embargo, listan una serie de problemas que pueden limitar su uso en la práctica:

- *Tiempo*. La etnografía puede ser un proceso muy largo, durando meses e incluso años en el contexto de una investigación social. Escalas de tiempo mucho más chicas son necesarias en el proceso de la IR.

- *Resultados*. La etnografía tiende a producir una gran cantidad de descripciones textuales detalladas como resultado de realizar un estudio. Comunicar efectivamente esos resultados a los ingenieros de requerimientos no es sencillo.

- *Cultura*. Existen diferencias significativas en el lenguaje y la cultura entre sociólogos y desarrolladores de software. Esto puede llevar a problemas de comunicación entre los dos grupos.

- *Aptitud*. La falta de un enfoque sistemático para conducir el estudio etnográfico hace que la técnica dependa de la aptitud individual del etnógrafo.

Cada uno de los métodos presentados es una mejora del anterior. El primer método implica un trabajo muy cercano entre los etnógrafos y los desarrolladores. El modelo de trabajo tiene períodos de trabajo de campo de los etnógrafos en paralelo con desarrollo de prototipos por parte de los ingenieros de software. Esto es seguido de reuniones donde se reportan los hallazgos al resto del equipo. Dentro de las fortalezas de este método se indica que el etnógrafo puede llevar un conocimiento profundo sobre el dominio a las reuniones de reportes de hallazgos. La particularidad del conocimiento que llevan los etnógrafos es que es *realmente lo que ocurre* en el trabajo. La debilidad de este enfoque es que la riqueza de las notas del etnógrafo quedan sin explotar en gran parte debido a que son muy detalladas, no estructuradas y de una naturaleza muy personal. El segundo método que se presenta cambia la forma de presentar los descubrimientos realizados por los etnógrafos. Los etnógrafos deben presentar sus análisis usando *puntos de vista*, técnica desarrollada en la universidad de Lancaster, cada uno de estos dirigido a un aspecto particular de la organización social del trabajo. También se usa un marco de presentación dividido en tres dimensiones de trabajo que estructuran los datos recabados por los etnógrafos: "coordinación distribuida", "planes y procedimientos" y "conciencia de trabajo". Tanto los puntos de vista como las dimensiones cambian la forma de presentar el trabajo de los etnógrafos, siendo esta una forma mucho más estructurada. Este hecho y que la orientación de los datos sea hacia la IR son las fortalezas más grandes de este enfoque. Este enfoque sigue teniendo la debilidad de que los problemas de comunicación entre los etnógrafos y los ingenieros de software no está resuelta. El tercer método deja a los ingenieros de software el trabajo de analizar la naturaleza social del espacio de trabajo. Para realizar este trabajo usan una guía etnográficamente informada. El método presenta a esta guía de una manera sistemática usando *puntos de vista* para estructurar la obtención y el análisis de requerimientos. Una de las más importantes fortalezas de este método es que es flexible, de modo que puede complementar a un enfoque de IR orientado a *puntos de vista*, o ser usado para proveer el análisis social como punto de entrada a cualquier otro enfoque. Una de las debilidades del método es que no elimina los problemas de comunicación, sino que los pasa desde uno de entendimiento entre personas en una conversación a uno de entendimiento entre personas vía un documento.

Discusión

Por el tipo de sistemas que normalmente se construyen y por la poca cantidad de usuarios que existen normalmente, no creemos necesario que se hagan estudios etnográficos en los proyectos de desarrollo académico de software.

2.12. Ingeniería de Requerimientos para Productos del Mercado (no a medida)

En [RBE98], se describe un proceso específico industrial de IR para software empaquetado, llamado REPEAT (Requirements Engineering ProcEss At Telelogic). El proceso es usado en la empresa Telelogic AB; empresa sueca que se dedica a la venta de herramientas CASE. El proceso es usado en Telelogic para obtener, seleccionar y administrar los requerimientos de una familia de productos llamada Telelogic Tau. Los autores afirman que la IR de software empaquetado es diferente de la IR de software hecho a medida, y por eso se creó el proceso REPEAT. En los proyectos hechos a medida el cliente está bien definido y la especificación de requerimientos es normalmente un contrato entre los desarrolladores y el cliente. Cuando se desarrolla un producto empaquetado el proceso de IR debe ser capaz de inventar requerimientos basado en usuarios finales previstos y seleccionar un conjunto de requerimientos que resulten en un producto de software que pueda competir en el mercado. REPEAT es un proceso que administra los requerimientos a través de un ciclo completo de desarrollo. El mismo cubre actividades típicas de la IR, tales como, obtención, documentación y validación, y tiene un fuerte foco en la selección de requerimientos y planificación de la liberación de versiones. Antes de instaurar REPEAT, en Telelogic se tenía un proceso ad hoc para la administración de los requerimientos, y se tenían problemas con la precisión de la liberación de las versiones y con la calidad de los productos. La versión 3.0 de la familia de productos fue liberada ocho meses después de lo planificado y la versión 3.1 tuvo tres meses de atraso. Entre la versión 3.0 y la 3.1 se necesitaron liberar varias versiones intermedias para enmendar problemas de calidad y agregar nuevos requerimientos. Cuando REPEAT fue aplicado la versión 3.2 tuvo sólo quince días de retraso y la versión 3.3 fue liberada tres días antes de lo planificado. La calidad del producto ha mejorado y esto se ha medido usando la cantidad de defectos reportados entre la versión 3.1 y la versión 3.3. Los autores están convencidos de que la mejor explicación para estos logros es la introducción del proceso REPEAT.

Discusión

Normalmente en el desarrollo académico de software no se construyen productos de mercado. Cabe resaltar que en el estudio de campo realizado ninguno de los productos tuvo esta característica.

2.13. Estado de la Práctica y Estudios Empíricos en la Ingeniería de Requerimientos

Hay pocos artículos en la literatura que traten el estado de la práctica en la IR. Dos artículos [LNJ02], [NL03] discuten los resultados obtenidos al realizar una encuesta de 22 preguntas accesible desde la Web. La encuesta se realizó entre estudiantes actuales y graduados de *Penn State Great Valley School of Graduate Professional Studies*. Sólo quedaron incluidos en la encuesta profesionales de la industria ya que la escuela de graduados acepta solamente a profesionales que están trabajando. Se tuvieron 194 encuestas completas, capturando una diversidad considerable de industrias, incluyendo entre estas algunas Fortune 500 y corporaciones multinacionales. Las compañías incluyeron a Lockheed Martin Managment and Data Systems, Merck, SAP, Unisys, Wyeth-Ayerst, M&M Mars, Vanguard, Boeing, AstraZeneca, Dupont, Siemens Medical Systems, Verizon, GlaxoSmithKline y numerosas empresas pequeñas. Los dominios de aplicación que abarcó la encuesta fueron muy variados, entre otros fueron los siguientes: telecomunicaciones, educación, gobierno, imágenes, aeroespacial, farmacia y biotecnología, finanzas, sistemas médicos, utilitarios, defensa, sistemas de soporte y manufactura. Respecto a la forma de obtención de requerimientos se obtuvo que más del 50 % de los encuestados usan escenarios o casos de uso. Por otro lado, para el análisis y modelado de los requerimientos 33 % indica que no usa ninguna metodología y 30 % usa análisis orientado a objetos. Otras preguntas tratan sobre el grado de formalidad usado para representar los requerimientos. En esta encuesta se ve que las representaciones formales son raras (7 %), sin embargo, las representaciones informales llegan al 51 %. Los autores usan dos preguntas para saber si la falta de formalidad impacta en el producto final: "Los usuarios finales encuentran fácil de usar al producto terminado" y "El producto terminado sirve a las necesidades del cliente o del usuario". Sólo 69 % de los encuestados que usan representaciones formales estuvieron de acuerdo con la primera oración, 79 % de los que respondieron representación informal y 76 % de los que indicaron semi-formal. Se encuentran resultados similares respecto a la segunda oración, 61,5 % de los que usan representaciones formales estuvieron de acuerdo con la segunda oración, 70 % de los que usan representación informal y 73 % de los que usan representación semi-formal. Otro dato interesante es que 59 % de los encuestados realizan inspecciones de requerimientos, utilizando para esto, un variado rango de técnicas.

Otro artículo que trata el estado de la práctica en IR es [NSK00]. En esta investigación se encuestan 12 compañías de Finlandia. La encuesta abarcó puntos tales como: las practicas actuales en IR, las necesidades de desarrollo y mejora en la IR y las expectativas que tiene la industria en la investigación en la ingeniería de software y en particular en la IR.

En [VCBC04] se presentan los resultados de una encuesta realizada sobre 164 proyectos de Australia y Estados Unidos. Estos proyectos tienen la particularidad de ser proyectos de desarrollo de software *in-house* y que el 86 % de las empresas son consideradas chicas. Se encuentra que es más importante para el éxito de un proceso tener una metodología de desarrollo de software definida y que esta sea apropiada para el proyecto que tener una metodología definida para la obtención de requerimientos. Otros descubrimientos que

surgen del análisis de la encuesta son los siguientes. Tener buenos requerimientos, que estén completos y precisos al comenzar el proyecto, con un alcance del proyecto bien definido, resultando en deliverables de software bien definidos, están todos positivamente correlacionados con el éxito del proyecto. Se halló que existe una evidente importancia en el involucramiento de los usuarios durante la obtención de los requerimientos. Se encontró que si los requerimientos están incompletos al comienzo del proyecto, completar los mismos durante el transcurso del proyecto está positivamente relacionado con el éxito del proyecto. Aunque hay afirmaciones teóricas distintas, en esta encuesta no se encontró que cambiar el alcance durante el proyecto esté relacionado con el fracaso del mismo. Por otro lado se encontró que administrar efectivamente los requerimientos y los cambios en los mismos a través de un repositorio centralizado está positivamente relacionado con el éxito de un proyecto. Cuando el tamaño del proyecto impacta sobre la obtención de los requerimientos es más probable que el proyecto falle. Basado únicamente en las preguntas de la encuesta que tratan el tema requerimientos, se tiene que el mejor predictor para el éxito de un proyecto es que los requerimientos sean buenos, que predice un 89 % de éxito, 58 % de fracasos y un 78 % de proyectos correctos en su totalidad. El artículo presenta un estudio de las preguntas que refieren a sponsors, clientes y usuarios, así como también a la gerencia del proyecto. Por simplicidad mencionamos aquí solamente las afirmaciones que por sí solas predicen el mayor éxito de los proyectos. Respecto a los sponsors, clientes y usuarios, la afirmación: "hubo un alto nivel de confianza de los clientes/usuarios en el equipo de desarrollo" tuvo un 70 % de proyectos correctos en su totalidad. En lo que respecta a la gerencia del proyecto, la afirmación: "los requerimientos fueron administrados efectivamente" predice un 77 % de proyectos correctos. Cuando se hace un estudio de las correlaciones entre las afirmaciones, se encuentra que la que logra predecir el más alto número de éxito en los proyectos (93 %) es la combinación entre "los requerimientos son buenos" y "los requerimientos fueron administrados efectivamente".

Juristo, Moreno y Silva encuestaron a 150 profesionales de organizaciones europeas, los cuales tienen un nivel de responsabilidad de mediano a alto en el proceso de IR en sus compañías [JMS02]. Estas encuestas cubren a 11 organizaciones en 7 países de Europa. Se halló que aún la inmadurez define las prácticas actuales en la IR. Otros resultados de la encuesta son los siguientes: Técnicas para la obtención y negociación de requerimientos no son usadas lo suficiente. No hay definidos formalmente documentos de especificación de requerimientos. Se detecta también que existe un problema en el momento de elegir qué herramientas son más apropiadas para el tipo de proceso y clase de aplicación.

Discusión

Los estudios empíricos ayudan a conocer cómo se aplican realmente los conocimientos del área. En nuestro caso no entendemos que sea necesario tener este tipo de estudios en el desarrollo académico de software. Sin embargo, los mismos sirven para ver qué se está usando en este momento y qué cosas tienen un efecto positivo en el desarrollo de software.

2.14. Estudio de Aplicaciones Reales

Existen en la literatura muchos artículos que presentan casos de estudio de la vida real. Breitman et al. [BLF99] presentan un extenso trabajo que busca analizar la posible aplicación de nuevos descubrimientos de la investigación en un caso de estudio de la vida real. El caso de estudio es el conocido London Ambulance Service System (LASS), descrito en [Fin93]. Este caso fue un gran fracaso en la construcción de un sistema y la idea del artículo es presentar los tópicos problemáticos del LASS así como presentar discusiones sobre cómo los recientes avances en los métodos, las técnicas y las herramientas de la Ingeniería de Requerimientos podrían haber sido aplicados para minimizar los problemas detectados. Se analiza el uso de *puntos de vista*, la evolución de los requerimientos, los aspectos sociales y los requerimientos no funcionales.

2.15. Administración de los Requerimientos

Sommerville [Som04] presenta de forma concisa y clara la administración de los requerimientos. Los requerimientos para un sistema de software grande están siempre cambiando. Entonces, los requerimientos deben evolucionar para reflejar estos cambios. La administración de los requerimientos es el proceso de entender y controlar los cambios en los requerimientos del sistema. Se necesita "seguirle el rastro" a cada requerimiento individual y mantener los vínculos entre los requerimientos dependientes de forma de poder calcular el impacto de los cambios en los requerimientos. Se debe establecer un proceso para hacer las propuestas de cambios. Este proceso debe ser usado cada vez que se propone un cambio en los requerimientos, la ventaja de usar el mismo es que todas las propuestas de cambios son tratadas consistentemente y que los cambios al documento de requerimientos son realizados de una forma controlada.

Discusión

Debido a la complejidad y al tamaño de los proyectos que se desarrollan en la academia no se considera necesaria la administración de los requerimientos ni contar con un proceso para administrar el cambio. Esto también se debe a que los proyectos del estudio de campo son proyectos que comienzan desde cero. En caso de proyectos que son continuación de proyectos anteriores la complejidad de los mismos comienza a aumentar así como también su tamaño. En estos casos surge como necesaria la administración de los requerimientos. En una segunda versión del Framework propuesto entendemos como necesario incluir actividades para la administración de los requerimientos.

2.16. Discusión General de la Ingeniería de Requerimientos

En esta subsección se presenta una breve discusión de alguno de los temas presentados en la sección anterior. La discusión se basa en nuestro objetivo primario: la mejora de la calidad en un entorno de desarrollo académico de software.

Como se desprende de esta sección, nosotros entendemos que la IR del desarrollo en la industria difiere de la IR del desarrollo en la academia. Esto simplemente responde a unos pocos pero importantes factores: tamaño y complejidad de los productos que se desarrollan, cantidad de *stakeholders*, cantidad de personas en el grupo de desarrollo y duración de los proyectos. Esto es una afirmación general, aclaramos que pueden existir proyectos particulares en la academia que tengan características de proyectos de software de gran tamaño de la industria.

Normalmente en el desarrollo académico de software las personas que realizan la IR son las mismas que desarrollan. Es común que estas personas no tengan una formación en IR y que su experiencia relevando y especificando requerimientos sea limitada. Parece razonable realizar un estudio del efecto de estas limitaciones sobre el producto final.

A modo de un breve resumen destacamos los siguientes puntos: Es importante para el desarrollo académico de software mantener un documento de especificación de requerimientos aunque este no necesita ser del todo exhaustivo ya que es poco común que se firme un acuerdo entre el cliente y los desarrolladores; normalmente los acuerdos son flexibles. Parece importante el uso de escenarios y casos de uso en los proyectos que tienen una importante interacción entre el usuario y el sistema. No tiene mayor relevancia e importancia en la mayoría de los proyectos académicos la identificación de *stakeholders*, tener un proceso para la IR, utilizar la IR orientada a *goals*, usar técnicas etnográficas para relevar los requerimientos y administrar los requerimientos. Este último punto cobra importancia cuando el proyecto de desarrollo es continuación de proyectos anteriores.

El uso de escenarios o de prototipos para “mostrar” y validar los requerimientos con los clientes se ve reforzado debido a que los clientes en el contexto académico normalmente no están muy dispuestos a leer grandes documentos.

Normalmente en el desarrollo académico de software notamos que existen pocos requerimientos en comparación con la industria y que la complejidad del sistema está más asociada a la profundidad del encare de cada requerimiento que a la cantidad de los mismos. Esto impacta en la forma de establecer el alcance ya que este no está dado fuertemente por un recorte de requerimientos. Entendemos que en el desarrollo académico de software la definición del alcance está ligada a cuán sofisticada va a ser la solución y a la forma en que esta es encarada.

3. Verificación y Validación

En esta sección se presenta el estado del arte en el área Verificación y Validación. Esta está dividida en distintos tópicos que creemos de importancia mencionar en un estudio del estado del arte de la V&V.

La V&V es un área muy vasta dentro de la ingeniería de software, abarcando distintos tópicos, tales como: revisiones de los requerimientos, revisiones del diseño y revisiones y pruebas de código. Se discuten algunos de estos desde el punto de vista del desarrollo académico de software y del Framework propuesto.

3.1. Introducción

La Verificación y Validación es definida de varias maneras diferentes. En nuestro trabajo entendemos a la verificación como el proceso que se lleva adelante para comprobar si una actividad del desarrollo se ha realizado de forma correcta. Por otro lado, entendemos a la validación como una actividad que es llevada a cabo por el cliente (usuario, cliente y/o stakeholder entre otros), con o sin personas del grupo de desarrollo, para comprobar si un producto está de acuerdo a sus expectativas y necesidades. Boehm [Boe79] expresó con dos preguntas las diferencias entre la verificación y la validación:

Verificación. ¿Estamos construyendo el producto correctamente?

Validación. ¿Estamos construyendo el producto correcto?

El principal objetivo de la V&V es encontrar defectos en los distintos productos del desarrollo de software. Estos después deben ser removidos para lograr un producto de mejor calidad. El proceso de remoción de defectos queda, para nosotros, fuera de la V&V.

La V&V se aplica a distintas fases y productos de la ingeniería de software. De todas formas, el papel preponderante de esta disciplina es cuando la misma es aplicada al código. Esto también se ve reflejado en el Framework donde solo dos actividades de V&V no son utilizadas sobre el código.

Es importante conocer la diferencia entre defectos y fallas. Las fallas son manifestaciones externas de los defectos. Estas se dan solamente cuando el software (o sistema) está en funcionamiento. Los defectos son internos a cada producto y pueden provocar fallas.

Es una buena práctica clasificar y registrar los distintos tipos de defectos que se encuentran. Cuando una organización tiene estos datos puede ocuparse de buscar expresamente los tipos de defectos en los que comúnmente incurre la misma. Además, si esta clasificación permite identificar en qué fase o fases del desarrollo se introducen la mayoría de los defectos entonces se puede mejorar el propio proceso de desarrollo.

Se usan dos tipos de técnicas para verificar y validar sistemas:

1. Las técnicas estáticas analizan a los productos de software, como por ejemplo, el documento de requerimientos, los diagramas de diseño y el código fuente. Estas técnicas no requieren poner en funcionamiento el sistema (ejecutar el sistema). Con este tipo de técnicas los defectos son encontrados de forma directa.

2. Las técnicas dinámicas prueban el funcionamiento del sistema. Requieren tener al sistema o porciones del mismo implementadas. Se usan datos de prueba, y luego se examinan tanto las salidas obtenidas como el comportamiento durante la ejecución para comprobar si el sistema se desempeña de la forma esperada. El objetivo de estas técnicas es provocar fallas, por lo tanto, permiten descubrir defectos de forma indirecta.

Se busca, mediante la verificación, detectar defectos antes de que el sistema esté en producción. Esta búsqueda, acompañada de la remoción de los defectos encontrados, tiende a la mejora de la calidad del producto final.

3.2. V&V de Requerimientos

Los objetivos de la V&V de requerimientos son chequear el conjunto de requerimientos que han sido definidos y descubrir posibles problemas con esos requerimientos [SS98].

Se debe verificar que los requerimientos sean correctos, no ambiguos, completos, consistentes, que estén ranqueados por importancia, que sean verificables, modificables y entendibles [IEE98].

Sommerville y Sawyer [SS98] proponen las siguientes formas de verificar y/o validar requerimientos:

- Chequear que el documento de requerimientos cumpla con los estándares.
- Realizar inspecciones[4] formales de requerimientos.
- Usar equipos multi-disciplinarios para revisar los requerimientos.
- Definir y usar *checklist* de verificación de requerimientos.
- *Animar* los requerimientos mediante prototipos.
- Escribir un borrador del manual de usuario.
- Generar casos de prueba para los requerimientos.

[4]Las inspecciones de código se tratan en la subsección 3.3 de este capítulo. Las inspecciones de requerimientos usan la misma metodología pero inspeccionan el documento de requerimientos en lugar del código fuente.

Algunos trabajos buscan realizar la verificación de los requerimientos de forma automática. En [DRT01] se presenta una propuesta para chequear algunas propiedades de calidad. En esta propuesta los requerimientos se deben escribir en XML y se usa el lenguaje XSLT para verificar de forma automática algunas propiedades de calidad deseadas. En [LF91] se presenta un lenguaje para representar vistas de distintos puntos de vista.[5] Proponen un conjunto de heurísticas para realizar un análisis sintáctico de las vistas. Este análisis es capaz de diferenciar entre perdida de información e información contrapuesta, brindando apoyo a la resolución de puntos de vista. En [FFGL00] se presenta un proyecto que busca crear una herramienta para evaluar, de forma automática, la calidad de una especificación de software en lenguaje natural. Existen varios emprendimientos que usan técnicas de procesamiento del lenguaje natural para verificar propiedades de calidad en las especificaciones de software.

Discusión

El desarrollo académico de software, como ya se discutió en la sección 2 de este capítulo, no parece tener grandes problemas con la Ingeniería de Requerimientos. Debido a esto en el Framework no se proponen actividades para verificar propiedades de calidad de la especificación.

Por otro lado, entendemos que el cliente debe revisar las especificaciones para confirmar si la comprensión del problema por parte del grupo de desarrollo es la adecuada. El Framework tiene actividades que consideran la validación, por parte del cliente, de los requerimientos.

3.3. Análisis de Código

En esta subsección se describen técnicas estáticas de verificación de código. Estas sirven para comprobar la correspondencia entre cierto *programa*[6] y su especificación funcional.

Estas técnicas revisan el código sin ejecutarlo por lo que no pueden decir nada acerca de la utilidad operacional ni de las características no funcionales del software. Debido a esto, las técnicas estáticas no sirven para la validación. Para validar el software o sistema se recurre a técnicas dinámicas; el testing.

El análisis de código puede realizarse mediante:

- Distintos tipos de revisiones del código fuente:
  - Revision de escritorio.
  - Inspección.

[5] Los puntos de vista (viewpoints) representan una encapsulación parcial de información sobre los requerimientos del sistema desde una perspectiva particular [SS98].

[6] En este contexto programa es una porción de código ejecutable; por ejemplo, una función, un método o una clase.

    - Recorrida (walkthrough).
- Análisis automatizado de código fuente.
- Verificación formal.

En IEEE Standard for Software Reviews [IEE97] se brindan algunas definiciones que son útiles en este contexto y se presentan a continuación:

Revisión. Proceso o reunión durante la cual un producto de software se presenta a personal del proyecto, gerentes, usuarios, clientes, representantes de usuarios u otras partes interesadas para comentarios o aprobación.

Revisión técnica. Evaluación sistemática de un producto de software por un equipo de personal calificado que examina la adecuación del mismo a su uso previsto e identifica discrepancias respecto a especificaciones y estándares. Las revisiones técnicas también pueden proporcionar recomendaciones de alternativas e incluso examinar alguna de ellas.

Inspección. Examen visual de un producto de software para detectar e identificar anomalías, incluyendo defectos y desviaciones de estándares y especificaciones. Las inspecciones son exámenes por pares dirigidas por facilitadores imparciales que están entrenados en técnicas de inspecciones. La determinación de acciones de remedio es un elemento obligatorio de una inspección de software, aunque la solución no debiera ser determinada durante la reunión de inspección.

Recorrida (walkthrough). Técnica de análisis estático en la que un diseñador o programador dirige miembros del equipo de desarrollo y otras partes interesadas a través de un producto de software y los participantes formulan preguntas y realizan comentarios acerca de posibles defectos, violación de estándares de desarrollo y otros problemas.

Las revisiones de escritorio son aquellas en las cuales el programador revisa su propio código en busca de defectos. Esta técnica es obligatoria en PSP. Existen distintas variantes de este tipo de revisión, siendo una de las más comunes la revisión cruzada, donde dos programadores intercambian código fuente para su revisión.

Las inspecciones y las recorridas tienen características en común. A continuación se presentan algunas de estas características:

- Son realizadas por un equipo. Los integrantes del equipo tienen uno o más roles:
    - Inspector (sólo en las inspecciones). En las inspecciones todos tienen el rol de inspector.
    - Autor. Es el autor de lo que se esté revisando.
    - Lector. Es quien lee lo que se está revisando.
    - Moderador. Ordena la discusión.

  - Secretario. Toma apuntes durante la reunión y hace el acta de la misma.

- El foco está en detectar defectos y no en brindar soluciones para los mismos.
- Son realizadas en la fase del ciclo de vida en la que se desarrolla el producto a revisar. Esto da como resultado un descubrimiento temprano de defectos.
- Son ejecutadas en un conjunto de pasos. Básicamente estos son: estudio individual del producto, reunión de inspección o recorrida, corrección por parte del autor y evaluación de la necesidad de una nueva revisión.
- Permiten unificar el estilo de programación y mejorar la forma de programar.
- Se deben usar para criticar el producto y no al desarrollador del producto.
- No dan buenos resultados si también se usan para evaluar a los desarrolladores.

Un proceso genérico de inspección consiste en una Planificación, donde se elige el equipo, se asignan los roles, se prepara el material necesario y se fija el calendario. Luego se realiza la reunión de Visión de Conjunto. En esta reunión se presenta el proceso de inspección y el producto a inspeccionar. Se sigue con una etapa de Preparación Individual, en esta cada integrante del equipo de inspección usa la técnica de lectura[7] escogida y revisa el producto en procura de defectos. La etapa posterior es la Reunión de Inspección. En esta etapa los revisores se juntan para analizar los defectos encontrados individualmente, eliminar falsos positivos y presuntamente encontrar nuevos defectos. Luego de recolectados los defectos el autor del producto trabaja en la corrección de los mismos. Esta etapa se llama Corrección. Por último, en la etapa de Seguimiento, el moderador debe decidir si se corrigieron los defectos o si es necesario comenzar nuevamente el proceso de inspección. Este proceso genérico se muestra en la Figura 2.5.

Fagan [Fag76] introdujo el proceso de inspecciones al desarrollo de software. En uno de sus estudios, mediante el proceso de inspección, se detectó el 67 % del total de faltas detectadas. Además, al usar inspecciones se tuvo, durante los 7 primeros meses de operación de un sistema, 38 % menos fallas que usando recorridas.

Basili [Bas97] estudia la efectividad de distintas técnicas de lectura, sea cual sea el método en el cual estas están embebidas. El método es por ejemplo, inspecciones o recorridas. Las técnicas de lectura pueden ser ad-hoc, basadas en escenarios o usando *checklists* entre otras. En este estudio las revisiones de código obtuvieron mejores resultados, considerando cantidad de defectos detectados, que las técnicas de testing.

En [KMKT91] se reporta que las revisiones de diseño y código detectaron entre 31 % y 50 % de los defectos en un proyecto experimental en un curso de entrenamiento en Nihon Unisys Ltd.

Tanaka et al. reportan que las revisiones de código detectaron el 31,7 % de los defectos en un estudio en OMRON [TSK+95].

[7] Una presentación de distintas técnicas de lectura queda fuera del alcance de esta tesis

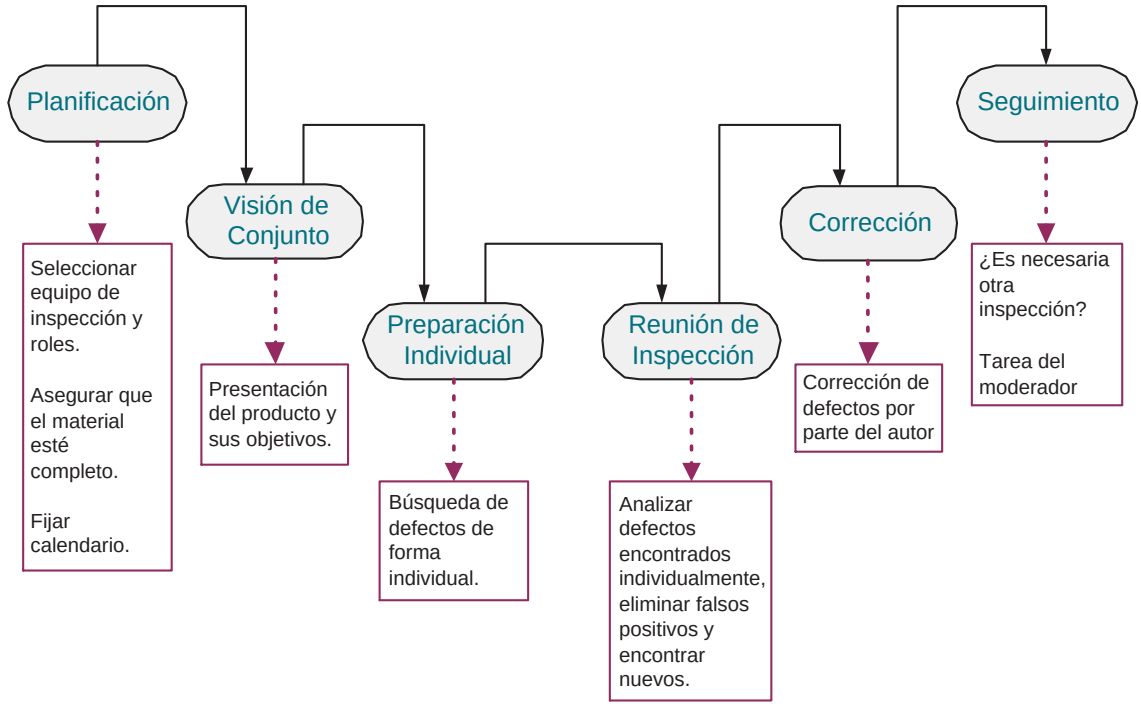


Figura 2.5: Proceso Genérico de Inspección

En [CSM99] se presenta que el 64 % de todos los defectos registrados fueron encontrados mediante inspecciones.

Se conocen en la literatura distintas variantes del proceso genérico de inspección que fue presentado. Nuevos procesos de inspecciones y variantes que se adaptan mejor a un tipo de producto o empresa constituyen una activa línea de investigación.

Las recorridas pueden tener un proceso similar al presentado para las inspecciones. La diferencia es que durante la reunión de grupo, el autor presenta el código fuente *simulando* su ejecución. No se profundiza en este tema ya que no aporta para la discusión del análisis de código dentro del desarrollo en el grupo CSI.

Los analizadores automáticos de código fuente analizan el código fuente, sin ejecutarlo, buscando distintas clases de defectos. Estas herramientas complementan a los compiladores. Por los mismos motivos mencionados para las recorridas no se entrará en detalle en este tema.

La verificación formal es la construcción de software de forma incremental a partir de una especificación en un lenguaje formal. Se busca demostrar (y no mostrar) que el programa es correcto respecto a su especificación. Los tiempos de desarrollo son mayores en comparación con los desarrollos *tradicionales*.

Discusión

Las inspecciones o las recorridas no son tenidas en cuenta en el Framework debido a dos factores. Primero, los grupos de desarrollo están compuestos por una cantidad insuficiente

de personas para poder realizarlas. Segundo, los desarrolladores no han sido entrenados en estás técnicas, por lo que agregarlas en el Framework, aún contando con grupos más grandes, provocaría una baja en la productividad y probablemente no mejoraría la calidad.

Las revisiones de escritorio ya fueron consideradas en la discusión de PSP; subsección 1.2 de este capítulo.

El análisis automático de código fuente tiene mejores resultados con lenguajes menos "modernos" que Java. La mayoría de los desarrollos dentro del grupo CSI son en Java. Además, los ambientes de desarrollo, tales como Eclipse[8] entre otros, brindan algunos aspectos de los analizadores automáticos de forma de complementar al compilador Java. Debido a estos motivos su uso no es ni exigido ni recomendado dentro del Framework.

Nos resulta totalmente desatinado pensar en una verificación formal dentro del contexto planteado. Los motivos son varios. Seguro bajaría la productividad en gran forma y probablemente los productos no fueran terminados. Otro motivo es que los desarrolladores no tienen ninguna experiencia en desarrollos de este tipo. Por último, parece inapropiado esta forma de desarrollo para el tipo de proyectos y productos del grupo CSI. Por estas y otras razones la verificación formal no es considerada en el Framework.

3.4. Testing

Las dos subsecciones anteriores presentaron técnicas estáticas de verificación sobre los requerimientos y el código. En esta subsección se realiza una introducción de las técnicas dinámicas de la V&V; mas conocidas como testing.[9] Esta introducción presenta los conceptos básicos del testing por lo que ayuda a comprender mejor las siguientes subsecciones que tratan temas específicos del testing.

El testing busca encontrar defectos mediante la provocación de fallas. Estas ocurren cuando el software no hace lo requerido o hace algo que no debe. Algunas razones para que esto ocurra son las siguientes:

- Las especificaciones no estipulan exactamente lo que el cliente necesita o quiere. Esto se puede deber a requerimientos faltantes o incorrectos.
- Algún requerimiento no se puede implementar.
- Defectos en el diseño.
- Defectos en el código.

Luego de provocadas las fallas se entra a dos procesos que quedan fuera de la V&V. Primero se realiza la identificación de defectos, que determina qué defecto o defectos causan la falla. Para esto existen diferentes técnicas, una de amplio uso es el debugging

[8] Eclipse: http://www.eclipse.org

[9] En español, prueba. Dentro del tema testing muchas palabras en inglés tienen amplia difusión y sus traducciones al español son poco usadas. Por esto se opta por usar el idioma inglés en muchos casos.

de código. Después se realiza la corrección de defectos, que es el proceso de cambiar el sistema para remover los defectos. Se recomienda luego de realizados los cambios ejecutar pruebas de regresión. Estas pruebas se tratan en la subsección 3.10 de este capítulo.

Muchas veces el defecto que provoca la falla se encuentra *alejado* del lugar donde la falla se manifiesta. Por esto, las pruebas elegidas para testear el sistema deben ayudar a localizar el defecto. Se tienen que intentar realizar pruebas de forma que se logre *cercar* lo más posible al defecto. También se debe considerar que las pruebas deben ser repetibles, así luego de la corrección se puede probar si realmente se ha removido el defecto que causaba la falla.

Es imposible realizar pruebas exhaustivas y probar todas las posibles secuencias de ejecución. El tiempo requerido, incluso para un pequeño programa, es excesivo. Conociendo este problema Dijkstra expresó lo siguiente: *"El testing demuestra la presencia de defectos pero nunca su ausencia"*.

Para realizar el testing lo primero que se debe tener son los casos de prueba (test cases). Estos se conforman por los datos de entrada al programa (datos de prueba) y los resultados esperados luego de la ejecución del mismo. Luego se ejecuta el caso de prueba (se ejecuta el programa con los datos de entrada del caso) y se compara el resultado esperado con el obtenido en la ejecución. Si el resultado esperado es distinto al resultado obtenido pueden ocurrir dos cosas: se provocó una falla o el resultado esperado es incorrecto.

Debido a la imposibilidad de realizar pruebas exhaustivas es que se necesitan estrategias para seleccionar casos de prueba significativos. Un caso de prueba es significativo cuando tiene una alta probabilidad de provocar una falla. Un test set (conjunto de casos de prueba) es de significancia cuando tiene un alto potencial para descubrir defectos; los casos incluidos en el conjunto son significativos y las fallas que provocan son causadas por variada cantidad de defectos. Entonces, la ejecución correcta de esta clase de test set aumenta la confianza en el producto que se prueba. Por lo tanto, la meta no es ejecutar una gran cantidad de casos de prueba, sino ejecutar un número suficiente de casos de prueba significativos.

La forma general de definir test set de significancia es agrupar elementos del dominio de la entrada del software en clases D_i, tal que, elementos de la misma clase se comportan, o se supone que se comportan, exactamente de la misma manera. Además, la unión de los D_i debe ser igual al dominio de entrada del software: $\bigcup_{all\ i} D_i = \mathrm{D}$. Entonces, se elige un único caso de prueba para cada clase D_i. Para las clases que no es seguro que su comportamiento sea el mismo, pero que tampoco se sepa cómo partir dichas clases en otras, se toma más de un caso de prueba para cada una.

Existen diversas clasificaciones o dimensiones del testing, a continuación se presentan alguna de ellas.

El testing se puede realizar sobre una porción de código ejecutable o el sistema entero. Dependiendo del nivel en el cual se realiza el testing tenemos una de las dimensiones del mismo. La clasificación que usaremos es la más estándar:

Prueba unitaria. Son las pruebas que se realizan en pequeñas porciones individuales de código. Estas porciones de código son componentes individuales normalmente

construidas por un único programador. En un paradigma orientado a objetos estas pruebas comprenden las pruebas de métodos de un objeto, las pruebas de clases y las pruebas de pequeños conjuntos de clases (class clusters).

Prueba de integración. Cuando los componentes individuales están probados y se van a integrar se realizan las pruebas de integración. Estas buscan encontrar fallas en las interacciones entre los distintos componentes.

Prueba del sistema. Estas son las pruebas que se realizan sobre el sistema completo. Estas pruebas comprenden tanto pruebas funcionales como no funcionales del sistema.

Las pruebas se pueden realizar con o sin conocimiento del código que se va a poner a prueba. A partir de esto surge otra clasificación:

Prueba de caja negra. Usando esta técnica los casos de prueba se derivan a partir de especificaciones del sistema o componente. Este tipo de pruebas ve al sistema (o componente) como una caja negra del cual se tiene su especificación, por lo que se conoce su comportamiento a nivel de entradas y salidas, pero no se conoce su implementación.

Prueba de caja blanca. Usando esta técnica los casos de prueba se derivan a partir de la estructura o implementación de la componente. En este tipo de pruebas se conoce el "interior" de la componente a ser testeada por lo que se las llama pruebas de caja blanca, caja de cristal o caja transparente. Debido al uso de la estructura de la componente a ser testeada también son llamadas pruebas estructurales. Normalmente estas pruebas son usadas en pequeñas porciones de código.

Las pruebas no sólo son usadas para probar los aspectos funcionales de un sistema (o componente) sino que también sirven para probar aspectos no funcionales del mismo. A partir de esto se tiene la siguiente clasificación:

Pruebas funcionales. Estas pruebas buscan mostrar que el sistema (o componente) se comporta correctamente respecto a las funciones que el mismo debe brindar.

Pruebas no funcionales. Estas pruebas buscan mostrar que el sistema (o componente) se desempeña según lo especificado o requerido. Se las llama también pruebas de desempeño o pruebas de performance.

De acuerdo a lo que se busca realizar con el testing este se puede dividir en pruebas de defectos y pruebas estadísticas, dando otra forma de clasificación:

Pruebas de defectos. Estas buscan encontrar las diferencias entre un sistema (o componente) y su especificación (funcional y no funcional). Las pruebas se diseñan para intentar revelar defectos existentes en el sistema.

Pruebas estadísticas. Estas pruebas son usadas para comprobar cómo funciona el sistema en condiciones operacionales. Las pruebas se diseñan para simular las entradas comunes de los usuarios y su frecuencia. Estas sirven para estimar la fiabilidad del sistema bajo uso operacional y para valorar el desempeño del mismo. También se llaman pruebas operacionales.

Estas clasificaciones son ortogonales. De todas formas algunas pruebas que surgirían de realizar el producto cartesiano entre ellas no son razonables. Por ejemplo, no parece razonable realizar una prueba unitaria de caja blanca que sea de desempeño y operacional.

Es oportuno realizar un pequeño punteo sobre los pros y los contras de las técnicas estáticas y de las dinámicas. De este se desprende que ambas técnicas son complementarias. Las técnicas estáticas:

- Son efectivas en la detección temprana de defectos.
- Sirven para verificar cualquier producto (requerimientos, diseño, código, casos de prueba, etc.).
- Proveen conclusiones de validez general.
- Están sujetas a los errores de nuestro razonamiento.
- Normalmente se basan en un modelo del producto y no en el producto.
- Raramente se usan para la validación.
- Siempre dependen de una especificación.
- No consideran el hardware o el software de base.

Por otro lado las técnicas dinámicas:

- Consideran el ambiente dónde es usado el software.
- Sirven tanto para verificar como para validar.
- Sirven para probar otras características además de la funcionalidad.
- Están atadas al contexto donde se ejecutan las pruebas.
- No son generales. Sólo se aseguran los casos probados.
- Solamente sirven para probar el software construido.
- Normalmente detectan un único defecto por prueba. Los defectos cubren a otros defectos o el software colapsa.

En las siguientes secciones se presentan distintos tópicos del testing. La discusión de la aplicación del testing dentro del desarrollo del grupo CSI se presenta en cada una de las siguientes subsecciones.

3.5. Técnicas de Caja Negra

Las técnicas de caja negra son aquellas en las que los casos de prueba se generan únicamente a partir de la especificación de lo que esté bajo prueba sin contemplar su estructura y/o implementación. Debido a la forma de generación es normal que porciones enteras de código queden sin ejercitarse al ejecutar las pruebas.

Existen diversas técnicas conocidas de caja negra entre las que se encuentran:

- Particiones en clases de equivalencia.
- Valores límite.
- Testing basado en diagramas de estado.
- Testing basado en tablas de decisión.
- Grafos causa-efecto.

La técnica de partición en clases de equivalencia consiste en partir la entrada del programa en clases de equivalencia. Usando la especificación se divide las entradas en clases que se supone se comportan de igual forma en el programa. Para identificar las distintas clases de equivalencia se han propuesto distintos métodos. En todos estos es importante identificar tanto las clases válidas como las inválidas. Las válidas contienen datos de entrada válidos y esperados por el programa. Por otro lado, las clases inválidas contienen datos de prueba que el programa no debería aceptar.

Luego de identificadas las clases de equivalencia, tanto las válidas como las inválidas, se selecciona al menos un dato de prueba para cada una. Luego, recurriendo nuevamente a la especificación se obtiene el resultado esperado para cada dato de prueba seleccionado.

La técnica de valores límite toma valores en los límites de las clases de equivalencia como datos de prueba. La experiencia muestra que los casos de prueba que exploran las condiciones límite producen mejor resultado que aquellas que no lo hacen. Entonces, esta técnica complementa a la técnica de partición en clases de equivalencia. Trabajando con ambas a la vez se tendrán casos *típicos* para cada clase de equivalencia identificada así como también representantes de los valores límite de cada una.

El testing basado en diagramas de estado es aplicable cuando lo que se está probando es estado dependiente; cuando es una máquina de estado.

Una máquina de estado es un sistema en el cual las salidas están determinadas tanto por la entrada actual como por las pasadas. El efecto provocado por las entradas pasadas es representado por un estado.

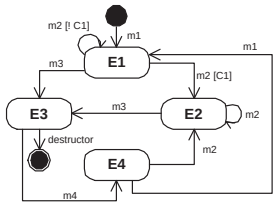
Existen distintas estrategias de testing a partir de un diagrama de estado.[10] Normalmente estas se basan en una forma de *cubrir* el diagrama. Por ejemplo, tener casos de prueba que logran ejecutar todas las transiciones entre estados.

[10] Aquí se entiende el diagrama de estados como la especificación funcional de un programa (sistema, unidad, clase o componente) que se comporta como una máquina de estado.

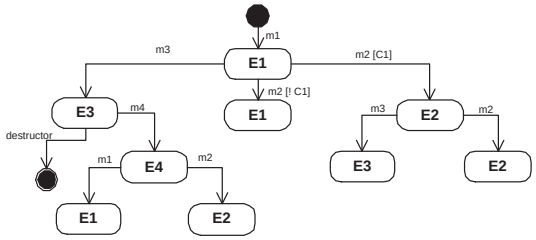
Una estrategia para derivar casos de prueba a partir de diagramas de estado es la estrategia N+ [Bin00]. La estrategia se divide en dos partes, All round-trip path y Sneak paths.

All round-trip path exige cubrir todos los caminos simples desde el estado inicial al final y todos los caminos que empiezan y terminan en un mismo estado. La estrategia construye un *árbol de transición* a partir del diagrama de estados. Luego, para cada camino que va desde el nodo inicial hasta una hoja se genera un caso de prueba.

Cada caso de prueba es una secuencia de ejecuciones de eventos. Para cada ejecución de un evento se tiene como dato de entrada el evento a ejecutar y una o ninguna condición. Como resultado esperado se tiene el estado que se alcanza al ejecutar ese evento con esa condición. El resultado esperado también lo puede componer la acción asociada a la transición. La Figura 2.6 muestra un diagrama de estados, uno de los posibles árboles de transición asociado y dos casos de prueba que se derivan del árbol. En el ejemplo de la Figura las transiciones no tienen acciones de forma de simplificar.



(a) Diagrama de Estado



(b) Árbol de Transición

	Entrada		**Resultado Esperado**	
Caso	**Evento**	**Condición**	**Estado**	**Acción**
1.1	m1		E1	
1.2	*	!C1	E1	
1.3	m2	C1	E2	
1.4	m4		E3	
2.1	m1		E1	
2.2	m3		E3	
2.3	des.		E. final	

(c) Casos de Prueba

Figura 2.6: Estrategia All-Round Trip Path

Sneak paths prueba los mensajes no esperados en un estado determinado. Para realizar este test se pone a lo que esté bajo prueba en un estado y se le envía un mensaje que no es esperado por ese estado. Esta estrategia obliga a probar todos los mensajes no esperados por los estados del diagrama de estado.

La técnica de generación de casos de prueba basada en tablas de decisión es indicada para implementaciones bajo test con las siguientes características:

- Las salidas son seleccionadas de acuerdo a casos distintos de las variables de entrada.

- Los casos pueden ser modelados por expresiones Booleanas mutuamente excluyentes en las variables de entrada.
- Los resultados a ser producidos por la implementación no dependen del orden en que se inicializan o se evalúan las variables de entrada.
- La respuesta no depende de las entradas o salidas anteriores.

Para realizar el testing usando tablas de decisión se realizan normalmente los siguientes pasos: Primero se modela la implementación con una tabla de decisión. Luego se elige una estrategia (cubrimiento) de generación de un test set. Por último se generan los casos de prueba a partir de la tabla y siguiendo la estrategia.

Las tablas de decisión se dividen en la *sección de condición* y en la *sección de acción*. La sección de condición presenta las condiciones, combinaciones de condiciones y variables de decisión. Estas variables son entradas o factores del sistema (ambiente). Una condición es una combinación de variables de decisión que debe ser reducible a verdadero o falso. La sección de acción lista las respuestas a ser producidas cuando la combinación correspondiente de condiciones evalúa como verdadero. Cada combinación única de condiciones y acciones es un *variante*.

A partir de la tabla de decisión se pueden generar distintos conjuntos de casos de prueba dependiendo de la estrategia de generación elegida. Algunas de estas estrategias son:

- Todos los variantes explícitos.
- Todos los variantes.
- Todos los verdaderos (All-true).
- Todos los falsos (All-false).
- Todos los primos.
- Cada condición/Todas las condiciones.
- Determinantes BDD.
- Negación de variable.
- Análisis de dominio de variables no binarias.

Todos los variantes explícitos genera casos de tal forma que todos los variantes que aparecen en la tabla de decisión son producidos al menos una vez. Esta es la estrategia más débil de generación de casos de prueba usando una tabla de decisión. El resto de las estrategias queda fuera del alcance de esta tesis.

Los grafos causa-efecto pueden ser vistos como una forma de identificar y analizar las relaciones a ser modeladas en una tabla de decisión. Estos grafos fueron propuestos por primera vez por Myers[Mye79].

Esta técnica ha demostrado ser efectiva tanto como para detectar inconsistencias en la especificación como para generar casos de prueba. La idea atrás de esta técnica es la de obtener las relaciones lógicas que existen entre las causas (variables de decisión) y los efectos (acciones). Cada causa es dibujada como un nodo de un grafo en una columna. Cada efecto es dibujado, también como un nodo de un grafo, en una columna enfrentada a la de las causas. Luego, se unen esos nodos mediante aristas que indican que la causa incide de alguna forma en la condición. Si un efecto tiene más de dos causas (aristas que llegan al mismo), se anota la relación lógica de las causas con símbolos lógicos entre las líneas que representan las aristas. Normalmente se generan nodos intermedios de forma de simplificar el grafo y obtener nodos a los cuales llegan dos aristas como máximo.

Existen diversas formas de generar los casos de prueba a partir del grafo causa-efecto. Una es pasar la representación del mismo a una tabla de decisión y luego aplicar las estrategias de generación ya presentadas. Otras formas quedan fuera del alcance de esta tesis.

Existen otras estrategias de testing que no son discutidas. Se considera que las mismas no son apropiadas para el desarrollo en el grupo CSI y que su presentación y posterior discusión no aporta a esta tesis.

Una línea de investigación muy activa en el testing de caja negra es la generación automática de casos de prueba a partir de distintos tipos de especificaciones [BDL+06, CCD+01, PPA05, Val05].

Discusión

El testing de caja negra es una herramienta fundamental dentro del Framework que se propone.

De las estrategias presentadas se propone el uso de partición en clases de equivalencia y valores límite para todos los niveles de testing; unitario, de integración y de sistema. Creemos que estas estrategias son fáciles de usar y que pueden mejorar la calidad de los productos desarrollados en el grupo CSI.

Estrategias como grafos causa-efecto o tablas de decisión son más complejas y necesitan de una mayor disciplina de la esperada en los grupos de desarrollo dentro de los proyectos en estudio. Ambas son dejadas fuera del Framework.

Las estrategias basadas en diagramas de estado son aplicables al testing del comportamiento de componentes o de clases que se comportan como máquinas de estado. El Framework sugiere el uso de esta estrategia para realizar el testing a nivel de clase y de componente.

3.6. Técnicas de Caja Blanca

Las técnicas de caja blanca son aquellas en las que los datos de prueba se generan únicamente a partir de la estructura y/o implementación de lo que se prueba. Luego,

usando los datos de prueba obtenidos y la especificación se determinan los resultados esperados, obteniendo los casos de prueba.

Debido a la forma de generar los casos de prueba puede ocurrir que algún requerimiento o funcionalidad sea pasado por alto sin ser probado.

Las técnicas de caja blanca pueden ser usadas en las pruebas unitarias. Su uso en las pruebas de integración es variado y no son usadas en las pruebas de sistema. Esto se debe a que los datos de prueba se derivan del código y se hace imposible esta derivación a nivel de sistema. Las pruebas unitarias, de integración y de sistema se presentan en la subsección 3.7 de este capítulo.

Las técnicas de caja blanca más conocidas son las siguientes:

- Técnicas basadas en el flujo de control.
- Técnicas basadas en el flujo de datos.
- Mutant testing (prueba de mutantes).

A continuación se presentan brevemente las tres técnicas.

Las técnicas basadas en el flujo de control buscan provocar fallas basándose en los posibles recorridos del grafo de flujo de control de un programa. Normalmente se trabaja de la siguiente manera para generar los casos de prueba:

1. Se elige un criterio de cubrimiento de flujo de control.
2. A partir del código (usando el grafo de flujo de control) se deriva un conjunto de datos de prueba de manera que la ejecución del programa con esos datos cumple con el criterio seleccionado.
3. Usando los datos de prueba y la especificación del programa se obtienen los resultados esperados, de esta manera se tiene el conjunto de casos de prueba que cumple con el criterio elegido.

Se presentan a continuación criterios de cubrimiento basados en el flujo de control:

Cubrimiento de sentencias. Cada sentencia dentro del código debe ser ejecutada al menos una vez.

Cubrimiento de decisión. Cada decisión dentro del código debe ser ejecutada al menos una vez con el valor true y otra vez con el valor false.

Cubrimiento de decisión/condición. Cada condición que compone una decisión en el programa debe ser ejecutada al menos una vez con el valor true y otra vez con el valor false. Además se debe cumplir con el criterio de decisión.

Criterio de cubrimiento de condición múltiple. Todas las combinaciones posibles de resultados de condición dentro de una decisión deben ser ejecutadas al menos una vez.

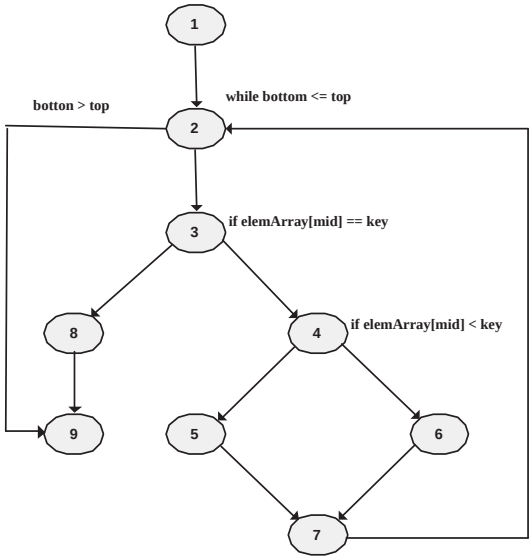
Criterio de cubrimiento de trayectorias independientes. Todas las trayectorias independientes del programa se ejecutan al menos una vez.

Criterio de cubrimiento de caminos. Todos los caminos se ejecutan al menos una vez.

La Figura 2.7 muestra un programa de búsqueda binaria y el grafo de flujo de control asociado. El programa recibe un arreglo de objetos ordenados y una clave. El mismo realiza una búsqueda y si encuentra la clave dentro de los elementos del arreglo devuelve el lugar dónde lo encontró (r.index = posición del elemento), y un valor booleano indicando que fue encontrado (r.found = true). En caso de no encontrarlo devuelve el valor booleano en false (r.found = false) y el índice con el valor -1 (r.index = -1). En el caso de la búsqueda binaria que se presenta todas las decisiones tienen una única condición. Esto ocasiona que los criterios de decisión, decisión/condición y condición múltiple se satisfagan con los mismos casos de prueba.

```
public void search (int key, int[] elemArray, Result r) {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;
        r.found = false; r.index = -1;
        while (bottom <= top) {
                mid = (top + bottom) / 2;
                if (elemArray[mid] = = key) {
                        r.index = mid;
                        r.found = true;
                        return;
                }
                else {
                        if (elemArray[mid] < key)
                                bottom = mid + 1;
                        else
                                top = mid - 1;
                }
        }
}
```

(a) Programa Java



(b) Grafo de Flujo de Control

Figura 2.7: Ejemplo Búsqueda Binaria

La Figura 2.8 presenta dos casos de prueba y las ramas que son ejecutadas en el grafo de flujo de control. El conjunto de casos de prueba, compuesto por estos dos casos de prueba, cumple con el criterio de decisión. Como ya se mencionó, los datos de prueba se obtienen usando el grafo de flujo de control y el código. Luego, se determina el resultado esperado *aplicando* la especificación a los datos de prueba obtenidos.

Dentro de las estrategias de caja blanca también se pueden estudiar los valores límite. En estos casos los valores límite son los que están en los bordes de las decisiones y/o condiciones.

Las técnicas basadas en el flujo de datos usan las declaraciones de los datos, datos *asesinados* (killed) y usos de los datos. Los objetos de datos pueden ser:

Datos de Prueba
key = 3
elemArray = [1, 5, 7]

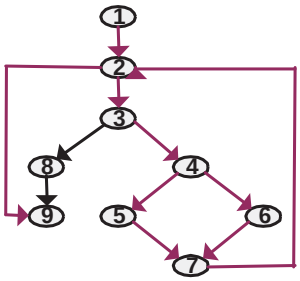
Resultado Esperado
index = -1
found = false

(a) Caso de Prueba 1

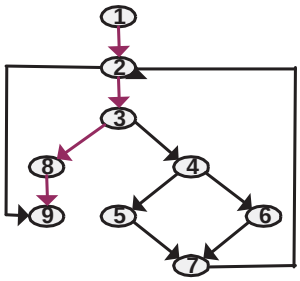
Datos de Prueba
key = 5
elemArray = [1, 5, 7]

Resultado Esperado
index = 1
found = true

(b) Caso de Prueba 2



(c) Cobertura del Caso 1



(d) Cobertura del Caso 2

Figura 2.8: Casos y Cobertura para la Búsqueda Binaria

- Definidos, creados o inicializados.
- Asesinados, no-definidos o liberados.
- Usados en un cálculo.
- Usados en un predicado.

Los criterios de cubrimiento usados con esta técnica se basan en ejecutar casos que lleven a un objeto de datos desde una situación a otra. Por ejemplo, desde su declaración a todos los lugares donde es usado sin ser vuelto a declarar antes. La forma de selección de casos es igual que en flujo de control. Primero se elige un cubrimiento de flujos de datos, luego se deriva un conjunto de datos de prueba que cumple con el criterio y por último se asocia a cada dato de prueba su resultado esperado según la especificación.

Los criterios de cubrimiento de flujo de datos son normalmente más finos que los de flujo de control. Sin embargo, es más complejo y costoso seleccionar los datos de prueba que cumplan con los cubrimientos de flujos de datos. Debido a la dificultad de generar los casos de prueba este tipo de testing es menos usado que el testing basado en el flujo de control.

La técnica de testing de mutantes se basa en la creación de mutantes del programa original y en el intento de distinguir a estos respecto del mismo. Los mutantes son programas similares al original y sólo se diferencian en un único cambio. La idea es generar un conjunto de programas mutantes a partir del original. Luego, para cada mutante se busca el lugar de modificación y se eligen datos de entrada de tal forma que al ejecutar el mutante con esos datos el resultado producido es diferente al obtenido al ejecutar el programa original con esos mismos datos. Si realmente al ejecutar ambos programas ocurre esto se dice que el mutante está muerto. El objetivo es matar a todos los mutantes. A

medida que se eliminan mutantes se van detectando fallas en el programa original. Estas fallas deben ser investigadas y corregidos los defectos que las provocan.

Existen herramientas que apoyan a esta técnica. Normalmente estas herramientas generan de forma automática los mutantes, ejecutan los casos de prueba sobre el programa original y los mutantes e informan cuáles mutantes sobrevivieron. Los mutantes que sobreviven son aquellos que para los casos de prueba ejecutados se comportan igual que el programa original.

Un problema que surge con esta técnica es la existencia de los mutantes equivalentes. Un mutante es llamado equivalente cuando su comportamiento es igual al del programa original en todas sus posibles ejecuciones. Estos mutantes no tienen forma de ser eliminados. El problema existente con estos mutantes es que no todos se pueden detectar de forma automática y el trabajo manual para detectarlos tiende a consumir mucho tiempo. En la Figura 2.9 se presentan dos mutantes del programa de búsqueda binaria de la Figura 2.7. Se marca con color la sentencia donde está la mutación de cada uno. La parte (a) de la Figura muestra un mutante equivalente. La parte (b) de la Figura muestra un mutante que no es equivalente y que además no es matado con los casos de prueba propuestos en la Figura 2.8.

```
public void search (int key, int[] elemArray, Result r) {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;
        r.found = false; r.index = -1;
        while (bottom <= top) {
                mid = (top + bottom) / 2;
                if (elemArray[mid] = = key) {
                        r.index = mid;
                        r.found = true;
                        return;
                }
                else {
                        if (elemArray[mid] <= key)
                                bottom = mid + 1;
                        else
                                top = mid - 1;
                }
        }
}
```

(a) Mutante Equivalente

```
public void search (int key, int[] elemArray, Result r) {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;
        r.found = false; r.index = -1;
        while (bottom <= top) {
                mid = (top + bottom) / 2;
                if (elemArray[mid] = = key) {
                        r.index = mid;
                        r.found = true;
                        return;
                }
                else {
                        if (elemArray[mid] < key)
                                bottom = mid + 2;
                        else
                                top = mid - 1;
                }
        }
}
```

(b) Mutante no Equivalente

Figura 2.9: Mutantes del Programa de Búsqueda Binaria

Hay algunas líneas de investigación activas dentro del testing de caja blanca. Se mencionan dos de ellas. Una es la generación de casos de prueba de forma automática a partir del código fuente [DBT01]. Otra es la detección automática de mutantes equivalentes [AHH04, OP97].

Discusión

Las técnicas de caja blanca y caja negra se complementan. Si ambas técnicas se combinan se logra aumentar la cantidad de fallas que se provocan al ejecutar los casos de

prueba. Por esto se incluyen también en el Framework propuesto.

Dentro de las técnicas de caja blanca se propone el uso del cubrimiento de decisión a nivel de métodos. Se entiende que mayores cubrimientos dentro de las técnicas de flujo de control no van a ser seguidos por los desarrolladores dentro del contexto en estudio. Además, entendemos que un mayor cubrimiento perjudicaría la productividad y empeoraría la relación costo/beneficio.

Las técnicas basadas en flujos de datos y el testing de mutantes no son considerados en el Framework debido a dos factores. Primero, los desarrolladores no están entrenados en ninguna de las dos estrategias. Segundo, el tiempo que requiere realizar cualquiera de los dos tipos de testing es excesivo para los proyectos que se consideran.

3.7. Testing Unitario, de Integración y de Sistema

Como ya se mencionó las pruebas se pueden dividir por nivel. Los niveles que consideramos son unitario, de integración y de sistema.

Las pruebas unitarias se realizan en pequeñas porciones individuales de código. Estas porciones de código son componentes individuales normalmente construidas por un único programador. En un paradigma orientado a objetos comprenden las pruebas de métodos de un objeto, las pruebas de clases y las pruebas de pequeños conjuntos de clases (class clusters).

Normalmente sucede que una unidad *usa* o es *usada* por otras unidades. En estos casos se debe contar con stubs y/o drives. Se considera, para simplificar, que una unidad A usa a otra unidad B cuando la unidad A invoca de alguna manera a la unidad B. Una unidad es usada por otra cuando se da el caso inverso.

Al momento de realizar el testing de una unidad A puede acontecer que las unidades que usan o son usadas por A no estén disponibles. Entonces, para realizar esta prueba se crea un driver y varios stubs. El driver es el que invoca a la unidad A y es quien va a proveer los casos de prueba para A. Se creará además un stub por cada unidad que use A y no esté disponible. El stub simula, en forma simplificada, el comportamiento de la unidad que está reemplazando. La Figura 2.10 muestra una unidad A que debe ser testeada y que usa a las unidades B y C. Además A es usada por D. Para testear A, tanto sea porque B, C y/o D no están disponibles o porque se quiere testear A de forma controlada y aislada se generan los stubs SB y SC y el driver DD. SB y SC son stubs de B y C respectivamente. DD es el driver de D.

Las técnicas de testing unitario pueden ser tanto de caja negra como de caja blanca.

Las unidades dentro del desarrollo orientado a objetos corresponden a las clases, las agregaciones de clases (class-clusters) y las componentes pequeñas. Los class-cluster son un conjunto pequeño de clases que interactúan entre si y que tienen una sola entrada desde el exterior del conjunto [Bin00].

Las pruebas de integración se realizan cuando las unidades están probadas y se quieren integrar. Como ya fueron realizadas las pruebas unitarias se considera que las unidades

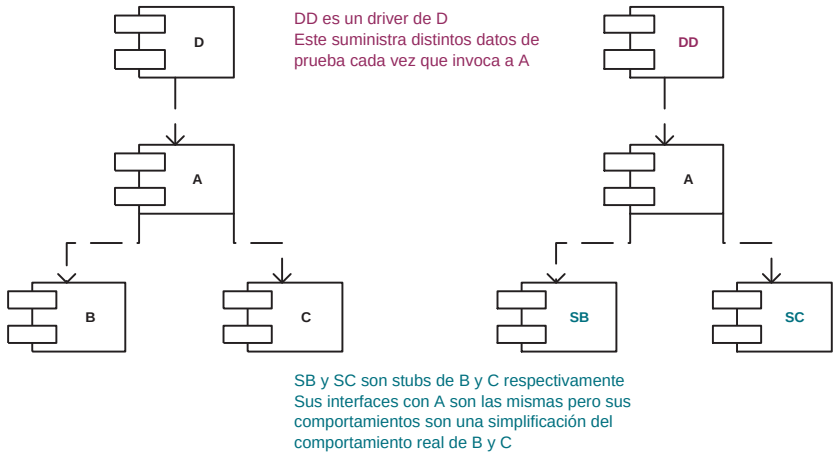


Figura 2.10: Prueba de Módulos Mediante Drivers y Stubs

funcionan adecuadamente de forma aislada. Entonces, las pruebas de integración buscan encontrar fallas en las interacciones entre las distintas unidades y comprobar si las mismas trabajan adecuadamente como un sistema integrado.

Existen varias estrategias de integración y estas normalmente no son usadas en su forma pura. Las tres estrategias más conocidas son Big-Bang, Bottom-up y Top-down. Para ejemplificar se toma un sistema en el cual las unidades son sólo usadas por una única unidad. La Figura 2.11 muestra este sistema con sus unidades y las formas en las cuales se pueden integrar usando las técnicas mencionadas. El énfasis de la prueba en cada caso está marcado con color. Este puede ser sobre unidades y/o interacciones entre las mismas.

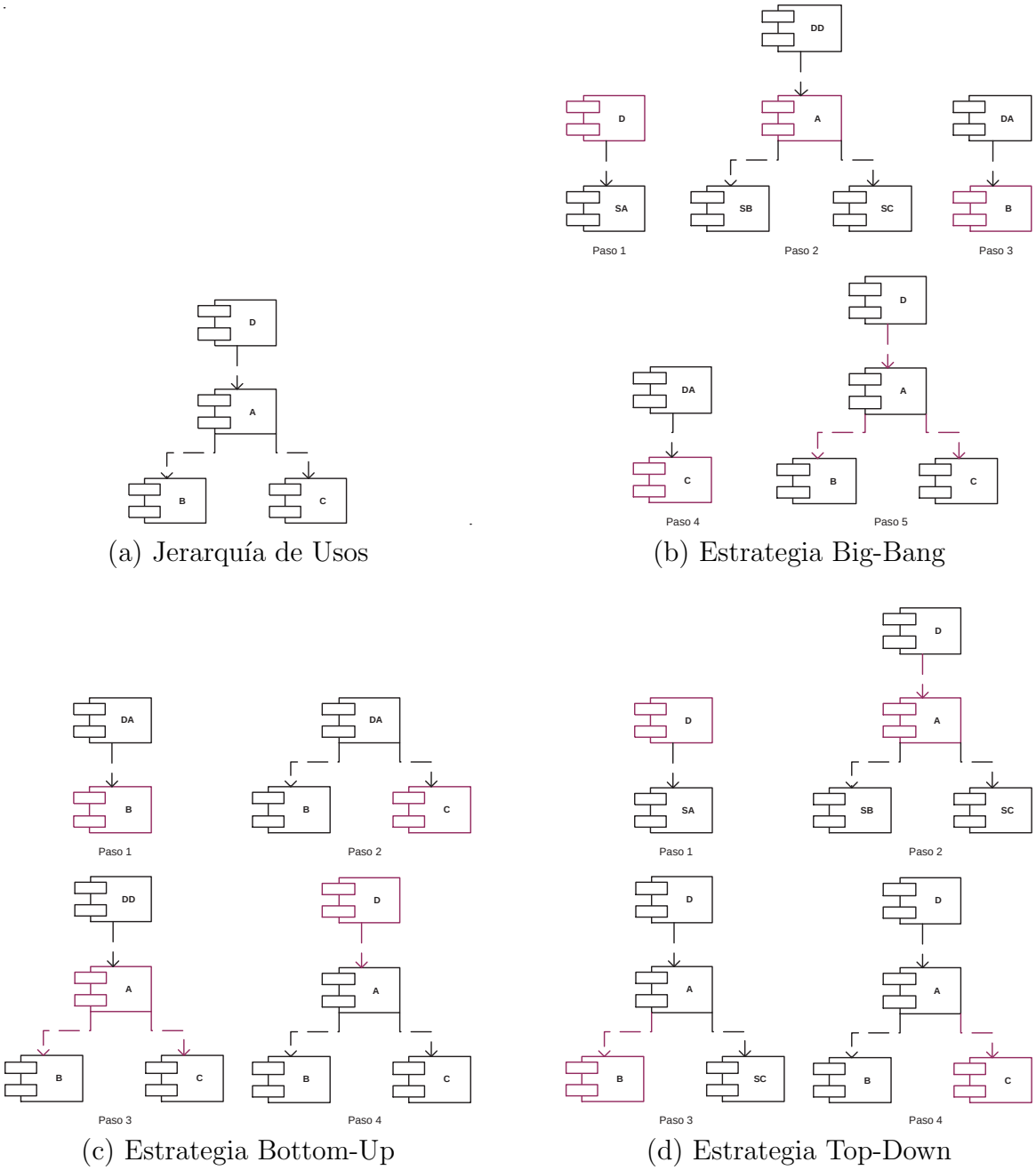
La estrategia Big-Bang prueba cada unidad de forma aislada y luego integra todas las unidades a la vez. Debido a esto se deben implementar stubs y drives para cada una de las unidades del sistema. Al integrar todos los módulos a la vez resulta difícil saber cuál es el origen de las fallas que se producen durante la integración. La Figura 2.11(b) muestra esta integración.

La estrategia Bottom-up comienza por las unidades que no usan a ninguna otra unidad y va *subiendo* en la jerarquía de usos. Esta estrategia requiere de drivers pero no de stubs. Esto se debe a que en el momento de probar una unidad todas las unidades que pueden ser usadas por la que está bajo prueba ya fueron probadas. La Figura 2.11(c) muestra una integración siguiendo Bottom-up.

Por último, la estrategia Top-down empieza por los módulos superiores de la jerarquía de usos y va *bajando* por la misma. Esta estrategia requiere de stubs pero no necesita drivers. Cada vez que se prueba una unidad la unidad que la invoca ya fue probada antes. La Figura 2.11(d) muestra una integración siguiendo Top-down.

Las técnicas de testing de integración pueden ser tanto de caja negra como de caja blanca. Al usar técnicas de caja blanca se busca cubrir las distintas formas en las cuales se comunican las unidades.

Las pruebas del sistema son las que se realizan sobre el sistema finalizado. Estas prue-



(a) Jerarquía de Usos

(b) Estrategia Big-Bang

(c) Estrategia Bottom-Up

(d) Estrategia Top-Down

Figura 2.11: Prueba de Integración

bas comprenden tanto pruebas funcionales como no funcionales del sistema. Las pruebas del sistema son normalmente de caja negra. Algunas pruebas no funcionales de sistema se presentan en la subsección siguiente.

La idea general de las pruebas funcionales de sistema es probar cada funcionalidad del mismo de forma individual y luego probar distintos ciclos de funcionalidades. Los ciclos de funcionalidades normalmente están comprendidos por ciclos de vida de entidades y ciclos del negocio o procesos de la organización.

Tanto las pruebas individuales de funcionalidades como las pruebas de ciclos pueden estar basadas en casos de uso. Formas de testing basado en casos de uso se presentan en [Bin00, Heu01].

Discusión

Las pruebas unitarias ya fueron mencionadas indirectamente en el estudio de las técnicas de caja blanca y negra. Las técnicas de caja negra son útiles para realizar pruebas de sistema y las técnicas basadas en casos de uso son útiles para sistemas en los cuales se usan casos de uso para especificarlos. Tanto las pruebas unitarias como de sistemas son parte del Framework.

Las pruebas de integración no forman parte del Framework explícitamente. Sin embargo, se espera que los grupos de desarrollo realicen integraciones controladas donde se prueba la interacción entre las unidades.

3.8. Testing de Performance

En secciones anteriores se ha tratado el testing funcional. En definitiva, existe algo (programa, clase, componente, sistema), que tiene cierta funcionalidad que debe ser probada. Normalmente estas funcionalidades tienen restricciones. A las restricciones de las funcionalidades se las llama requerimientos no funcionales. Al testing de estos requerimientos se le conoce por testing de performance o de desempeño.

En esta sección se presentan algunos tipos de pruebas de performance. Sin embargo, no se tratan técnicas ni procedimientos para realizar estas pruebas porque en estos casos las técnicas de testing son altamente dependientes del sistema a testear.

La nomenclatura alrededor de estos tipos de pruebas es variada. Se usa la nomenclatura de Myers [Mye79] y se presentan solo algunas pocas de las pruebas que caen dentro de esta categoría.

Prueba de estrés (esfuerzo). Esta prueba implica someter al sistema a grandes cargas o esfuerzos considerables. Un esfuerzo grande es un pico de volúmenes de datos, normalmente por encima de los límites especificados, en un corto período de tiempo. No solo se debe considerar volúmenes de datos sino también de usuarios, dispositivos, etc. Una analogía con un dactilógrafo nos indica que este tipo de prueba se corresponde a probar si puede escribir a razón de 50 palabras por minuto.

Prueba de volumen. Esta prueba implica someter al sistema a grandes volúmenes de datos. El propósito es conocer si el sistema puede manejar el volumen de datos especificado. En la analogía del dactilógrafo esto se corresponde a conocer si el mismo puede escribir un documento enorme.

Prueba de facilidad de uso. Esta prueba trata de encontrar problemas en la facilidad de uso. A modo de ejemplo y para fijar ideas se brindan algunas consideraciones a tener en cuenta:

- ¿Ha sido ajustada cada interfaz de usuario a la inteligencia y nivel educacional del usuario final y a las presiones del ambiente sobre él ejercidas?
- ¿Son las salidas del programa significativas, no-abusivas y desprovistas de la "jerga de las computadoras"?
- ¿Son directos los mensajes de error o requieren de un doctor en ciencias de la computación?

Prueba de seguridad. Con este tipo de pruebas se busca testear la seguridad del sistema. Se diseñan casos de prueba que intentan burlar los controles de seguridad del sistema bajo test.

Prueba de rendimiento. Estas pruebas buscan comprobar si el rendimiento del sistema es el especificado. Requerimientos de rendimiento pueden ser aquellos vinculados con los tiempos de respuesta de sistemas interactivos o sistemas de tiempo real y con tiempos de finalización de tareas. Normalmente estos requerimientos se especifican bajo ciertas condiciones de carga y cierta configuración del sistema.

Prueba de configuración. Mediante este tipo de pruebas se busca conocer el comportamiento del sistema en distintas configuraciones de hardware y de software. Normalmente se busca conocer las distintas configuraciones mínimas del sistema.

Prueba de recuperación. Con estas pruebas se intenta conocer el comportamiento del sistema luego de una caída. Para lograr esto se simulan o crean fallas y luego se prueba la recuperación del sistema.

Discusión

Si bien este tipo de testing es necesario en la industria entendemos que en la gran mayoría de los desarrollos académicos de software este no se necesita. Al menos, no es necesario desde un punto de vista formal de diseño de casos de prueba. Como máximo se prueba, muy informalmente, si el sistema soporta una cierta cantidad de usuarios o conexiones.

Entendemos que los problemas de calidad hay que atacarlos gradualmente y que no tiene sentido probar performance de un sistema si las funcionalidades deseadas no funcionan como se espera. Debido a la realidad de los desarrollos actuales del CSI se entiende que es más importante contar con pruebas funcionales que con pruebas de desempeño. Por este motivo el Framework no propone realizar este tipo de pruebas.

3.9. ¿Cuándo Finalizar el Testing?

El testing es provechoso cuando se logra provocar fallas, por lo que genera dudas sobre cuándo se debe detener el mismo. En términos generales, si se detiene el testing cuando se dejaron de provocar fallas se está parando el mismo cuando este no es efectivo;[11] por lo tanto, se detiene el testing cuando este no se está realizando de forma correcta. Por otro lado, si se detiene el testing cuando se provocan *muchas* fallas entonces el producto final entregado es un producto defectuoso. Por lo tanto, ¿existe un punto óptimo entre no provocar fallas y provocar muchas fallas en el cual detener el testing?

Existen distintas estrategias para determinar el momento de finalización de las pruebas. Las estrategias que se presentan terminan el testing cuando:

- el tiempo asignado para el mismo ha expirado.
- todos los casos de prueba se han ejecutado sin detectar fallas.
- se ha alcanzado un cierto nivel de cubrimiento.
- la cantidad de defectos remanentes es menor que un número dado.
- la efectividad de la prueba ha disminuido hasta un valor predefinido.
- la confiabilidad ha alcanzado un límite dado.
- el aumento en la confiabilidad no justifica los costos del testing.

Estas reglas para detener el testing son presentadas brevemente a continuación.

La presentación de los cinco primeros criterios es extraída de Myers [Mye79].

El primer criterio, si bien es usado, es inútil. El tiempo para el testing puede haber terminado sin haber logrado hacer nada. Además, este criterio no mide ni la calidad de las pruebas ni la calidad del producto.

Terminar las pruebas en base a la ejecución sin fallas de todos los casos de prueba también es inútil ya que tampoco mide la calidad de los mismos. Los casos de prueba pueden ser *malos*; no tienen una alta probabilidad de provocar fallas. También puede ocurrir que los casos de prueba se hayan agotado. Los casos de prueba se corren y se provocan fallas. Luego se depura el software para eliminar el defecto que causa la falla y se vuelven a correr los casos de prueba. Esta sucesión de corridas de casos de prueba y depuraciones del software lleva a agotar los casos de prueba.

Detener la prueba al lograr un cierto cubrimiento de código es apropiado a nivel unitario pero es poco apropiado a nivel de sistema. A nivel unitario se pueden establecer criterios de cubrimiento y estos se pueden alcanzar. Por lo tanto, es posible realizar casos de prueba que cumplan con el criterio y remover todos los defectos hasta que estos casos ejecuten correctamente. Sin embargo, a nivel de sistemas es difícil cumplir con un criterio

[11] A menos que el software ya no tenga más defectos. Esto es poco probable y, suponerlo, muy peligroso.

de cubrimiento de código. De todas formas se pueden establecer criterios mínimos de cubrimiento y usarlos como criterios necesarios pero no suficientes. Por ejemplo, se puede pedir que el 70 % de las instrucciones del software sea cubierta al ejecutar el conjunto de casos de prueba del sistema.

El cuarto criterio propone finalizar el testing en base a los defectos remanentes. Por ejemplo, detectar el 90 % de los defectos del software en el testing de sistema, o dejar el sistema con 3 defectos.

Para aplicar este tipo de criterios es necesario estimar los defectos remanentes del sistema. Se presentan brevemente dos métodos para estimar los defectos remanentes: siembra de defectos y pruebas independientes.

En la siembra de defectos se introducen defectos en el código. Luego se cuentan los defectos encontrados durante las pruebas y se estiman los defectos remanentes. Para la estimación se usa la siguiente relación:

$$\frac{\text{Defectos sembrados detectados}}{\text{Total de defectos sembrados}} = \frac{\text{Defectos no sembrados detectados}}{\text{Total de defectos no sembrados}}$$

La cantidad de defectos sembrados totales y detectados son conocidos. La cantidad de defectos no sembrados detectados también es conocida. Despejando la cantidad de defectos totales no sembrados se obtiene una estimación de la cantidad de defectos remanentes en el sistema. Este método asume que los defectos sembrados son representativos de los defectos reales, tanto en tipo como en complejidad.

En las pruebas independientes se tienen dos grupos que testean el mismo software y a partir de los defectos que ambos grupos detectan se estiman los defectos remanentes. Consideremos que D_1 es la cantidad de defectos encontrada por uno de los grupos, D_2 es la encontrada por el otro grupo y D_c son los defectos encontrados en común. Se tiene entonces que $D_c \leq D_1$ y $D_c \leq D_2$. Se define la efectividad como la cantidad de defectos encontrados sobre la cantidad de defectos totales. D_T es la cantidad de defectos totales que tiene el software. Entonces, la efectividad de uno de los grupos es $\frac{D_1}{D_T}$ y la del otro es $\frac{D_2}{D_T}$. Se toma como hipótesis que la efectividad es igual para cualquier parte del programa, entonces el primer grupo encontró D_c de los D_2 defectos encontrados por el otro grupo. Por lo tanto, la efectividad del primer grupo también vale $\frac{D_c}{D_2}$. Igualando las ecuaciones y despejando los defectos totales obtenemos una estimación de los defectos remanentes:

$$D_T = \frac{D_1 * D_2}{D_c}$$

En los casos que se considera la efectividad de las pruebas se detienen las mismas cuando es poco probable encontrar nuevos defectos en un lapso de tiempo. Para estos casos se registra el número de fallas provocadas por unidad de tiempo. Cuando este número cae más allá de cierto valor definido se detienen las pruebas. Hay que tener especial cuidado de no haber disminuido la cantidad de casos de prueba que se ejecutan o haberlos *agotado* y por eso no detectar nuevas fallas.

Las propuestas basadas en la confiabilidad usan la predicción de la tasa de fallas remanentes en el sistema para decidir si parar el testing. Para las propuestas en las cuales

se mide el aumento de confiabilidad y el costo para decidir si se justifica o no lograr ese aumento se necesita tener un costo monetario equivalente a lo que implica encontrar la falla por un usuario. Este segundo modelo no se estudia en esta tesis.

La predicción de la confiabilidad se basa normalmente en datos históricos del testing del software. Se usan modelos para predecir la confiabilidad a futuro y cada uno necesita una cierta cantidad de datos del pasado para los parámetros del modelo. Algunos de estos modelos son: modelo de Goel-Okumoto, modelo básico de Musa, modelo de Littlewood-Verall y modelo de Poisson logarítmico.[12] Yang y Chao comparan todos estos modelos y dos propuestos por ellos [YC95]. Un estudio en profundidad de la confiabilidad y de los modelos de confiabilidad queda fuera del alcance de esta tesis.

Discusión

En el Framework se propone la finalización del testing unitario al cumplir con el criterio de cubrimiento establecido para el proyecto. Como mínimo se pide que se use el cubrimiento de decisión.

Las técnicas más útiles para decidir cuándo detener las pruebas a nivel de sistema no son aplicables al desarrollo dentro del grupo CSI. El cálculo de defectos remanentes mediante inyección de defectos o grupos independientes es inviable. La inyección de defectos requiere conocer el tipo de defectos en los que se incurre normalmente y este dato no se tiene. Para realizar testing con grupos independientes se tiene que tener al menos dos grupos testeando el mismo software. En el contexto planteado los grupos de desarrollo son a lo sumo de cuatro personas, y creemos que si se realiza este tipo de estimación de defectos remanentes la pérdida en productividad va a ser considerable y la ganancia en calidad va a ser mínima.

Detener el testing basado en la disminución de fallas producidas por unidad de tiempo es la que entendemos más adecuada para el desarrollo académico de software. Sin embargo, para aplicar este enfoque se necesita una disciplina que no es habitual en los grupos de desarrollo que estamos estudiando.

Los métodos basados en la confiabilidad del software requieren conocimientos previos y datos históricos lo cual hace que también se descarte esta propuesta. Los grupos de desarrollo no tienen los conocimientos necesarios sobre la confiabilidad del software y sus modelos de predicción. Por otro lado, al igual que el enfoque anterior, el recolectar datos históricos del testing requiere mucha disciplina. Además, para aplicar los modelos de forma efectiva se necesitan varias instancias de la prueba del sistema, probablemente más instancias de las que realice cualquiera de los grupos de desarrollo.

Por lo mencionado en los párrafos anteriores es que se deja fuera del Framework una propuesta acerca de cuándo detener el testing a nivel de sistema.

[12] Estos modelos no tienen citas bibliográficas ya que no fueron estudiados en los artículos donde se presentan.

3.10. Otros Tipos de Pruebas

Existen otros tipos de pruebas que no fueron mencionadas en las secciones anteriores. Para algunas de estas se realiza una muy breve descripción en esta sección.

Pruebas de aceptación. Estas son las pruebas que realiza/n el/los cliente/s para conocer si el sistema funciona de acuerdo a sus requerimientos y necesidades. Estas pruebas se pueden hacer de diversas maneras y su formalidad suele estar ligada al tipo de producto.

Pruebas alfa. Este tipo de pruebas se realiza cuando el producto es para múltiples clientes.[13] Las pruebas alfa son realizadas en el ambiente de desarrollo por personas de la empresa de desarrollo que conocen el sistema a probar desde una visión de usuario.

Pruebas beta. Al igual que las pruebas alfa estas pruebas se llevan a cabo cuando se desarrolla para múltiples clientes. Normalmente estas tienen lugar luego de las alfa. Estas las realizan clientes elegidos sobre el producto final.

Pruebas en paralelo. Las pruebas en paralelo se usan cuando un sistema *nuevo* sustituye a otro *viejo*. Durante estas pruebas ambos sistemas funcionan a la vez. Se comparan los resultados de ambos sistemas para conocer si el sistema nuevo funciona adecuadamente.

Pruebas piloto. El sistema ya finalizado se pone a funcionar en producción pero de forma localizada. Con esto se logra testear el sistema en el propio ambiente de producción disminuyendo el impacto causado por las fallas.

Pruebas estadísticas. Son pruebas que se realizan para probar el desempeño y la fiabilidad del sistema bajo condiciones operacionales. Se tiene que tener o construir un perfil operacional para poder realizarlas. De este perfil se deriva, entre otras cosas, la probabilidad de cada entrada al sistema.

Pruebas de regresión. Las pruebas de regresión son las pruebas que se vuelven a ejecutar luego de modificaciones al programa. La idea es probar que lo que antes funcionaba sigue funcionando luego de las modificaciones. Estas pruebas se realizan en todos los niveles del testing; unitario, integración y sistema.

Para cada uno de estos tipos de pruebas existen algunas líneas de investigación latentes. Ese estudio queda fuera del alcance de este trabajo.

[13] Este tipo de producto es un producto de "mercado". Ejemplos de estos productos son: procesadores de texto, sistemas operativos y sistemas EPR.

Discusión

Las pruebas de aceptación son una parte fundamental del Framework. Con este tipo de pruebas buscamos que el equipo de desarrollo se comprometa a conseguir un producto de calidad y que el cliente se comprometa con el trabajo que está proponiendo.

Las pruebas de regresión se proponen en el Framework. Entendemos que este tipo de pruebas sirve para lograr varios de los aspectos de calidad que busca el grupo CSI. Estas no solo tienden a disminuir la cantidad de fallas en un sistema y a estabilizar las funcionalidades sino que ayudan en el mantenimiento del software. Además, las propias pruebas sirven de especificación de partes del sistema y ayudan a comprenderlo, mejorando de esta manera la modificabilidad y la extensibilidad del mismo. En el Framework se sugiere realizar estas pruebas en todos los niveles del testing.

3.11. Planificación de la V&V

El modelo "V" es una forma de presentar el momento en el cual crear los casos de prueba y el momento de ejecutarlos. La Figura 2.12 muestra este modelo. Esta muestra que

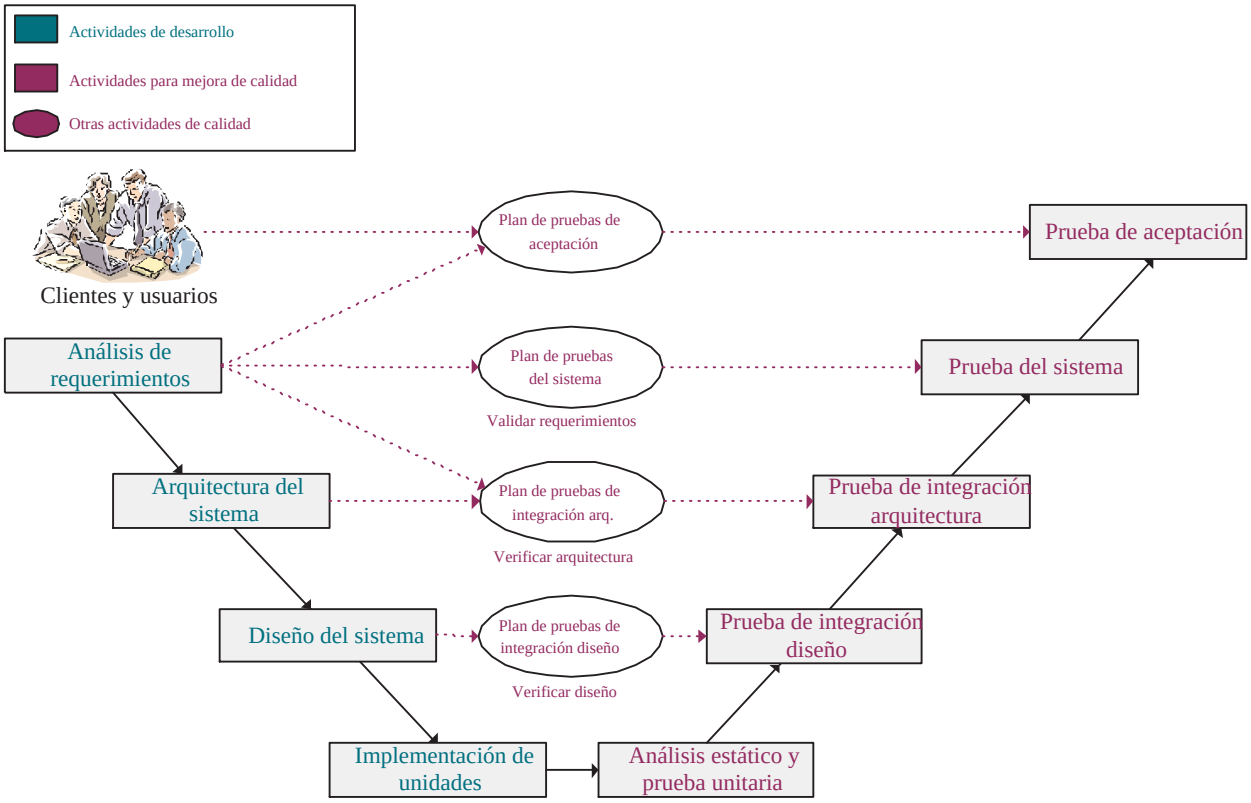


Figura 2.12: Modelo V

cuando se tienen los requerimientos se pueden crear los casos de prueba del sistema, cuando se tiene la arquitectura y el diseño se pueden crear todos los casos de prueba de integración

y que, en el momento de codificar se realiza el análisis de código y la prueba unitaria. Finalizada la etapa de construcción, y teniendo las unidades probadas, se ejecutan las pruebas de integración y luego de estas las del sistema. Además, la propia generación de los casos de prueba del sistema sirve como forma de validar los requerimientos. Lo mismo ocurre con los casos de prueba de integración y la verificación del diseño. Por último, la Figura relaciona a los clientes o usuarios con la realización de la prueba de aceptación.

Este modelo brinda líneas generales sobre el momento de generar y el momento de ejecutar las pruebas, sin embargo, no debe tomarse como proceso de desarrollo. Esto último responde a que si el proceso de desarrollo es el modelo V se obtiene un proceso de desarrollo en cascada. La idea detrás de este modelo es usarlo "dentro" de cualquier modelo de desarrollo. La intención primaria es ubicar las actividades principales de las pruebas. Por ejemplo, si se trabaja con un modelo iterativo e incremental, en cada iteración se puede usar el modelo V.

Hay quienes sostienen que no siempre es conveniente seguir este modelo. Si los requerimientos son muy cambiantes la construcción temprana de casos de prueba puede ser una perdida de tiempo y esfuerzo. Ocurre que muchos de estos casos deben ser eliminados ya que prueban requerimientos que cambiaron y ya no se van a ejecutar.

Otros aspectos de la planificación de la V&V que no son tratados en la tesis son los siguientes:

- Balancear los enfoques estáticos y dinámicos de la verificación.
- Definir estándares y procedimientos para las revisiones y las pruebas de software.
- Establecer el proceso de testing.
- Establecer los requerimientos de seguimiento de incidentes.
- Calendarización de todas las pruebas y asignación de recursos.
- Establecer procedimientos para el registro de las pruebas.
- Establecer los requerimientos necesarios de hardware y de software para la realización de las pruebas.
- Definir cómo y quiénes ejecutan las pruebas.
- Definir criterios de terminación de las pruebas.
- Definir métodos y herramientas a usar durante la verificación.

Discusión

Entendemos que en el contexto planteado, una planificación que contenga todos los puntos mencionados, es excesiva. Creemos que los grupos no seguirían su propia planificación por resultar engorrosa y quitar productividad. Sin embargo, en el Framework

incluimos la planificación del testing pero limitando la misma. El plan sólo debe contener qué partes del software van a ser testeadas, cuándo serán testeadas dichas partes y qué tipo de cubrimiento es el exigido para cada una.

Por otro lado sugerimos el uso del modelo V sea cual sea el proceso de desarrollo escogido por el grupo. Con esto se pretende lograr que los grupos de desarrollo realicen actividades de verificación lo antes posible. De esta manera se introduce desde el principio del proyecto una preocupación por la calidad y actividades que tienden a mejorarla.

3.12. Otros Temas del Testing

En esta sección se presentan de forma compacta dos temas: cubrimiento en el testing y herramientas de ayuda al testing.

El cubrimiento del testing es una medida de cuánto o qué porciones se cubren con los casos de prueba de un producto del desarrollo de software.[14] El cubrimiento puede ser visto a nivel de requerimientos. Este cubrimiento nos indica qué requerimientos son probados con casos de prueba. Normalmente se indica qué casos de prueba cubren qué requerimientos. Lo mismo sucede con el diseño. En este caso los casos de prueba pueden cubrir distintas características del diseño, por ejemplo, todos los estímulos en un diagrama de secuencia. Por último se puede considerar el cubrimiento a nivel de implementación. En este último caso el cubrimiento se asocia a lo visto en las técnicas de caja blanca. Un criterio de cubrimiento indica qué porción del código y de qué forma se quiere cubrir con los casos de prueba.

Las herramientas de ayuda al testing son cada vez más usadas en la industria. A continuación presentamos una descripción de las que consideramos más relevantes.

Verificación automatizada estática. Estas herramientas se presentaron en la sección de Análisis de Código. Estas realizan chequeos de sintaxis, generan grafos de flujo de control y buscan detectar problemas de estructura en el código fuente.

Ejecución automática de pruebas. Estas herramientas permiten que luego de generados los casos de prueba estos se puedan ejecutar de forma automática. Un ejemplo a nivel unitario es la herramienta JUnit.[15]

Captura/Reproducción. Estas herramientas permiten capturar la interacción de un usuario con un sistema. Mientras el programa se ejecuta y el usuario interactúa con el mismo la herramienta *graba* lo que realiza el usuario y lo que responde el sistema. Luego, esta grabación se puede cambiar de forma manual mediante scripts que provee la herramienta. Este caso de prueba se puede ejecutar y también multiplicar cambiando unos pocos parámetros. IBM Rational Robot[16] es un ejemplo de estas herramientas.

[14] Acá por producto se entiende no sólo a la implementación sino a cualquier producto desarrollado durante el proyecto de software. Por ejemplo, documentos de requerimientos, diseño y código.

[15] JUnit: http://www.junit.org

[16] IBM Rational Robot: http://www-306.ibm.com/software/awdtools/tester/robot/

Soporte a la integración. Estas herramientas ayudan en la generación de drivers y de stubs para realizar la prueba de integración.

Evaluación de cobertura. Estas herramientas sirven para evaluar la cobertura del código alcanzada al ejecutar determinados casos de prueba. Estas se basan en la instrumentación automática de código. La instrumentación deja *marcas* en el código de forma de tener trazas luego de ejecutados los casos de prueba. A partir de estas trazas se conoce el cubrimiento alcanzado.

Generadores de casos de prueba. Estas herramientas buscan generar casos de prueba de forma automática. Esta generación puede ser tanto a partir del código como a partir de una especificación. Algunos ejemplos fueron presentados en las secciones Técnicas de Caja Negra y Técnicas de Caja Blanca.

Carga y desempeño. Estas herramientas ponen cargas altas en el sistema para ver cómo reacciona el mismo. Algunos ejemplos de este tipo de herramientas son: Mercury Load Runner,[17] Compuware QALoad[18] y OpenSTA.[19]

Seguimiento de defectos. Estas herramientas permiten registrar los defectos encontrados y realizar el seguimiento. Normalmente permiten definir un *workflow* por el que pasan los defectos e incidentes.Por ejemplo, un incidente es primero dado de alta, luego es corregido y por último es verificada la corrección. Herramientas dentro de esta categoría son, por ejemplo, Bugzilla[20] y Mantis.[21]

Gestión del test. Estas herramientas ayudan a gestionar gran cantidad de conjuntos de casos de prueba. Normalmente son ambientes que integran las herramientas que se mencionaron antes.

Discusión

Las herramientas de apoyo al testing son cada vez más usadas y normalmente más necesarias. En el Framework se busca tener automatizadas las pruebas y conocer el cubrimiento de código alcanzado.

Tener automatizada la ejecución de las pruebas permite mejorar la productividad y dar mayores garantías de que el software se desempeña adecuadamente. Además brinda ganancias en calidad en otros aspectos, tales como mantenibilidad y modificabilidad. Las herramientas a usar pueden ser solamente de ejecución automatizada o también con agregados de ayuda en la construcción de los casos, por ejemplo herramientas de captura/reproducción.

La evaluación de la cobertura de código alcanzada se introduce en el Framework. Se recomienda realizar esta evaluación a nivel de clases con alguna herramienta que instrumente el código. Si se realiza de forma manual la perdida en productividad y los errores al

[17] Load Runner: http://www.mercury.com/us/products/performance-center/loadrunner/
[18] QALoad: http://www.compuware.com/products/qacenter/qaload.htm/
[19] OpenSTA: http://www.opensta.org
[20] Bugzilla: http://www.bugzilla.org/
[21] Mantis: http://www.mantisbt.org/

hacer la tarea pueden ser muy grandes. Entendemos que la determinación de la cobertura de código alcanzada por las pruebas es una tarea inherentemente automática.

Pensamos que el uso de otras herramientas, al menos durante el primer año de aplicación del Framework, es innecesario para el tipo de desarrollo y sobre todo el tipo de grupos de desarrollo dentro del CSI. Creemos que los grupos no están entrenados lo suficiente en este tipo de herramientas y que no tienen la disciplina necesaria como para usarlas. Por consiguiente su uso no es obligatorio ni sugerido dentro del Framework. En la sección 3 "Trabajo a Futuro" del capítulo 5 se presenta una breve discusión sobre la posibilidad de usar otras herramientas de soporte al testing.

3.13. Discusión General de la Verificación y Validación

La verificación y la validación es la disciplina más importante en el Framework propuesto. Creemos que, dada la realidad de los grupos de desarrollo, realizar pruebas de forma continua durante el proceso de construcción de software es la forma de lograr una mejora en la calidad.

Lamentablemente, por distintas razones, las revisiones de código tales como las recorridas o las inspecciones no pueden ser usadas en los proyectos. Esto deja al testing toda la responsabilidad del aseguramiento y control de la calidad del código.

El testing de caja negra es el más adecuado para los grupos de desarrollo del CSI. Por esto se seleccionan las estrategias de partición en clases de equivalencia, valores límite y estrategias basadas en diagramas de estado para ser parte del Framework. Sin embargo, técnicas muy interesantes y eficaces tales como los grafos causa-efecto, no son consideradas debido a su alta complejidad y poco conocimiento por parte de los desarrolladores.

Se propone en el Framework el testing de caja blanca a nivel de clases. Se pide que el cubrimiento alcanzado sea el de arcos. Otro tipo de cubrimientos son dejados de lado debido a su alta complejidad.

La realización de pruebas de aceptación, de regresión y de planes "livianos" del testing también son consideradas actividades importantes del Framework propuesto.

Otras actividades del testing, tales como las pruebas de performance son dejadas fuera del Framework.

Muchas de las actividades descritas en esta sección son pensadas para grandes desarrollos, en tamaño del producto, en recursos y en tiempo, y no son aplicables para el tipo de desarrollo que se está investigando. Varias de las actividades que se encuentran en el Framework y son de la disciplina V&V son "recortes" o adaptaciones de actividades presentadas en esta sección.

4. Métricas y Medidas de Diseño

En esta sección se presenta el estado del arte en el área de métricas y medidas de diseño. Esta sección está dividida en distintos tópicos que creemos de importancia mencionar en un estudio del estado del arte.

Se muestra en esta sección la importancia de tener métricas de diseño para mejorar la calidad del producto final. Se presentan algunas referencias sobre la importancia de las métricas de diseño y se discute la importancia que las mismas tienen para el desarrollo académico de software y por lo tanto su incorporación al Framework propuesto.

Se presentan distintos modelos de calidad del diseño basado en métricas. Se presenta el motivo de la elección del modelo elegido para el Framework de mejora de la calidad.

Por último se presentan y discuten otro tipo de trabajos relacionados, como por ejemplo el uso de métricas de diseño para detectar de forma automática defectos en el diseño.

4.1. La Importancia de las Métricas y las Medidas para el Diseño

"La calidad del diseño afecta la calidad del producto final. Por lo tanto, hay una fuerte necesidad de garantizar la calidad de un diseño. Para lograr un diseño de alta calidad debe haber formas de juzgar los mismos mediante atributos deseables de calidad, por ejemplo mantenibilidad. Este juicio debe ser cuantitativo y objetivo para que sea aplicable" [EWEBBF04].

"[...] hay una necesidad de métricas y modelos que puedan ser aplicados en las primeras etapas del desarrollo (requerimientos y diseño), para asegurar que el análisis y el diseño tienen propiedades internas favorables que llevan a tener un desarrollo de un producto final de calidad. Esto dará a los desarrolladores una oportunidad de arreglar los problemas, remover las irregularidades y las no conformidades a los estándares, y eliminar tempranamente en el ciclo de desarrollo complejidades no deseadas. Esto debe ayudar significativamente a reducir el re-trabajo durante y después de la implementación, y a diseñar planes de test efectivos y a mejorar la planificación del proyecto y de los recursos" [BD02].

"El diseño está en el corazón de cualquier disciplina de la ingeniería y la ingeniería de software no es una excepción. Qué tan buenos son los diseños determina la efectividad y viabilidad de los sistemas de software que construimos [...] Estamos todavía en la etapa de madurez que está más cerca de la artesanía que de una disciplina de ingeniería. Si bien podemos decir, usando la experiencia, que el diseño X es mejor que el diseño Y, hacemos estos juicios desde un punto de vista de un crítico de arte, no desde un punto de vista de un ingeniero recurriendo a medidas derivadas de una base teórica bien establecida" [Per01].

Discusión

Entendemos que la calidad del diseño es de suma importancia para la calidad del producto final. También creemos que las métricas de diseño pueden ayudar a mejorar el mismo de forma relativamente temprana. De todas formas, también compartimos lo mencionado en [Per01] respecto a lo inmaduro de la disciplina en estos momentos. Por esto último es que se debe tener mucho cuidado al aplicar las métricas y debido a esto en el Framework presentado se marca el cuidado que se debe tener al interpretar las mismas.

4.2. Modelos de Calidad del Diseño OO

Existen varios modelos de calidad del diseño orientado a objetos basado en métricas. A continuación se mencionan algunos de ellos.

MOOSE

Las métricas para el diseño orientado a objetos propuestas por Chidamber y Kemerer en el artículo *A Metrics Suite for Object Oriented Design* [CK94], tienen las siguientes particularidades:

- Las mismas fueron validadas en la práctica en dos sitios de desarrollo.
- Fueron evaluadas analíticamente usando el conjunto de principios de la medida propuesto por Weyuker.[22]
- Usan a la Ontología de Bunge[23] como base teórica.

El modelo tiene 6 métricas y en el artículo se presenta para cada una la base teórica de la misma, la parte analítica que confronta las mismas con los principios de Weyuker y la interpretación de los datos obtenidos en los dos sitios donde fueron validadas. A continuación restringimos la presentación a las métricas y su base teórica.

Métrica 1: Weighted Methods Per Class (WMC)

Definición: Dada una clase C con métodos $M_1, \ldots M_n$ tal que $c_1, \ldots c_n$ es la complejidad de cada método respectivamente se define WMC de la siguiente manera:

WMC (C) = $\sum_{i=1}^{n} c_i$

La complejidad de cada método no se define de forma deliberada, de esta manera se logra mayor flexibilidad en la métrica. Puede ser apropiada alguna métrica tradicional de programación estructurada.

[22] El artículo *Evaluating Software Complexity Measures* de Weyuker presenta los principios mencionados. El mismo no fue leído para realizar esta tesis.

[23] Los libros *Treatise on Basic Philosophy: Ontology I: The Furniture of the World* y *Treatise on Basic Philosophy: Ontology II: The World of Systems* de Bunge, presentan la Ontología. Los mismos no fueron leídos para realizar esta tesis.

Base Teórica:

- El número de métodos y la complejidad de los mismos es un indicador de cuánto tiempo y esfuerzo se requiere para desarrollar o mantener la clase.
- Cuanto mayor sea la cantidad de métodos en una clase va a ser mayor el impacto potencial en los hijos de la misma. Esto es debido a que los hijos van a heredar todos los métodos de la clase.
- Normalmente las clases con gran cantidad de métodos tienden a ser más específicas y dependientes de la aplicación, limitando la posibilidad de reuso.

Métrica 2: Depth of Inheritance Tree (DIT)

Definición: En casos que se considera la herencia múltiple, el DIT es el máximo del largo desde la clase en consideración hasta la raíz del árbol de jerarquía de herencia. En caso de herencia simple es la profundidad desde la raíz hasta el nodo que representa a la clase.

Base Teórica: Es una medida que indica cuántos ancestros pueden afectar a la clase.

- Cuanto más profunda esté la clase en la jerarquía es mayor la cantidad de métodos que hereda, de esta manera es más complejo predecir su comportamiento.
- Árboles más profundos constituyen una complejidad más grande de diseño, esto es debido a que más métodos y clases están participando.
- Cuanto más profunda en la jerarquía esté una clase particular más grande es el reuso potencial de los métodos heredados.

Métrica 3: Number Of Children (NOC)

Definición: Dada una clase C se define NOC de la siguiente manera:

NOC (C) = Cantidad de sub-clases inmediatas de C.

Base Teórica: Es una medida de cuántas sub-clases van a heredar los métodos de la clase padre.

- Cuanto más grande es la cantidad de hijos más grande es el reuso. La herencia es una forma de reuso.
- Cuanto más grande es la cantidad de hijos más grande es la posibilidad de abstracciones impropias de la clase padre. Puede estar denotando un mal uso de la herencia.
- El número de hijos da una idea de la influencia potencial que tiene una clase en el diseño. Si una clase tiene un gran número de hijos va a requerir un mayor testing de los métodos de la clase.

Métrica 4: Coupling Between Object Classes (CBO)

Definición: CBO de un objeto es la cantidad de otros objetos con los cuales está acoplado. Un objeto está acoplado a otro si uno actúa sobre otro, por ejemplo, uso de métodos o variables de instancia del otro.

Base Teórica: Un objeto está acoplado a otro si uno de ellos actúa sobre el otro.

- El excesivo acoplamiento entre objetos va en detrimento de un diseño modular y puede impedir el reuso. Cuanto más independiente es una clase más fácil es el reuso.
- De forma de mejorar la modularidad y promover la encapsulación el acoplamiento interclases debe mantenerse a un mínimo. Cuanto mayor es el acoplamiento el diseño es más sensible al cambio, por lo que el mantenimiento es más difícil de realizar.
- Medidas sobre el acoplamiento son de utilidad para poder determinar qué tan complejo el testing de ciertas partes del diseño puede llegar a ser. Cuanto mayor sea el acoplamiento más riguroso debe ser el testing.

Métrica 5: Response For a Class (RFC)

Definición: RFC = |RS| siendo RS el conjunto de respuesta de la clase.

Base Teórica: El conjunto de respuesta de una clase puede ser definido de la siguiente manera:

RS = $\{M\} \bigcup_{all\ i} \{R_i\}$

siendo $\{R_i\}$ = al conjunto de métodos llamados por el método i y $\{M\}$ = al conjunto de todos los métodos de la clase. El RFC de una clase es el conjunto de métodos que puede ser potencialmente ejecutado en respuesta al mensaje recibido por un objeto de la clase. Tener en cuenta que sólo se define para el primer nivel de acoplamiento. Al incluir métodos que son llamados por fuera de la clase, también mide la comunicación potencial entre la clase y otras clases.

- Si un gran número de métodos puede ser invocado en respuesta a un mensaje, el testing de la clase es más complicado debido a que requiere un mayor nivel de entendimiento del diseño por parte del testeador.
- Cuanto más grande el número de métodos que pueden ser invocados por una clase mayor es la complejidad de la misma.

Métrica 6: Lack of Cohesion in Methods (LCOM)

Definición: Consideramos una clase C_1 con n métodos $M_1, M_2 \ldots M_n$. Sea I_i = conjunto de las variables de instancia usadas por el método M_i. Hay $_n$ de estos conjuntos $I_1 \ldots I_n$. Sea P = $\{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ y Q = $\{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. Si los n conjuntos $I_1 \ldots I_n$ son $\emptyset$ entonces P = $\emptyset$.

$$LCOM = \begin{cases} |P| - |Q| & \text{si } |P| > |Q| \\ 0 & \text{en cualquier otro caso} \end{cases}$$

Base Teórica:

- Es deseable que los métodos de una clase tengan una alta cohesion. Esto promueve la encapsulación.
- La falta de cohesion implica que probablemente las clases deban ser divididas en dos o más subclases.
- Una baja cohesion aumenta la complejidad, aumentando así la posibilidad de introducir errores durante el proceso de desarrollo.

MOOD

Brito e Abreu y Carapuça en el artículo [eAC94], definen un conjunto de métricas para diseños orientados a objetos. Establecen siete criterios que deben cumplir las métricas por ellos propuestas y que se listan a continuación:

1. Las métricas deben ser definidas formalmente.
2. Las métricas que no son de tamaño deben ser independientes del tamaño del sistema.
3. Las métricas no deben tener dimensiones o deben ser expresadas en algún sistema unitario consistente.
4. Las métricas se deben poder obtener temprano en el ciclo de desarrollo.
5. Las métricas deben ser escalables (por ejemplo tienen que servir para subsistemas).
6. Las métricas deben ser fáciles de computar.
7. Las métricas deben ser independientes del lenguaje de diseño o de implementación.

Dos métricas se relacionan con la encapsulación y el ocultamiento de la información, estas son MHF y AHF.

Dada una clase C_i se define:

$M_v(C_i)$ - número de métodos visibles (interface) en la clase C_i.

$M_h(C_i)$ - número de métodos ocultos en C_i.

$M_d(C_i) = M_v(C_i) + M_h(C_i)$.

TC = es el número total de clases en el sistema.

Entonces MHF se define como:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Definiendo similarmente a A_d, A_v y A_h se define AHF como:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Otras dos métricas se relacionan con la herencia, estas son, Method Inheritance Factor (MIF) y Attribute Inheritance Factor (AIF).

$M_d(C_i)$ = número de métodos definidos en C_i.

$M_n(C_i)$ = número de métodos nuevos definidos en C_i. Nuevos denota a todos los métodos que no son heredados sobrescritos.

$M_i(C_i)$ = número de métodos heredados en C_i que no son sobrescritos.

$M_o(C_i)$ = número de métodos sobrescritos en C_i.

$M_a(C_i)$ = número de métodos disponibles en C_i. Disponibles denota a los métodos definidos en la clase más los heredados.

El número total de métodos definidos se define:

TMD = TMN + TMO = $\sum_{k=1}^{TC} M_d(C_k)$

siendo el total de nuevos métodos definidos:

TMN = $\sum_{k=1}^{TC} M_n(C_k)$

y el número total de métodos sobrescritos:

TMO = $\sum_{k=1}^{TC} M_o(C_k)$

También se define el número total de métodos heredados y el número total de métodos disponibles como:

$TM_i = \sum_{k=1}^{TC} M_i(C_k)$

$TM_a = TM_d + TM_i = \sum_{k=1}^{TC} M_a(C_k)$

Entonces MIF se define como:

$$MIF = \frac{TM_i}{TM_a}$$

Similarmente se define Attribute Inheritance Factor AIF como:

$$AIF = \frac{TA_i}{TA_a}$$

Dos métricas se definen para evaluar el acoplamiento y los *clusters*.

Una clase C_c es cliente de una clase C_s si la primera contiene alguna referencia a la segunda, tanto sea un método o un atributo.

Toda clase en un sistema es cliente potencial del resto de las clases del sistema. Entonces el valor máximo de la relación cliente-proveedor (sin considerar que el cliente y el proveedor son la misma clase), está dado por TC^2 - TC.

Definimos la siguiente función:

$$isClient(C_c, C_s) = \begin{cases} 1 & \text{si } C_c \text{ es cliente de } C_s \text{ y } C_c \neq C_s \\ 0 & \text{en cualquier otro caso} \end{cases}$$

Entonces se define CF como:

$$CF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{TC} isClient(C_i, C_j)]}{TC^2 - TC}$$

Si TCC es el número de *class clusters*, se define el Clustering Factor de la siguiente manera:

$$CLF = \frac{TCC}{TC}$$

Una métrica se define en relación al polimorfismo, Polymorphism Factor (PF).

DC(C_i) = es el número de descendientes de C_i.

Cuando todos los métodos de las clases que heredan de otra son sobrescritos, entonces tenemos la mayor cantidad de diferentes situaciones polimórficas que puede haber en un sistema. Esto está dado por el siguiente número:

$\sum_{i=1}^{TC}[M_d(C_i) * DC(C_i)]$

Sin embargo, para un sistema dado el número total de situaciones polimórficas está dado por el siguiente número:

$\sum_{i=1}^{TC}[\sum_{j=1}^{DC(C_i)} M_o(C_j)]$

Notar que la suma interna se refiere a los hijos de C_i.

Se define el Polymorphism Factor de la siguiente manera:

$$PF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{DC(C_i)} M_o(C_j)]}{\sum_{i=1}^{TC}[M_d(C_i) * DC(C_i)]}$$

La última métrica que se define tiene que ver con el reuso y su nombre es Reuse Factor.

Se consideran dos tipos de reuso, aquel que corresponde al uso de una clase de una librería y el reuso realizado mediante la herencia. En este último caso solamente se consideran los métodos heredados y no los atributos, debido a que los primeros son mucho más difíciles de mantener y construir que los atributos. Para representar si una clase pertenece a una librería o no se define la siguiente función:

$$inLibrary(C_i) = \begin{cases} 1 & \text{si } C_i \text{ es una clase de la librería} \\ 0 & \text{en cualquier otro caso} \end{cases}$$

Entonces se define Reuse Factor de la siguiente manera:

$$RF = \frac{\sum_{i=1}^{TC} inLibrary(C_i) + MIF * \sum_{i=1}^{TC}[1 - inLibrary(C_i)]}{TC}$$

Se definen tres tipos de límites para las métricas. Los límites son: límite inferior recomendado (LI), intervalo recomendado (INT) y límite superior recomendado (LS). Estos

se presentan en la tabla 2.1. No se definen los valores para cada límite ya que no se tenían suficientes datos en el momento que los autores escribieron el artículo. En un artículo posterior, Evaluating the impact of object-oriented design on software quality,[24] Brito e Abreu y Melo realizan una validación empírica de MOOD.

MÉTRICA DE MOOD	**LI**	**INT**	**LS**
Method Inheritance Factor		X	
Attribute Inheritance Factor		X	
Coupling Factor			X
Clustering Factor	X		
Polymorphism Factor		X	
Method Hiding Factor	X		
Attribute Hiding Factor	X		
Reuse Factor	X		

Tabla 2.1: Forma de las Heurísticas de Diseño Basadas en MOOD

Ejemplos (no realistas) del uso de estos límites son los siguientes:

- El MIF debe estar entre 0,25 y 0,37
- El CF debe ser menor a 0,52
- El RF debe ser mayor a 0,42

Notar que todas las métricas presentadas en MOOD tienen valores entre 0 y 1.

QMOOD

En el artículo *A Hierarchical Model for Object-Oriented Design Quality Assessment* [BD02], se define el modelo QMOOD de calidad de diseño orientado a objetos. QMOOD se define en cuatro niveles y tres conexiones entre los mismos como se presenta en la Figura 2.13. Las conexiones son mapeos que se definen para vincular niveles adyacentes.

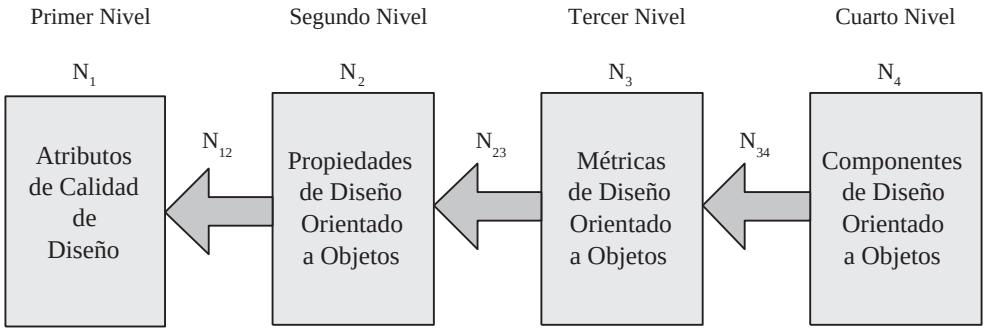


Figura 2.13: Niveles y Conexiones en QMOOD

En el Cuadro 2.2, se muestran los atributos de calidad del diseño, estos corresponden al primer nivel definido en QMOOD.

[24]Este artículo no fue considerado para la realización de la tesis

ATRIBUTOS DE CALIDAD	DEFINICIÓN
Reusability	Refleja la presencia de características del diseño orientado a objetos que permiten a un diseño ser reusado en un nuevo problema sin un esfuerzo significativo.
Flexibility	Características que permiten incorporar cambios en el diseño.
Understandability	Propiedades de un diseño que permiten que sea comprendido fácilmente. Esto se relaciona directamente con la complejidad de la estructura del diseño.
Functionality	Las responsabilidades asignadas a las clases de un diseño, las cuales están disponibles para las clases a través de sus interfaces públicas.
Extendibility	Se refiere a la presencia de propiedades en un diseño existente que permiten la incorporación de nuevos requerimientos en el mismo.
Effectiveness	Se refiere a la posibilidad de un diseño de lograr la funcionalidad y el comportamiento deseados usando conceptos y técnicas de diseño orientado a objetos.

Tabla 2.2: Definiciones de Atributos de Calidad de QMOOD

Las propiedades del diseño, correspondientes al nivel dos, son conceptos tangibles que pueden ser evaluados examinando la estructura interna y externa, las relaciones, y la funcionalidad de componentes del diseño. Las propiedades incluidas en QMOOD se muestran en el Cuadro 2.3.

Cada una de las propiedades que se identifican en el nivel dos representan un atributo o característica del diseño que está lo suficientemente definido como para ser evaluado objetivamente usando una o mas métricas de diseño. Para los diseños orientados a objetos esta información debe incluir la definición de las clases, las jerarquías de clases, y las declaraciones de todas las operaciones de la clase junto con los tipos de los parámetros y las declaraciones de los atributos. Las métricas de QMOOD se presentan en el Cuadro 2.4 y son las que componen el nivel tres.

Las componentes del diseño que son identificables y que definen la arquitectura de un diseño orientado a objetos son: objetos, clases y las relaciones entre ellas. La calidad de un objeto está determinada por quienes lo constituyen, que incluyen a los atributos, los métodos y otros objetos (composición). Otra componente que puede ser identificada en los diseños OO, son las estructuras de generalización-especialización, o jerarquía de clases, que organizan familias de clases relacionadas. Entonces, un conjunto de componentes que puede ayudar a analizar, representar e implementar un diseño OO debe incluir: atributos, métodos, objetos (clases), relaciones y jerarquías de clases.

PROPIEDADES DEL DISEÑO	**DEFINICIÓN**
Tamaño del Diseño	Una medida del número de clases usadas en el diseño.
Jerarquías	Las jerarquías se usan para representar conceptos de generalización-especialización diferentes en un diseño. Es el número de clases que no heredan de otras y tienen hijos en el diseño.
Abstracción	Una medida del aspecto de generalización-especialización en el diseño. Las clases en un diseño que tienen uno o más descendientes exhiben esta propiedad de abstracción.
Encapsulamiento	Definido como el mantener juntos los datos y el comportamiento dentro de un solo constructor. En el diseño orientado a objetos la propiedad se refiere especialmente a diseñar clases que impiden accesos a declaraciones de atributos mediante la definición de los mismos como privados, protegiendo, de esta manera, la estructura interna de los objetos.
Acoplamiento	Define la interdependencia de un objeto con otros objetos en un diseño. Es una medida del número de objetos que pueden ser accedidos por otro objeto de forma de funcionar correctamente.
Cohesión	Evalúa lo relacionados que están los métodos y los atributos en una clase. Un fuerte solapamiento en los parámetros de un método y tipos de atributos es una indicación de una fuerte cohesión.
Composición	Mide las relaciones “parte-de”, “tiene”, “consiste-en”, “parte-todo”, las cuales son relaciones de agregación en un diseño orientado a objetos.
Herencia	Una medida de la relación “es-un” entre clases. Esta relación está relacionada con el nivel de anidamiento entre clases en una jerarquía de herencia.
Polimorfismo	La habilidad de sustituir objetos cuyas interfaces encajan en tiempo de corrida. Es una medida de los servicios que son dinámicamente determinados en tiempo de corrida en un objeto.
Mensajes	Una cuenta del número de métodos públicos que están disponibles como servicios a otras clases. Esta es una medida de los servicios que una clase provee.
Complejidad	Una medida del grado de dificultad para entender y comprender la estructura interna y la externa de las clases y sus relaciones.

Tabla 2.3: Definiciones de Propiedades del Diseño de QMOOD

MÉTRICA	NOMBRE	DEFINICIÓN
DSC	Tamaño del Diseño en Clases	Esta métrica es el número total de clases en el diseño.
NOH	Número de Jerarquías	Esta métrica es el número de jerarquías en el diseño.
ANA	Número Promedio de Ancestros	El valor de esta métrica es el número promedio de clases de las cuales una clase hereda información. Es computado determinando el número de clases en todos los caminos desde las clases raíz a todas las clases en la estructura de herencia.
DAM	Número Promedio de Ancestros	Esta métrica es la razón entre el número de atributos privados y "protected", y el número total de atributos declarados en la clase. Lo deseado es un alto valor de DAM. Rango de valores de 0 a 1.
DCC	Acoplamiento Directo entre Clases	Esta métrica es el número de clases diferentes con las cuales una clase se relaciona directamente. La métrica incluye clases que están directamente relacionadas mediante declaraciones de atributos y mensajes en métodos.
CAM	Cohesión entre Métodos de una Clase	Esta métrica computa la relación entre métodos de una clase basada en la lista de parámetros de los métodos. La métrica es computada usando la suma de la intersección de parámetros de un método con el máximo conjunto independiente de todos los tipos de parámetros en la clase. Es preferible un valor de esta métrica cercano a 1,0. Rango de valores de 0 a 1.
MOA	Medida de la Agregación	Esta métrica mide el uso de la relación parte-todo, llevada a cabo usando atributos. Esta métrica es el número de declaraciones de datos cuyos tipos son clases definidas por el usuario.
MFA	Medida de la Abstracción Funcional	Esta métrica es la razón entre el número de métodos heredados por una clase y el número total de métodos accesibles por métodos definidos en la clase. Rango de valores de 0 a 1.
NOP	Número de Métodos Polimórficos	Esta métrica es el número de métodos que exhiben comportamiento polimórifico.
CIS	Tamaño de la Interface de Clase	Esta métrica es el número de métodos públicos en una clase.
NOM	Número de Métodos	Esta métrica es el número de métodos definidos en una clase.

Tabla 2.4: Descripciones de las Métricas de Diseño de QMOOD

Quedan fuera del alcance de este estado del arte las conexiones y mapeos entre los niveles adyacentes.

En el artículo mencionado, los autores presentan dos evaluaciones de QMOOD. Una de ellas compara lo obtenido usando las métricas de QMOOD con la opinión de 13 evaluadores independientes. Se usó un test estadístico para determinar la correlación entre las medidas del modelo y la evaluación de los evaluadores independientes. Con un nivel de significancia del 5 %, sólo para dos evaluadores no se cumplieron las hipótesis del test.

OODQM

En el artículo [Mar94], Martin presenta un conjunto de métricas para medir la calidad de un diseño orientado a objetos en términos de la interdependencia entre los subsistemas del diseño. Los diseños que son altamente interdependientes tienden a ser rígidos, no reutilizables y difíciles de mantener. Pero, la interdependencia es necesaria si los subsistemas del diseño tienen que colaborar. Por lo tanto, algunas formas de dependencia son deseables mientras que otras no. En el artículo se presenta un pattern en el cual todas las dependencias son de la forma deseada. También se describe un conjunto de métricas que mide la conformidad de un diseño al pattern presentado. En el artículo se muestra que las *buenas* dependencias dependen de algo que es *Estable* mientras que las *malas* dependen de algo que es *Inestable*. Si los subsistemas no dependen de otros se les llama *Independientes*. Si los subsistemas tienen muchos dependientes de ellos se les llama *Responsables*. Los subsistemas Independientes y Responsables no tienen razones para cambiar (no dependen de nadie) y tienen muchas razones para no cambiar (muchos subsistemas dependen de ellos). Tres métricas se identifican para detectar la Independencia, Responsabilidad e Inestabilidad:

Ca: Afferent Couplings: Es el número de clases fuera de este subsistema que depende de clases dentro de este subsistema.

Ce: Efferent Couplings: Es el número de clases dentro de este subsistema que depende de clases fuera de este subsistema.

I: Instability: (Ce/(Ca+Ce)): Esta métrica tiene un rango de valores de 0 a 1. I=0 indica una categoría totalmente estable. I=1 indica una categoría totalmente inestable.

Las clases que son estables pueden ser lo suficientemente flexibles para poder aceptar cambios. Estas clases deben ser *Abstractas*. Entonces, si un subsistema es estable debe ser lo suficientemente abstracto para que pueda ser extendido. Subsistemas estables que son extensibles son flexibles y no restringen el diseño. Razonando de forma análoga, se quiere que los subsistemas inestables no sean abstractos. Para la Abstracción se define la siguiente métrica:

A: Abstractness: (cantidad de clases abstractas en el subsistema)/(cantidad total de clases en el subsistema). Esta métrica tiene valores de 0 a 1. 0 significa un subsistema concreto y 1 significa un subsistema completamente abstracto.

Considerando que los subsistemas que queremos son aquellos que son totalmente Estables y Abstractos a la vez o totalmente Inestables y Concretos a la vez, y sabiendo que no todos los sistemas pueden caer en estas dos categorías, se define la *secuencia principal*.

Esta secuencia es una recta que va desde el punto (1,0) subsistema totalmente inestable y totalmente concreto a (0,1) subsistema totalmente estable y totalmente abstracto. Un subsistema que esté en la *secuencia principal* tiene el número correcto de clases abstractas y concretas en proporción a las dependencias efferent y afferent. Es deseable que todos los subsistemas estén cerca de la *secuencia principal*. Esto lleva a otra métrica que es la distancia a la *secuencia principal*.

D: Distance: $|A+I-1|/2$: Es la distancia perpendicular del subsistema a la *secuencia principal*. Esta métrica varía desde 0 a 0,707. La métrica normalizada para que varíe desde 0 a 1 se llama **Dn**.

Se plantea que los subsistemas que no tengan valores cercanos a cero en la métrica D sean revisados para lograr plantear una restructura.

Una Comparación de los Modelos

En un trabajo [EWEBBF04], El-Wakil, El-Bastawisi, Boshra y Fahmy estudian y comparan cuatro modelos de calidad de diseño orientado a objetos: MOOSE [CK94], Lorenz and Kidd metrics suite,[25] MOOD [eAC94] y QMOOD [BD02]. Se introducen seis propiedades de calidad de diseño que los modelos deben poseer y luego se evalúa cada uno de los modelos contra las propiedades definidas. A continuación se listan las seis propiedades:

1. Depender solamente de características de diseño de alto nivel. Esto permite evaluar el diseño en etapas tempranas.

2. Deben ser establecidos explícitamente los objetivos del modelo y las características de calidad a ser evaluadas.

3. Las métricas deben ser definidas con precisión. La existencia de ambigüedad en la definición de las métricas permite muchas interpretaciones para la misma.

4. Los modelos deben expresar las relaciones entre las características y las métricas de diseño de una forma clara, y preferentemente formal.

5. Debe existir una interpretación de los resultados. Los valores producidos por un modelo deben tener una interpretación que pueda ser usada para tomar decisiones sobre el diseño.

6. Los modelos deben ser validados empíricamente.

Los resultados de la evaluación de los modelos de calidad de diseño contra las propiedades deseables expuestas por los autores, están resumidas en el Cuadro 2.5. Los autores concluyen que QMOOD es el más completo y exhaustivo modelo de calidad de los cuatro estudiados.

[25] Se presenta en el libro: Object Oriented Software Metrics, de Lorenz y Kidd. No se tuvo en cuenta para esta tesis.

Propiedad	MOOSE	LK Métricas OO	MOOD	QMOOD
1	NO	SI	SI	SI
2	NO	NO	Densidad de errores, Densidad de fallas y Retrabajo Normalizado	Reusabilidad, Flexibilidad Entendibilidad, Extensibilidad y Efectividad
3	SI, Excepto WMC	SI	SI	SI
4	NO	NO	SI	SI
5	NO	NO	SI	SI
6	Validado	NO	Validado	Validado

Tabla 2.5: Evaluación de los Modelos contra las Propiedades

Discusión

En esta sección se presentaron algunos de los modelos de calidad del diseño basado en métricas que se conocen. Como se mencionó en la sección anterior, estos modelos todavía son bastante inmaduros. Todos los modelos son interesantes pero hay una falta grande de investigación en su aplicabilidad práctica. Dentro de estos modelos coincidimos con la conclusión final de la comparación que se realiza en [EWEBBF04], donde se toma a QMOOD como el modelo más completo. De todas formas, se elige lo que entendemos como el conjunto de métricas más sencillo de aplicar y de obrar en consecuencia de las medidas obtenidas que son las presentadas por Martin en [Mar94] y que llamamos OODQM.

Muchos investigadores plantean que la inmadurez del área Ingeniería de Software se debe a un constante cambio en los tópicos de investigación académica. Esta visión parece razonable si tenemos en cuenta que algunos de los modelos que presentamos en esta sección son del año 1994 y que los mismos no se han llegado a validar empíricamente de una forma adecuada. Esto provoca que no se conozcan con exactitud los rangos de aceptación de cada una de las métricas propuestas.

4.3. Otros

En [XSZC00], Xenos y otros evalúan más de 200 métricas de diseño. Los autores definen un conjunto de siete meta-métricas con las cuales realizan la evaluación.

Purao y Vaishnavi, en el artículo [PV03], realizan un sondeo de las métricas de productos (en contraste con métricas de proceso), para sistemas orientados a objetos. El propósito es entender, clasificar y analizar la investigación en curso de métricas orientadas a objetos. Se estudian 375 métricas aproximadamente.

En los artículos [BBM95] y [BBM96], Basili y otros estudian el conjunto de métricas definidas en MOOSE [CK94]. El artículo presenta los resultados de un estudio conducido en la Universidad de Maryland en el cual realizaron una validación experimental del conjunto de métricas respecto a su habilidad de identificar clases propensas a tener faltas. Los datos fueron recolectados durante el desarrollo de ocho sistemas de gestión de la información de mediano porte, basados en requerimientos idénticos. Los ocho proyectos fueron desarrollados usando un modelo de ciclo de vida secuencial, un método de análisis y diseño orientado a objetos conocido y el lenguaje de programación C++. Usando un modelo probabilístico y las mediciones obtenidas para el diseño, se clasifican las clases en dos tipos, las que se les predijo que van a tener faltas y las que no. Como se esperaba, las clases que se predijeron como con faltas tienen un gran número de faltas (250 faltas en 48 clases). Para evaluar el impacto del modelo de predicción usado, se considera, de forma de simplificar los cálculos, que las inspecciones de las clases son 100 % efectivas en encontrar faltas. En este caso, 80 clases (se predijo que tenían faltas), sobre 180 van a ser inspeccionadas y 48 clases con faltas sobre un total de 58 clases con faltas van a ser identificadas antes del testing (durante la inspección). Si se toman en consideración las faltas individuales, 250 faltas de 268 van a ser detectadas durante la inspección, esto se puede ver en el Cuadro 2.6. El mismo indica el número de clases en cada caso; entre paréntesis se indica la cantidad de faltas para las clases. Para resumir, se puede decir que los resultados muestran que las métricas OO estudiadas son útiles predictoras de clases con faltas.

	Predecida Sin Faltas	**Predecida Con Faltas**
Real Sin Faltas	90	30
Real Con Faltas	10 (18)	48 (250)

Tabla 2.6: Clasificación de los Resultados con MOOSE.

Se comparan los resultados obtenidos con métricas de código, además de que estás métricas sólo pueden ser recolectadas tarde en el proceso, estas indicaron ser peores predictoras de la probabilidad de faltas en una clase que las métricas MOOSE. La cantidad de clases con faltas detectadas con MOOSE fue de un 88 % mientras que con métricas de código fue de un 83 %. Respecto a la totalidad de las faltas fue de un 93 % contra un 86 % respectivamente. La cantidad de clases que se predijo correctamente con faltas fue de un 60 % en MOOSE y un 45 % con las métricas de código. Respecto a los defectos estos fueron un 92 % con MOOSE y un 86 % con métricas de código. Para presentar los porcentajes de los defectos, las clases fueron ponderadas de acuerdo a la cantidad de faltas, a las clases sin faltas se les dio un peso de 1.

Existen varios intentos e investigaciones para detectar defectos en el diseño de forma automática. Muchos de estos trabajos se hacen con un enfoque en las métricas de diseño. Las métricas indican dónde se encuentran los defectos del diseño. En [MG05], se presenta una investigación en progreso que busca detectar y corregir de forma automática los defectos en la arquitectura de diseños orientados a objetos. Para esto distinguen entre varios tipos de defectos como anti-patterns, defectos de diseño y *code smell*. Intentan detectar

cada tipo de defecto mediante un conjunto de métricas y rango de valores aceptables para las mismas.

Discusión

Como ya se mencionó, los esfuerzos para evaluar métricas y ver en qué contexto pueden ser usadas existen, pero lamentablemente, estos no son muchos.

El experimento por el cual parece que las métricas de MOOSE sirven para identificar clases con faltas nunca fue extendido ni corroborado en otras investigaciones por lo que las conclusiones son un poco débiles.

Los trabajos de detección automática de defectos en el diseño mediante el uso de métricas comienzan a surgir con cierta fuerza. Esta línea de trabajo no está madura pero es prometedora.

4.4. Discusión General de las Métricas y Medidas de Diseño

Esta sección indica claramente dos intereses en el desarrollo académico de software. El primero está ligado totalmente a la calidad del producto y es la utilización de métricas para lograr mejorar el diseño. El segundo responde a un interés científico de lograr recolectar información sobre las medidas obtenidas para aportar a la investigación del área.

Creemos que lo más importante a tener en cuenta es la inmadurez del área, y planteamos considerar las mediciones obtenidas con cuidado. El planteo en el Framework es sumamente conservador, planteando que al obtener resultados malos en algunas medidas se debe revisar el diseño para ver si lo obtenido es resultado de una mala opción del diseño o de una medición sin valor. Qué es un resultado malo en una medida lo resuelven los propios grupos teniendo en cuenta la media de todas las distancias a la secuencia principal y la variación estándar de la misma.

Entendemos que se deben realizar más estudios en el área ya que nuevos resultados pueden significar grandes progresos a la hora de evaluar y modificar un diseño orientado a objetos. Por otro lado, no es menor tener métricas de diseño para poder evaluar de forma objetiva un diseño. Teniendo estas métricas se obtendría, tomando prestadas las palabras de Perry, un avance importante desde un arte hacia la ingeniería.

Entendemos que, dado el estado actual de la práctica de esta área, en el desarrollo académico de software vale la pena tener un conjunto de métricas muy acotado, en principio uno o dos modelos únicamente, y no ser estricto en su uso. Esto último corresponde a lo ya mencionado, si una medida da “mal” no se asume que el diseño es incorrecto sino que el mismo se revisa para detectar cuál es el motivo de que la métrica usada haya dado tal valor.

Framework para la Mejora de la Calidad

3

Ninguna de las propuestas más conocidas para la mejora de la calidad parece servir para el desarrollo académico de software. Por esto, para mejorar la calidad de los prototipos, proponemos un Framework específico para este tipo de desarrollo. Este capítulo presenta el mismo.

La parte central del Framework es un conjunto de actividades que deben realizar los desarrolladores. Estas pertenecen a las disciplinas Especificaciones, Verificación y Validación, y Métricas y Medidas. La ejecución de las actividades propuestas busca mejorar sustancialmente la calidad de los productos desarrollados.

Las actividades del Framework se agrupan en seis Niveles de Calidad. Cada Nivel define un tipo de prototipo que se espera conseguir al trabajar en el mismo. La decisión de qué Nivel es el apropiado se basa en la calidad que debe tener el producto final.

Contenido

1. Generalidades del Framework

Como se vio en el capítulo 2, ninguna de las propuestas más conocidas para la mejora de la calidad es aplicable de forma directa para el desarrollo académico de software. Por esto, para mejorar la calidad de los prototipos, proponemos un Framework específico para este tipo de desarrollo.

Consideramos que el Framework debe ayudar tanto a los desarrolladores como a los clientes durante la ejecución de los proyectos. Entendemos que la mejor forma es proponer un conjunto de actividades que ellos deben realizar durante el desarrollo de forma de mantenerlo controlado y lograr la mejora de las propiedades de calidad requeridas por el CSI.

Las actividades propuestas acompañan las distintas Etapas del Desarrollo: Requerimientos, Diseño y Pruebas. A partir del estudio de la realidad del CSI, entendemos que no es necesario definir actividades para ser aplicadas durante la Etapa de Implementación, ya que esta etapa no es la de mayores problemas. El Cuadro 3.1 muestra las actividades del Framework y las Etapas en las cuales se deben realizar. Las actividades *Test de regresión*, *Cantidad de fallas/Casos de prueba* y *Midiendo el cubrimiento* se pueden ejecutar para todos los tipos de pruebas. Las actividades *Establecimiento del cubrimiento* y *Planificación del testing* se pueden realizar en casi cualquier Etapa y por eso no se marcan en el Cuadro. Sin embargo, se recomienda que se hayan ejecutado antes de comenzar el testing.

Las actividades del Framework son entidades independientes, pero se organizan según el Nivel de Calidad que se quiera obtener en el prototipo de software. Cada nivel define un tipo de prototipo que se espera conseguir al trabajar en el mismo. La decisión sobre el Nivel en el cual trabajar se basa en la calidad que debe tener el producto final y debe ser tomada por el cliente. Trabajar en un nivel implica ejecutar con responsabilidad profesional y ética las actividades que este agrupa.

Los Niveles van del uno al seis, y a medida que se aumenta de nivel se agregan actividades, manteniendo las del nivel anterior. El Cuadro 3.2 muestra los Niveles de Calidad y las actividades del Framework. Se marca con una celda gris a las actividades que deben ser realizadas en un Nivel.

La Figura 3.1 muestra los seis Niveles con sus actividades agrupadas en las Etapas del Desarrollo. En el Cuadro se ve claramente que hasta el Nivel cuatro se trabaja solamente con el sistema completo, en el Nivel cinco se consideran las componentes de forma aislada y en el Nivel seis se trabaja con las clases que conforman cada componente.

El Framework también define un rol llamado *Responsable de Calidad*. Este rol lo debe llevar adelante una persona que conozca ampliamente el Framework. Esta persona tiene que ser externa al cliente y al grupo de desarrollo. Este rol tiene como objetivo colaborar con los clientes y los desarrolladores para que apliquen el Framework de forma óptima.

ACTIVIDADES	**ETAPAS**					
	REQ	DAN	DBN	PS	PI	PU
Relevar requerimientos y priorizarlos	■					
Generar casos de uso y priorizarlos	■					
Validar requerimientos	■					
Validar casos de uso	■					
Especificación de componentes		■				
Midiendo el diseño		■				
Especificación de clases			■			
Especificación de métodos			■			
Testing de los casos de uso				■		
Testing funcional del sistema				■		
Test de aceptación				■		
Testing de componentes					■	
Testing intraclases						■
Testing de métodos						■
Test de regresión				■	■	■
Cantidad de fallas/Casos de prueba				■	■	■
Midiendo el cubrimiento				■	■	■
Establecimiento de cubrimiento						
Planificación del testing						

REQ Requerimientos
DAN Diseño de Alto Nivel
DBN Diseño de Bajo Nivel
PS Pruebas de Sistema
PI Pruebas de Integración
PU Pruebas Unitarias

Tabla 3.1: Actividades del Framework y Etapas del Desarrollo de Software

ACTIVIDADES	NIVELES DE CALIDAD					
	1	**2**	**3**	**4**	**5**	**6**
Relevar requerimientos y priorizarlos						
Generar casos de uso y priorizarlos						
Validar requerimientos						
Validar casos de uso						
Testing de los casos de uso						
Testing funcional del sistema						
Test de aceptación						
Establecimiento de cubrimiento (de requerimientos y casos de uso)						
Cantidad de fallas/Casos de prueba (de requerimientos y casos de uso)						
Midiendo el cubrimiento (de requerimientos y casos de uso)						
Planificación del testing (de requerimientos y casos de uso)						
Test de regresión (de requerimientos y casos de uso)						
Especificación de componentes						
Midiendo el diseño						
Establecimiento del cubrimiento (de componentes)						
Planificación del testing (de componentes)						
Testing de componentes						
Test de regresión (de componentes)						
Cantidad de fallas/Casos de prueba (de componentes)						
Midiendo el cubrimiento (de componentes)						
Especificación de clases						
Especificación de métodos						
Planificación del testing (de clases)						
Establecimiento del cubrimiento (de clases)						
Testing intraclases						
Testing de métodos						
Test de regresión (de clases)						
Cantidad de fallas/Casos de prueba (de clases)						
Midiendo el cubrimiento (de clases)						

Tabla 3.2: Actividades del Framework y Niveles de Calidad

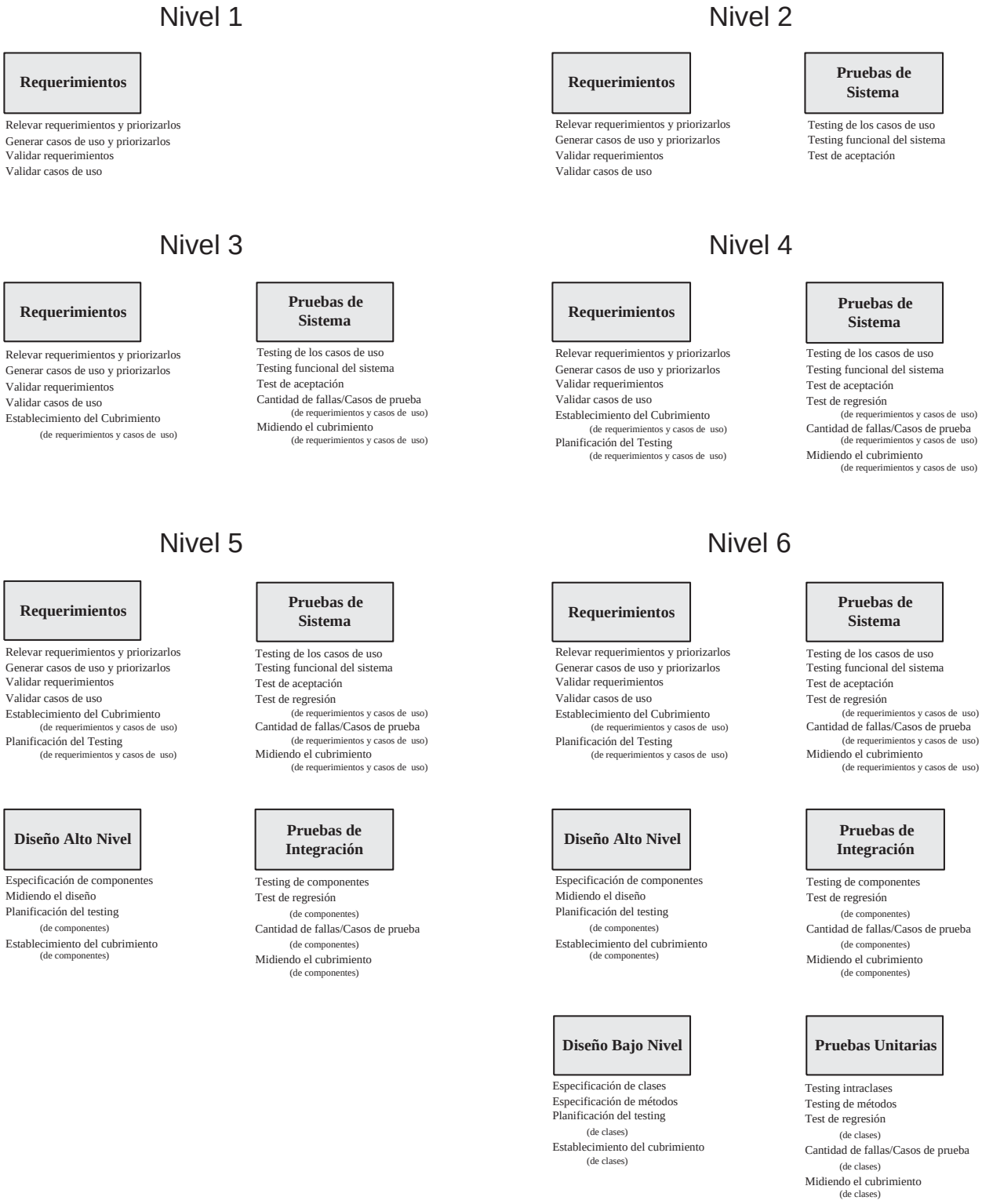


Figura 3.1: Niveles de Calidad, Etapas del Desarrollo y Actividades del Framework

1.1. Selección de las Actividades del Framework

Las actividades dentro de la ingeniería de software se pueden dividir en actividades de *construcción* y actividades de *apoyo* a la construcción. Las actividades de construcción son aquellas que pertenecen a las disciplinas de Requerimientos, Diseño, Implementación y Verificación. Estas disciplinas afectan directamente al producto que se quiere construir. Las actividades de apoyo pertenecen a disciplinas como Gestión de Proyectos, Gestión de la Calidad y Gestión del cambio, entre otras. Estas afectan de forma indirecta al producto de software.

Para el Framework tomamos solamente actividades de construcción. Esta decisión se basa en que entendemos que, para este tipo de desarrollo, las actividades de apoyo aportan poco para los objetivos que se quieren lograr. Esto se debe a que las actividades de apoyo son principalmente actividades de gestión y se ha mostrado en el capítulo 2 que la gestión suele ser sencilla en proyectos con grupos de dos a cuatro personas y con uno o dos clientes, características que tienen los proyectos del grupo CSI. Realizar de forma sistemática y continua actividades de apoyo terminaría por obstaculizar los proyectos, generando gran cantidad de *overhead*. Debido a esto, las pocas actividades de gestión que se tengan que realizar se dejan libradas al grupo de desarrollo. Creemos que considerando solamente actividades de construcción, en el contexto del CSI, se puede mejorar la calidad de los productos, mantener un Framework sencillo y no perder productividad.

La selección de las actividades del Framework, dentro de las actividades de construcción, surge de un análisis de los problemas de calidad encontrados en el desarrollo dentro del CSI y del estudio de los modelos de calidad existentes. Se detecta que los desarrolladores no tienen control sobre el software que producen, desconocen las fallas del mismo y no tienen revisado o probado lo que desarrollan. Considerando otra vez que las actividades de gestión no son apropiadas para este tipo de desarrollo, se dejan de lado todas aquellas actividades de construcción que también son de gestión. Entonces, se busca un conjunto de actividades puramente técnicas que puedan solucionar los problemas mencionados. Otras actividades que se descartan son aquellas que fueron discutidas en el capítulo 2 y que entendemos que no son aplicables en este contexto de trabajo. Las actividades del Framework las clasificamos en actividades de: Especificaciones,[1] Verificación y Validación, y Métricas y Medidas.

Son las actividades de Especificación las que hacen posible la mejora de la calidad de los productos de software. Estas buscan tener documentado el software y que el producto que se construya sea el adecuado. De esta manera, se logran entender el software y sus partes. Además, permiten derivar los casos de prueba de forma más sencilla. Las actividades incluidas en Métricas y Medidas buscan controlar el diseño, que este sea menos propenso a contener defectos y mantener controlado el testing. Este control permite obtener diseños que son modificables y extensibles, y conocer la efectividad del testing. Las actividades pertenecientes a la disciplina Verificación y Validación buscan mantener un control sobre el desarrollo, conocer las fallas de este y tener conocimiento del nivel de revisión y de

[1]Se usa Especificaciones para referirse a la IR junto con otro tipo de especificaciones distintas a la de requerimientos. Por ejemplo, la especificación del comportamiento de una componente de software.

testing realizado sobre el software.

Creemos que el conjunto de actividades propuesto, aplicado con responsabilidad, mejora la calidad del software desarrollado en el ámbito académico y ataca los problemas detectados dentro del grupo CSI.

En las secciones 2, 3 y 4 se presenta la especificación de las actividades que componen el Framework en las disciplinas Especificaciones, Verificación y Validación, y Métricas y Medidas, tal cual fue presentada a los grupos de desarrollo.

En la sección 5 se presenta la especificación de los Niveles de Calidad del Framework.

En la sección 6 se presentan las conclusiones de este capítulo.

2. Especificaciones

El uso correcto de especificaciones, tanto de requerimientos como otras, mejora la calidad del producto de software. Un producto correctamente especificado puede ser verificado, mantenido, extendido y comprendido. Un producto cuyas especificaciones no son buenas tiende a ser un producto de baja calidad.

Se puede dividir en dos a las actividades propuestas dentro de esta disciplina: la Especificación de Requerimientos y el resto de las especificaciones. Llamaremos a estas últimas Otras Especificaciones.

Especificar los requerimientos es vital para asegurar la calidad de un producto. La especificación contribuye al entendimiento entre el cliente y los desarrolladores, sirve de entrada a la generación de los casos de prueba del sistema, ayuda durante todo el desarrollo a mantenerse enfocado y, por sobre todo, sirve para definir claramente el problema a resolver.

El Cuadro siguiente muestra las actividades propuestas dentro del área Especificación de Requerimientos.

Actividades de Especificación de Requerimientos	
Act. 1	Relevar Requerimientos y Priorizarlos
Act. 2	Validar Requerimientos
Act. 3	Generar Casos de Uso y Priorizarlos
Act. 4	Validar Casos de Uso

Según lo discutido en el capítulo 2, es difícil que en los proyectos académicos se falle en la obtención de requerimientos. De todas formas, se presentan cuatro actividades para obtener los mismos, priorizarlos y validarlos. Estas actividades buscan estrechar la relación cliente-desarrollador y asegurarse de que se va a construir el producto correcto. Se brindan dos tipos de actividades, una basada en documentos de especificación de requerimientos y la otra basada en casos de uso. Esta última brinda considerable facilidad para representar

sistemas con una importante interacción con el usuario. Además, la representación con casos de uso brinda una forma sencilla de validar sistemas.

Como vimos y discutimos en el estado del arte, no sirven para este tipo de desarrollo procesos y/o técnicas "complejas" para la Ingeniería de Requerimientos.

Estas actividades, en este contexto, tienen como factor positivo el lograr tener por escrito los requerimientos. Que, en el desarrollo académico de software, los requerimientos sean fáciles de obtener y entender puede provocar que los mismos no sean documentados. Entonces, lo que es un factor positivo se puede volver en contra durante el desarrollo. Además, corresponde señalar que normalmente a los investigadores no les gusta escribir especificaciones de requerimientos, por lo que estas actividades aportan a la rigurosidad del desarrollo académico de software.

Estas cuatro actividades se ejecutan fácilmente, no son pesadas de realizar y se ajustan a la facilidad de obtención y facilidad de entendimiento de los requerimientos. De esta manera se logra tener documentados los requerimientos sin entorpecer al grupo de desarrollo.

A continuación se presenta la especificación de cada una de las actividades listadas.

Act. 1 - Relevar Requerimientos y Priorizarlos

Descripción

Esta actividad busca relevar los requerimientos y tenerlos priorizados. Para relevar los requerimientos existen diversas técnicas conocidas; se deja libertad de trabajo en este sentido. Si bien también existen varias técnicas para priorizar los requerimientos únicamente enfatizamos que esto no depende del equipo de desarrollo sino del cliente. Es el cliente quien decide qué requerimientos tienen más importancia que otros para el producto. Luego, el orden de implementación de los mismos puede ser de común acuerdo o depender más del equipo de desarrollo. Queda entendido que se deben usar técnicas para relevar requerimientos. Es bueno tener un "ida y vuelta" con el documento de requerimientos luego de tener la primera reunión.
Esta actividad no busca ir más allá de tener una lista de requerimientos y cada uno con una prioridad asignada. Se entiende que esta lista ayuda a definir el alcance del sistema. Esto se puede marcar dentro del propio documento de requerimientos donde se hace distinción entre un requerimiento dentro del alcance y otro fuera del mismo. Tampoco se detallan técnicas para determinar el alcance pero se enfatiza que el alcance es un acuerdo realizado entre el cliente y los desarrolladores.
El documento debe cumplir con buenas prácticas de escritura de requerimientos. De las buenas prácticas queremos resaltar una, y no por ser más importante que el resto sino porque se va a utilizar en otras actividades, "los requerimientos deben poder ser testeables". Es decir, para cada requerimiento que se escriba tiene que existir, al menos, una prueba (test) con la cual se pueda mostrar que el requerimiento funciona al menos para ese caso de prueba.

Entrada: *No aplica.*

Salida: Documento de Requerimientos.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Entender el producto que se debe construir.
Obj. 2	Reducir la probabilidad de desarrollar un producto no adecuado.
Obj. 3	Poder desarrollar casos de prueba concretos contra los requerimientos.
Obj. 4	Saber qué funcionalidades son más importantes y desarrollar el software en consecuencia.
Obj. 5	Mejorar la mantenibilidad del producto.
Obj. 6	Mejorar la modificabilidad del producto.

Act. 2 - Validar Requerimientos

Descripción

Validar los requerimientos consiste en revisar y aceptar (o rechazar) por parte del cliente el documento de requerimientos (este documento es salida de la actividad: Relevar Requerimientos y Priorizarlos). Esta actividad consiste en: validar los requerimientos, asegurarse que la prioridad de cada uno es la adecuada y determinar si el alcance fue definido correctamente.
Esta validación tiene como objetivo dar más seguridad, tanto al cliente como a los desarrolladores, de que se va a desarrollar el producto adecuado.
Se sugiere el siguiente proceso:

1. Entrega del documento de requerimientos al cliente. Reunión del grupo de desarrollo con el cliente donde los desarrolladores explican cómo está armado el documento.
2. Lectura del documento por parte del cliente.
3. Reunión entre cliente y desarrolladores para presentar y solucionar discrepancias.

Entrada: Documento de Requerimientos.

Salida: Documento de Requerimientos Validado.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1 Entender el producto que se debe construir.
Obj. 2 Brindar mayor seguridad tanto al cliente como a los desarrolladores sobre el producto a construir.
Obj. 3 Reducir la probabilidad de desarrollar un producto no adecuado.
Obj. 4 Asegurar qué funcionalidades son más importantes y desarrollar el software en consecuencia.

Act. 3 - Generar Casos de Uso y Priorizarlos

Descripción

Si bien existen varias formas de especificar los requerimientos de un sistema, una muy utilizada y aceptada para los sistemas de información es la basada en Casos de Uso. Los casos de uso sirven tanto para especificar los requerimientos como para "mostrarle" al cliente cómo se va a comportar el sistema en los distintos escenarios. Esto se puede hacer mediante la prototipación de los casos de uso, o mostrando las interacciones entre los actores humanos y el sistema usando dibujos de las interfaces gráficas.
Esta forma de especificar también puede ser usada como técnica para relevar requerimientos.
En caso de usar esta técnica, es bueno hacer referencia a los requerimientos del Documento de Requerimientos que se están cubriendo con cada caso de uso. No se entiende como necesario tener el Documento de Requerimientos para comenzar a trabajar en el Documento de Casos de Uso. Como los Casos de Uso se pueden utilizar como técnica para relevar los requerimientos, el Documento de Requerimientos y el de Casos de Uso se pueden realizar en paralelo.
La trazabilidad antes mencionada, desde los Casos de Uso al Documento de Requerimientos se puede tener también desde los requerimientos hacia los Casos de Uso.
Priorizar los Casos de Uso tiene la misma intención que priorizar los requerimientos.

Entrada: *No aplica.*

Salida: Documento de Casos de Uso.
Este documento incluye el Diagrama de Casos de Uso así como también la descripción de cada caso de uso del sistema. Esta descripción debe ser testeable. Es decir, se deben poder generar casos de prueba a partir de la descripción del Caso de Uso para asegurarse de que en esos casos el Caso de Uso se comporta como es esperado.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Entender el producto que se debe construir.
Obj. 4	Reducir la probabilidad de desarrollar un producto no adecuado.
Obj. 5	Poder desarrollar casos de prueba concretos contra los casos de uso.
Obj. 6	Saber qué Casos de Uso son prioritarios y desarrollar el software en consecuencia.
Obj. 2	Mejorar la mantenibilidad del producto.
Obj. 3	Mejorar la modificabilidad del producto.

Act. 4 - Validar Casos de Uso
Descripción Esta actividad es al Documento de Casos de Uso lo que Validar Requerimientos es al Documento de Requerimientos. Sus objetivos y su forma de trabajo son muy similares. **Entrada:** Documento de Casos de Uso. **Salida:** Documento de Casos de Uso Validado. **Objetivos de calidad y ganancia al realizar la misma** Obj. 1 Entender el producto que se debe construir. Obj. 2 Brindar mayor seguridad tanto al cliente como a los desarrolladores sobre el producto a construir. Obj. 3 Reducir la probabilidad de desarrollar un producto no adecuado. Obj. 4 Asegurar qué Casos de Uso son más importantes y desarrollar el software en consecuencia.

Como se mencionó anteriormente se define otro tipo de especificaciones: Otras Especificaciones. Este tipo de especificaciones constituye una base para el testing y para la mantenibilidad, reutilización y modificabilidad del producto a construir.

Estas especificaciones son normalmente utilizadas solamente por el equipo de desarrollo, aunque no se descarta que puedan existir clientes que quieran evaluar y/o validar las mismas.

El siguiente Cuadro presenta las actividades dentro del área Otras Especificaciones.

Actividades de Otras Especificaciones	
Act. 5	Especificación de Componentes
Act. 6	Especificación de Clases
Act. 7	Especificación de Métodos

Como se vio en el Cuadro 3.1, estas tres actividades acompañan la etapa de diseño del sistema. Normalmente, los diseños de sistemas anteriores desarrollados en el CSI son razonables. Sin embargo, si bien se usan diagramas de diseño, no queda especificada la funcionalidad de cada componente o clase. Entonces, parece interesante incluir estas actividades de forma de tener documentada la funcionalidad de cada parte que compone el sistema.

Estas actividades permiten razonar acerca del diseño. Definir la funcionalidad de cada componente y clase ayuda a que estas sean cohesivas. Debe quedar claro que estas actividades complementan, pero no sustituyen, a los diagramas de diseño. Estas actividades permiten también mejorar la mantenibilidad y la modificabilidad del sistema debido a que el mismo es más fácil de comprender. Por otro lado, estas especificaciones se pueden

usar también para derivar casos de prueba de caja negra para las componentes, las clases y los métodos.

A continuación se presenta la especificación de cada una de las actividades listadas.

Act. 5 - Especificación de Componentes

Descripción

Se debe especificar la funcionalidad que brinda la componente. Para esto se pueden utilizar distintas formas de especificación: definir las pre y post condiciones que deben cumplir los métodos que se pueden invocar desde fuera de la componente, describir en lenguaje natural la funcionalidad brindada por la componente y representar la funcionalidad con un diagrama de estados de la componente usando como transiciones entre los estados los métodos que se pueden invocar desde fuera de la componente.
Para mostrar la interacción y las interfaces entre las distintas componentes se pueden usar diagramas UML.

Entrada: Documento de Requerimientos. Se debe haber comenzado con el diseño de la aplicación.

Salida: Documento de Especificación de Componentes.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Entender las componentes a construir y las interfaces entre ellas.
Obj. 2	Mejorar la mantenibilidad del producto.
Obj. 3	Mejorar la modificabilidad del producto.
Obj. 4	Poder desarrollar casos de prueba concretos para las componentes.
Obj. 5	Poder descubrir más fácilmente la localización de los defectos.

Act. 6 - Especificación de Clases

Descripción

Estas especificaciones son una descripción informal del comportamiento (funcionalidad) de la clase de software. Se puede realizar mediante un texto en lenguaje natural describiendo la funcionalidad que brinda la clase. Se recomienda usar diagramas de estados de la clase.
Un Diagrama de Clases ayuda a ver la interacción entre las distintas clases.

Entrada: Documento de Especificación de Componentes.

Salida: Documento de Especificación de Clases.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1 Entender las clases a construir y las interfaces entre ellas.
Obj. 2 Mejorar la mantenibilidad del producto.
Obj. 3 Mejorar la Modificabilidad del producto.
Obj. 4 Poder desarrollar casos de prueba concretos para las clases.
Obj. 5 Poder descubrir más fácilmente la localización de los defectos.

Act. 7 - Especificación de Métodos

Descripción

Se especifican los métodos describiendo su funcionalidad.

Entrada: Documento de Especificación de Clases.

Salida: Documento de Especificación de Clases con la especificación de los métodos.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1 Entender los métodos a desarrollar.
Obj. 2 Mejorar la mantenibilidad del producto.
Obj. 3 Mejorar la modificabilidad del producto.
Obj. 4 Poder desarrollar casos de prueba concretos para los métodos.
Obj. 5 Poder descubrir más fácilmente la localización de los defectos.

Estas son todas las actividades que se definen para la disciplina de Especificaciones.

3. Verificación y Validación

LA verificación y validación es un proceso cuyos objetivos son: detectar la mayor cantidad posible de defectos de los distintos productos de software (documentos de requerimientos, diseño y código, entre otros) y evaluar la calidad de estos productos.

Una forma de dividir las técnicas de verificación y validación es en dinámicas y estáticas. El Framework sólo contiene actividades del tipo dinámico. El motivo de esta decisión fue presentado en el capítulo 2.

El testing es la forma dinámica de la verificación y la validación. El testing no es solamente ejecutar casos de prueba y comparar los resultados esperados con los resultados obtenidos; el testing, entre otras cosas, se debe planificar, definir un proceso y establecer la adecuación del mismo. Las siguientes actividades se consideran importantes dentro del testing para construir productos de calidad. Además, entendemos que estas son aplicables en el desarrollo académico de software.

El siguiente Cuadro muestra las actividades dentro del área de Verificación y Validación.

Actividades de Verificación y Validación	
Act. 8	Planificación del Testing
Act. 9	Establecimiento de Adecuación
Act. 10	Test de Aceptación
Act. 11	Testing Funcional del Sistema
Act. 12	Testing de los Casos de Uso
Act. 13	Testing de Componentes
Act. 14	Testing Intraclases
Act. 15	Testing de Métodos
Act. 16	Test de Regresión

A continuación se describe cada una de estas actividades.

Como se vio en el capítulo 2, la planificación del testing es excesiva y engorrosa para el desarrollo académico de software. Sin embargo, hemos notado que en los proyectos del CSI las actividades de testing se realizan sin tener noción de qué es lo más importante para ser testeado, cuándo es el mejor momento para testearlo y con qué cubrimiento realizarlo. Entonces, parece razonable definir una actividad de planificación *liviana* del testing para el desarrollo académico de software.

Esta actividad sólo pretende definir qué testear, cuándo hacerlo y con qué cubrimiento realizarlo. De esta forma, se logra tener un marco para realizar el resto de las actividades de testing. Este marco permite ganar en productividad en el testing y dedicar mayores recursos a las partes críticas del sistema.

A continuación se presenta la especificación de la actividad *Planificación del Testing*.

Act. 8 - Planificación del Testing

Descripción

Esta actividad sirve para planificar el testing, definiendo pocas pero importantes cosas para hacer el testing más efectivo. Como mínimo se debe planificar: qué testear, cuándo testear y relacionar las distintas partes del sistema con el documento de cubrimientos que es salida de la actividad *Establecimiento de Adecuación*.

Qué testear
El documento de Plan de Testing que es salida de esta actividad debe tener definida una sección donde se identifica qué partes del software serán testeadas.

Cuándo testear
Cada parte del software debe tener definido cuándo va a ser testeada.

Relación con el documento "Documento de Cubrimientos"
Para cada parte del software se especifica qué tipo de cubrimiento (definido en el Documento de Cubrimientos) se va a adoptar. De esta manera se pueden definir criterios más exigentes para partes más críticas y/o importantes del software y menos exigentes para partes menos críticas. Esto genera una mayor productividad debido a que los esfuerzos de testing están racionalizados.

Entrada: Toda la documentación que se tenga del producto.

Salida: Plan de Testing.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Mejorar la productividad debido a la planificación adecuada del testing.
Obj. 2	Mejorar las partes críticas y/o importantes del software.
Obj. 3	Disminuir la cantidad de fallas en el sistema.

Entendemos que mejorando las actividades de testing en los proyectos del CSI se pueden mejorar sustancialmente distintas propiedades de calidad de los productos. El mayor problema detectado es que en el momento de realizar el testing los desarrolladores no saben qué casos seleccionar. Creemos que estableciendo criterios de cubrimiento se mejora esta situación. En el momento de seleccionar nuevos casos de prueba se eligen aquellos que aproximen más, al conjunto de casos de prueba resultante, a cumplir con el cubrimiento. De esta forma, se logra tener un conjunto de casos de prueba adecuado para el producto según el cubrimiento escogido.

Por los motivos expresados es que proponemos una actividad específica para definir los distintos cubrimientos. La actividad es llamada *Establecimiento de Adecuación*. Esta actividad pretende ordenar el posterior diseño de los casos de prueba logrando mejores resultados en el testing.

A continuación se presenta la especificación de esta actividad.

Act. 9 - Establecimiento de Adecuación

Descripción

Un criterio de cubrimiento indica una parte a cubrir, tanto sea de código como de especificación de requerimientos o como de algún otro producto del desarrollo. Un ejemplo de criterio de cubrimiento de código es: Se desea que al ejecutar todos los casos de prueba, todas las decisiones del código se hagan al menos una vez verdaderas y al menos una vez falsas. Este criterio es conocido como *Criterio de Decisión*. Un ejemplo de criterio de cubrimiento de requerimientos es: Se tendrá al menos un caso de prueba para cada requerimiento especificado.
Una unidad, componente, sistema, etc., se dice que está adecuadamente testeado si ha sido cubierto con el criterio seleccionado. Se puede medir cuánto se ha cubierto del criterio. Por ejemplo, al ejecutar los casos de prueba se cubrieron 75 % de las líneas de código del sistema.
Se divide el tipo de cubrimiento según el tipo de test. En este caso se divide en los siguientes tipos de cubrimiento: Cubrimiento de Especificación de Requerimientos, Cubrimiento de Casos de Uso, Cubrimiento de Componentes, Cubrimiento de Clases, Cubrimiento de Métodos. El documento que se genera como salida de esta actividad debe dividirse en estas cinco clases de cubrimiento, pudiendo estar varias de ellas vacías. Los tipos de cubrimiento se detallan en el cuadro siguiente.
Si existe un Plan de Testing, este documento puede tener referencias al mismo y viceversa. El Plan de Testing permite tener un ajuste en los criterios y no ser estrictos en utilizar los mismos criterios para todo el sistema. Si existe dicho plan se pueden establecer distintos criterios de cubrimiento y en el Plan de Testing mencionar qué partes del sistema utilizan un criterio y qué partes utilizan otro. Por ejemplo, el plan de testing puede decir que los métodos de las clases que están en la componente de nombre *"crítica"* deben tener un cubrimiento estructural de condición múltiple. Para el resto de los métodos no se establece criterio. La explicación del método de cubrimiento se encuentra en el Documento de Cubrimiento (salida de esta actividad). El Documento de Cubrimiento puede ser parte del Plan de Testing o ser un documento independiente.

Entrada: Dependiendo del tipo de cubrimiento a definir la entrada puede ser:
Documento de Requerimientos.
Documento de Casos de Uso.
Documento de Especificación de Componentes y de Especificación de Clases con especificación de los métodos.

Salida: Documento de Cubrimientos.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Tener una meta a alcanzar en el momento de diseñar los casos de prueba.
Obj. 3	Aumentar las probabilidades de obtener un producto de mejor calidad.
Obj. 4	Conocer cuánto va a estar testeado o evaluado el producto al finalizar el desarrollo.

Cubrimientos

Cubrimiento de especificación de requerimientos

El menor cubrimiento aceptable de una especificación de requerimientos es al menos tener un caso de prueba para cada requerimiento especificado en el alcance del producto. Se pueden especificar criterios más fuertes como: *para cada funcionalidad se estudiarán las posibles particiones en clases de equivalencia tanto de la entrada como de la salida y se generarán casos de prueba para cubrirlas todas.*

Cubrimiento de casos de uso

El mínimo cubrimiento aceptable es el cubrimiento del escenario del flujo normal; aunque el mismo es considerado muy pobre. Un criterio más fuerte es cubrir todos los escenarios posibles del caso de uso. Uno aún más fino es cubrir todos los escenarios posibles del caso de uso y considerar tanto los datos de entrada como de salida de cada escenario y utilizar la técnica de partición en clases de equivalencia.

Cubrimiento de componentes

Es sumamente dependiente de la forma en la cual fue especificada la componente. Se deben cubrir todas las funcionalidades que brinda la misma. Si además se cuenta con un diagrama de estados se pueden usar criterios que dependen del cubrimiento de dicho diagrama.

Cubrimiento de clases

Este tipo de cubrimiento se usa cuando la clase es estado-dependiente e influye el orden en el cual son ejecutados los métodos de la misma. El menor cubrimiento aceptable es ejecutar todas las transiciones válidas desde cada estado que existe en el diagrama de estados de la clase.

Cubrimiento de métodos

Tomar a cada método como una función y exigir criterios de cubrimiento de código y funcionales. El mínimo criterio es aquel donde el método cumple con todas sus funcionalidades y se cumple con el criterio de decisión.

Entendemos que el testing funcional del sistema es necesario para cualquier tipo de desarrollo de software. Creemos que agregar actividades que realicen este tipo de test ayuda a mejorar la calidad del producto final.

Proponemos tres actividades para este tipo de test:

La actividad *Test de Aceptación* busca comprometer al cliente con el desarrollo y aprobar el producto. La realización de esta actividad en distintas iteraciones logra un mayor entendimiento entre clientes y desarrolladores. Además, brinda la posibilidad de conocer mejor con qué casos se va a aceptar el producto. Esto último también ayuda a entender qué funcionalidades son las más críticas para el cliente.

Las actividades *Testing Funcional del Sistema* y *Testing de los Casos de Uso* son las actividades de testing de sistema propuestas en el Framework para que lleven adelante los desarrolladores. La primera está basada en el documento de requerimientos y la segunda en el documento de casos de uso. Entendemos que proponer estas actividades termina por disminuir la cantidad de defectos remanentes en el sistema así como también la severidad de los mismos. Por otro lado, al tener diseñados estos casos de prueba se aumenta la capacidad de modificación y mantenimiento del sistema.

Estas actividades buscan ordenar el testing a nivel de sistema. Para las mismas se define un cubrimiento mínimo que se debe alcanzar. De esta manera, se logra que los desarrolladores generen mejores casos de prueba y que el funcionamiento del producto sea conocido y esté controlado.

A continuación se presentan las especificaciones de estas tres actividades.

Act. 10 - Test de Aceptación

Descripción

Este test es el que realiza el cliente para aceptar o no el producto desarrollado. El conjunto de casos de prueba a testear debe ser definido, en lo posible, al comienzo del desarrollo, a medida que se van realizando los requerimientos y los casos de uso. Estos casos deben ser definidos por el cliente y no por el equipo de desarrollo. Al igual que todas las actividades esta también es dependiente del proceso de desarrollo y no tiene que ser imaginada como una actividad que se ejecuta una única vez al finalizar el proceso de desarrollo.
Sugerencias:

- Establecer al menos un caso de prueba para cada requerimiento o escenario de caso de uso que esté en el alcance del sistema.
- Considerar las prioridades establecidas en los requerimientos (o casos de uso) y realizar más casos de prueba para esos requerimientos (o casos de uso)
- El cliente puede no conocer buenas técnicas para derivar casos de prueba, entonces se recomienda pedir ayuda al Responsable de Calidad y/o basarse en un subconjunto de la salida de las actividades Testing Funcional del Sistema y/o Testing de los Casos de Uso.

Esta actividad se divide básicamente en dos etapas, una es la definición de los casos de prueba y la otra es la ejecución de los mismos. Normalmente ambas las realiza el cliente.
Realizar esta actividad también ayuda a comprender mejor los requerimientos. Es común que al realizar los casos de prueba de aceptación surjan problemas no detectados durante el relevamiento de los requerimientos así como también requerimientos nuevos.

Entrada: Documento de Requerimientos y Documento de Casos de Uso.

Salida: Documento de Casos de Prueba de Aceptación.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Entender el producto que se debe construir. Ayuda a detectar errores en los requerimientos.
Obj. 2	Mejorar la mantenibilidad del producto.
Obj. 3	Mejorar la modificabilidad del producto.
Obj. 4	Reducir la probabilidad de desarrollar un producto no adecuado.
Obj. 5	Conocer cómo se va a evaluar el producto de forma temprana.
Obj. 6	Ayudar al propio cliente a saber qué es lo que quiere desarrollar.
Obj. 7	Disminuir la cantidad de fallas en el sistema.
Obj. 8	Asegurar que ciertos casos pueden ser ejecutados sin detectar fallas.
Obj. 9	Brindar mayor seguridad en el momento de hacer demos de los productos

Act. 11 - Testing Funcional del Sistema

Descripción

Dado el Documento de Requerimientos se desarrollan casos de prueba para mostrar qué requerimientos no funcionan correctamente. Esta actividad está constituida por tres etapas:

1. Generación de los casos de prueba (en los niveles de calidad 3 y superiores depende de la actividad *Establecimiento de Adecuación* en lo referente a Cubrimiento de Especificación de Requerimientos). Estos casos de prueba deben ser documentados indicando el requerimiento a testear, datos de entrada y resultado esperado.
2. Ejecución de los casos de prueba. A partir del Nivel 4 se usan estos tests, o un subconjunto de los mismos, como Test de Regresión. La ejecución es obligatoriamente automática a partir del Nivel 5 de calidad.
3. Registro de errores. A partir del Nivel 3 es obligatorio llevar un registro de errores. El mismo debe incluir qué casos de prueba produjeron el error y cuántas veces se ha ejecutado el caso y mantenido el error. A partir del Nivel 5 es obligatorio el uso de una herramienta CASE para el seguimiento (tracking) de fallas.

Sugerencias:

- Definir los casos de prueba en paralelo con la definición de los requerimientos.
- Considerar las prioridades y riesgos establecidos para los requerimientos y establecer una mayor cantidad de casos de prueba para los de mayor prioridad y/o riesgo.

Entrada: Documento de Requerimientos.

Salida: Documento de Casos de Prueba de Especificación de Requerimientos.
Documento de Reporte de Errores o Herramienta CASE con los mismos (su obligatoriedad depende del Nivel de Calidad).
Automatización de los Casos de Prueba (su obligatoriedad depende del Nivel de Calidad).

Objetivos de calidad y ganancia al realizar la misma

Obj. 1 Detectar fallas en la ejecución de cada requerimiento.
Obj. 2 Reducir el número de fallas por requerimiento.
Obj. 3 Mejorar la mantenibilidad del producto (aumenta si los casos de prueba están automatizados).
Obj. 4 Mejorar la modificabilidad del producto (aumenta si los casos de prueba están automatizados).
Obj. 5 Asegurar que ciertos casos pueden ser ejecutados sin detectar fallas.

Act. 12 - Testing de los Casos de Uso

Descripción

Dado el Documento de Casos de Uso se desarrollan casos de prueba para mostrar qué casos de uso no funcionan correctamente. Esta actividad está constituida por tres etapas:

1. Generación de los casos de prueba (en los Niveles de Calidad 3 y superiores depende de la actividad *Establecimiento de Adecuación* en lo referente a Cubrimiento de Casos de Uso). Estos casos de prueba deben ser documentados indicando en cada uno: Caso de Uso a testear, escenario, datos de entrada y resultado esperado.
2. Ejecución de los casos de prueba. A partir del Nivel 4 se usan estos tests (o un subconjunto de los mismos) como Test de Regresión. La ejecución es obligatoriamente automática a partir del Nivel de Calidad 5.
3. Registro de errores. A partir del Nivel 3 es obligatorio llevar un registro de errores. El mismo debe incluir qué casos de prueba produjeron el error y cuántas veces se ha ejecutado el caso y mantenido el error. A partir del Nivel 5 es obligatorio el uso de una herramienta CASE para el seguimiento (tracking) de fallas.

Sugerencias:

- Definir los casos de prueba en paralelo con la definición de los casos de uso.
- Considerar las prioridades y riesgos establecidos para los casos de uso y establecer una mayor cantidad de casos de prueba para los de mayor prioridad y/o riesgo.

Entrada: Documento de Casos de Uso.

Salida: Documento de Casos de Prueba de Casos de Uso.
Documento de Reporte de Errores o Herramienta CASE con los mismos (su obligatoriedad depende del Nivel de Calidad).
Automatización de los Casos de Prueba (su obligatoriedad depende del Nivel de Calidad).

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Detectar fallas en la ejecución de los escenarios de los casos de uso.
Obj. 2	Reducir el número de fallas por caso de uso identificado.
Obj. 3	Mejorar la mantenibilidad del producto.
Obj. 4	Mejorar la modificabilidad del producto.
Obj. 5	Asegurar que ciertos casos pueden ser ejecutados sin detectar fallas.

La cantidad de fallas y la severidad de las mismas en los productos actuales del CSI es grande. Lamentablemente, los desarrolladores del grupo no realizan otro testing que el testing a nivel de sistema. Entendemos que realizar testing en otros dos niveles, de componentes y de clases, ayuda a obtener productos de mejor calidad, con menos cantidad de defectos y defectos menos severos.

En este entendido proponemos tres actividades: *Testing de Componentes*, *Testing Intraclases* y *Testing de Métodos*.

La primera nos va a permitir conocer y controlar el sistema a nivel de componentes. Con esta actividad se busca tener componentes con una cantidad de defectos mucho menor que la que se tiene actualmente. De esta manera, se logra tener menos defectos a nivel global y aumentar la capacidad de mantenimiento y modificabilidad del producto de software.

Las actividades *Testing Intraclases* y *Testing de Métodos* buscan los mismos objetivos que la actividad a nivel de componentes, pero a nivel de clases y de métodos respectivamente. Realizando estas dos actividades, decrece aún más la cantidad y severidad de defectos del sistema.

A continuación se presentan las especificaciones de estas tres actividades.

Act. 13 - Testing de Componentes

Descripción

Mediante este tipo de test se busca encontrar los defectos de las componentes del sistema. El conjunto de casos de prueba debe ser definido, en lo posible, al definir las componentes y las interfaces entre las mismas.
Sugerencias:

- Establecer al menos un caso de prueba para cada uno de los métodos que se pueden invocar desde fuera de la componente.
- Generar casos de prueba usando la estrategia all round-trip path si la componente es estado-dependiente.

Dado el Documento de Especificación de Componentes se desarrollan casos de prueba para mostrar qué funcionalidades de la componente no funcionan correctamente. Esta actividad está constituida por tres etapas:

1. Generación de los casos de prueba. En los Niveles de Calidad 5 y superiores depende de la actividad Establecimiento de Cubrimiento en lo referente a Cubrimiento de Componentes. Estos casos de prueba deben ser documentados.
2. Ejecución de los casos de prueba. A partir del Nivel 5 se usan estos tests, o un subconjunto de los mismos, como Test de Regresión.
3. Registro de errores. A partir del Nivel 5 es obligatorio llevar un registro de errores. El mismo debe incluir qué casos de prueba produjeron el error y cuántas veces se ha ejecutado el caso y mantenido el error. A partir del Nivel 5 es obligatorio el uso de una herramienta CASE para el seguimiento (tracking) de fallas.

Entrada: Documento de Especificación de Componentes.

Salida: Documento de Casos de Prueba de Componentes.
Documento de Reporte de Errores o Herramienta CASE con los mismos (su obligatoriedad depende del Nivel de Calidad).
Automatización de los Casos de Prueba (su obligatoriedad depende del Nivel de Calidad).

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Detectar fallas en la ejecución de las funcionalidades de las componentes.
Obj. 2	Reducir el número de fallas por componente.
Obj. 3	Mejorar la mantenibilidad del producto.
Obj. 4	Mejorar la modificabilidad del producto.
Obj. 5	Asegurar que ciertos casos pueden ser ejecutados sin detectar fallas.

Act. 14 - Testing Intraclases

Descripción

Mediante este tipo de test se busca encontrar los defectos de las clases del sistema. El conjunto de casos de prueba debe ser definido, en lo posible, al definir las clases y las interfaces entre las mismas.
Sugerencias:

- Establecer al menos un caso de prueba para cada uno de los métodos que se pueden invocar desde fuera de la clase.
- Generar casos de prueba usando la estrategia all round-trip path si la clase es estado-dependiente.
- Testear la interacción entre clases de la misma componente. La interacción entre clases de distintas componentes debe ser testeada en la actividad Testing de Componentes.

Dado el Documento de Especificación de Clases se desarrollan casos de prueba para mostrar qué funcionalidades de la clase no funcionan correctamente. Esta actividad está constituida por tres etapas:

1. Generación de los casos de prueba. En el Nivel de Calidad 6 depende de la actividad *Establecimiento de Adecuación* en lo referente a Cubrimiento de Clases.
2. Ejecución de los casos de prueba. En el Nivel 6 se usan estos tests, o un subconjunto de los mismos, como Test de Regresión.
3. Registro de errores. En el Nivel 6 es obligatorio llevar un registro de errores. El mismo debe incluir qué casos de prueba produjeron el error y cuántas veces se ha ejecutado el caso y mantenido el error. En el Nivel 6 es obligatorio el uso de una herramienta CASE para el seguimiento (tracking) de fallas.

Entrada: Documento de Especificación de Clases.

Salida: Documento de Casos de Prueba de Clases.
Documento de Reporte de Errores o Herramienta CASE con los mismos (su obligatoriedad depende del Nivel de Calidad).
Automatización de los Casos de Prueba (su obligatoriedad depende del Nivel de Calidad).

Objetivos de calidad y ganancia al realizar la misma

Obj. 1 Detectar fallas en la ejecución de las funcionalidades de las clases.
Obj. 2 Reducir el número de fallas por clase.
Obj. 3 Mejorar la mantenibilidad del producto.
Obj. 4 Mejorar la modificabilidad del producto.
Obj. 5 Asegurar que ciertos casos pueden ser ejecutados sin detectar fallas.

Act. 15 - Testing de Métodos

Descripción

Esta actividad complementa la actividad Testing Intraclase. En la actividad Testing Intraclase se testean los métodos que pueden ser accedidos desde fuera de la clase. Esta actividad complementa estos casos de prueba con pruebas para los métodos de uso privado de la clase. Además se exige que las pruebas cumplan al menos con el criterio de cubrimiento de decisión. Dado el Documento de Especificación de Clases que contiene las especificaciones de los métodos se desarrollan casos de prueba para mostrar qué métodos no funcionan correctamente. Esta actividad está constituida por tres etapas:

1. Generación de los casos de prueba. En los Niveles de Calidad 6 y superiores depende de la actividad Establecimiento de Cubrimiento en lo referente a Cubrimiento de Métodos.
2. Ejecución de los casos de prueba. En el Nivel 6 se usan estos tests, o un subconjunto de los mismos, como Test de Regresión.
3. Registro de errores. En el Nivel 6 es obligatorio llevar un registro de errores, el mismo debe incluir qué casos de prueba produjeron el error y cuántas veces se ha ejecutado el caso y mantenido el error. En el Nivel 6 es obligatorio el uso de una herramienta CASE para el seguimiento (tracking) de fallas.

Entrada: Documento de Especificación de Clases con la especificación de los métodos.

Salida: Documento de Casos de Prueba de Métodos.
Documento de Reporte de Errores o Herramienta CASE con los mismos (su obligatoriedad depende del Nivel de Calidad).
Automatización de los Casos de Prueba (su obligatoriedad depende del Nivel de Calidad).

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Detectar fallas en la ejecución de las funcionalidades de los métodos.
Obj. 2	Reducir el número de fallas por método.
Obj. 3	Mejorar la mantenibilidad del producto.
Obj. 4	Mejorar la modificabilidad del producto.

Los problemas más graves en el desarrollo en el CSI son la gran cantidad de defectos de los productos y no conocer dónde se encuentran las fallas. Las actividades de testing presentadas buscan mejorar esta situación.

Sin embargo, durante la corrección de defectos es normal que se introduzcan nuevos defectos. Usualmente este tipo de problemas se corrige mediante los tests de regresión. Este tipo de actividad es sencilla de realizar, y entendemos que es un gran aporte para el desarrollo académico de software ya que normalmente no se hace. Por esto, y también para lograr un mayor control sobre el producto, es que se propone la actividad *Test de Regresión*. Esta actividad propone la realización de test de regresión a nivel unitario, de componentes y de sistema.

A continuación se presenta la especificación de esta actividad.

Act. 16 - Test de Regresión

Descripción

Este test es el que se realiza cuando hay cambios en el sistema y sirve para verificar que lo que estaba funcionando no dejó de funcionar. Es común que al intentar solucionar un defecto en el sistema se introduzcan nuevos defectos, esta práctica intenta evitarlo. El test de regresión se puede realizar para cualquiera de los niveles de test: unitario, componentes, sistema.
Es importante definir cuál es el conjunto de casos de prueba a utilizar para las pruebas de regresión (todos o un subconjunto). Existen técnicas para determinar qué casos seleccionar para las pruebas de regresión e inclusive cuáles de esos casos ejecutar dependiendo de dónde se realizó un cambio en el código. Cualquiera de estas técnicas es compleja. En el caso de tener las pruebas automatizadas se recomienda usarlas todas como tests de regresión. En el caso de no tenerlas automatizadas se recomienda elegir un subconjunto representativo que asegure que la funcionalidad se sigue comportando correctamente luego de los cambios.
El momento indicado de realizar un test de regresión depende del tipo de test. De todas maneras, siempre que se tenga algo estable, sea un método, una clase, una componente o una funcionalidad conviene realizar test de regresión.

Entrada: Documento con los Casos de Prueba dependiendo del nivel al cual se vaya a ejecutar los test de regresión (sistema, componentes, clases y/o métodos).

Salida: Informe de Fallas. Se puede realizar un documento de cuales son los test a usar en el test de regresión si estos no son todos los casos de prueba construidos.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Mantener estable el sistema.
Obj. 2	Aumentar la confiabilidad en el sistema.
Obj. 3	Brindar mayor confianza a los propios desarrolladores para realizar cambios.
Obj. 4	Aumentar significativamente la calidad del producto final mediante la práctica común del test de regresión.

Estas son todas las actividades que se definen para la disciplina de Verificación y Validación.

4. Métricas y Medidas

Distintas métricas se han propuesto para todo el desarrollo de software, entre otras, métricas del diseño y métricas del testing. Es importante, en el momento de escoger una métrica, establecer o conocer el motivo por el cual se escoge la misma. También es importante, en el momento de tomar las mediciones, saber cuáles serán las acciones a tomar de acuerdo a los resultados obtenidos. En pocas palabras, no sirve medir simplemente por el arte de medir.

En el Framework se usan las distintas medidas para mantener el producto bajo control e intentar mantener la calidad esperada.

Las métricas propuestas en este trabajo se pueden separar en métricas de diseño y métricas de testing. Se propone una única actividad dentro de las métricas de diseño.

Actividad de Medidas de Diseño
Act. 17 Midiendo el Diseño

Al evaluar diseños de desarrollos anteriores en el CSI, se nota que son correctos. Sin embargo, sabemos que el diseño no es revisados por los desarrolladores. También sabemos que los desarrolladores en el CSI no tienen la disciplina necesaria para realizar algún tipo de revisión sobre el diseño. Por esto proponemos una actividad que tome ciertas mediciones del diseño de forma automática: *Midiendo el Diseño*.

Entendemos que una actividad que se realiza de forma automática va a ser llevada a cabo por el grupo de desarrollo. Esta busca obtener datos de forma rápida y no tediosa sobre el diseño, y marcar aquellas mediciones que no son *"buenas"*. En el momento de obtener estas medidas no satisfactorias, los desarrolladores revisarán el diseño en busca de soluciones.

Con una actividad de este estilo buscamos que los desarrolladores tengan un mayor control sobre el diseño y hagan, al menos, una revisión *liviana* del mismo. Esta actividad se puede ejecutar rápidamente con lo cual no resta productividad al grupo de desarrollo. Además, esta actividad termina provocando una mejora en el diseño propuesto.

A continuación se presenta la especificación de la actividad.

Act. 17 - Midiendo el Diseño

Descripción

Esta actividad contribuye a la mejora del diseño de la aplicación. Para realizar esta actividad se deben definir las métricas que van a ser usadas. Para cada métrica se definen los valores para los cuales se considera que el diseño es correcto.
Sugerencias:

- Siempre que las mediciones realizadas den valores que no están dentro del rango aceptado se deberá chequear el diseño de la aplicación. Es importante que en el momento de obtener medidas con valores fuera del rango aceptado no se considere que el diseño está mal sino que se evalúe el mismo mediante una revisión.
- Usar una herramienta CASE sobre el diseño o directamente sobre el código fuente.

También se sugiere usar al menos las siguientes métricas:

Ca: Afferent Couplings: El número de clases fuera de la componente analizada que depende de las clases dentro de la componente.
Ce: Efferent Couplings: El número de clases dentro de la componente analizada que depende de clases fuera de la componente.
D: Distance: —(A+I-1)/2—: La distancia perpendicular de una componente a la secuencia principal. Siendo A e I las siguientes:
A: Abstractness: (cantidad de clases abstractas en la componente)/(cantidad total de clases en la componente).
I: Instability: (Ce/(Ca+Ce)).

Las métricas sugeridas se pueden estudiar en el artículo "OO Design Quality Metrics - An Analysis of Dependencies" de Robert Martin [Mar94].

Entrada: Alguna forma de expresar el diseño.

Salida: Documento de Métricas a Usar.
Documento de Medidas Obtenidas para Cada Métrica.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Mejorar el diseño de la aplicación.
Obj. 2	Aumentar la posibilidad de reuso.
Obj. 3	Mejorar la mantenibilidad y modificabilidad del producto.
Obj. 4	Facilitar el testing del sistema.

Se proponen dos actividades dentro de las métricas de testing:

Actividades de Métricas de Testing	
Act. 18	Cantidad de Fallas/Casos de Prueba
Act. 19	Midiendo el Cubrimiento

Proponemos dos actividades de forma de conocer qué tan efectivo es el testing que se realiza. Una de estas actividades es simplemente dividir la cantidad de casos de prueba que fallaron sobre la cantidad total de casos de prueba. La actividad la llamamos *Cantidad de Fallas/Casos de Prueba* y se aplica a nivel unitario, de componentes y de sistema.

Esta actividad es trivial y su ejecución no requiere ningún esfuerzo extra. El resultado da una idea de qué tan bien se está realizando el testing y de la calidad del producto. Con este resultado se puede saber dónde hay que realizar más testing y qué partes del producto son de baja calidad.

La otra actividad complementa a la actividad *Establecimiento de Adecuación*. La actividad *Midiendo el Cubrimiento* busca conocer el cubrimiento real alcanzado durante el testing. Este es otro punto de vista acerca de qué tan bien se está realizando el testing. Conocer el cubrimiento a alcanzar y el cubrimiento alcanzado hasta el momento permite construir casos de prueba que mejoren el cubrimiento. De esta manera, se logran construir conjuntos de casos de prueba de mayor calidad.

Act. 18 - Cantidad de Fallas/Casos de Prueba

Descripción

Esta medida indica la densidad de errores que se puede encontrar en el sistema. La forma de medir es trivial. Luego de ejecutar cada caso de prueba se cuentan los casos que fallaron y se hace la división entre los casos que fallaron y los casos que se ejecutaron. Esta métrica tiene valores entre 0 y 1.
Se recomienda separar los resultados en los distintos niveles de testing. Se sugiere la siguiente división:

- CF/CP para requerimientos del sistema. Sólo se consideran los casos de prueba que se desprenden de las actividades Testing de los Casos de Uso y Testing Funcional del Sistema.
- CF/CP para componentes. Sólo se consideran los casos de prueba que se desprenden de la actividad Testing de Componentes.
- CF/CP para clases y métodos. Sólo se consideran los casos de prueba que se desprenden de las actividades Testing Intraclases y Testing de Métodos.

Cabe aclarar que los casos de la actividad Test de Regresión siempre corresponden a alguna de las actividades antes descritas.

Entrada: *No aplica*

Salida: Documento de Cantidad de Fallas indicando la densidad de las fallas.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1 Tener una idea de cuántas fallas quedan en el sistema.
Obj. 2 Tener una idea de la confiabilidad del sistema.

Act. 19 - Midiendo el Cubrimiento

Descripción

Para medir cuánto se alcanza del criterio de cubrimiento hay que tener establecido a qué nivel se realiza la medición. Se divide de la misma forma que está dividida la actividad *Establecimiento de Adecuación*.

Cubrimiento de Especificación de Requerimientos
En este caso, para medir cuánto se logra completar del criterio de cubrimiento se analiza cada caso de prueba y se establece qué requerimiento está testeando y con qué partición de las entradas y salidas. Se recomienda el uso de una planilla o de una herramienta CASE para mantener actualizada la medición. Es suficiente tener una traza desde los casos de prueba a los requerimientos. Cada vez que se agrega o elimina un caso de prueba simplemente se debe cambiar el documento de trazas y ver si afecta a la completitud alcanzada del criterio de cubrimiento.

Cubrimiento de Casos de Uso
Similar al punto anterior.

Cubrimiento de Componentes
Similar a los puntos anteriores. Se puede medir el cubrimiento a nivel de diagrama de estados de la componente (si este existe).

Cubrimiento de Clases
Similar al punto anterior.

Cubrimiento de Métodos
En este caso se recomienda fuertemente usar una herramienta que mida el cubrimiento de código alcanzado luego de ejecutar los casos de prueba. Además, se debe analizar si los casos de prueba cubren la funcionalidad del método. Una planilla con el cubrimiento usado y el porcentaje alcanzado del mismo es suficiente para mantener un seguimiento durante el desarrollo e intentar agregar casos de prueba que disminuyan la distancia entre el criterio elegido y el cubrimiento logrado realmente.

Entrada: Depende del cubrimiento a medir.

Salida: Planillas de Mediciones de Cubrimiento.

Objetivos de calidad y ganancia al realizar la misma

Obj. 1	Obtener una idea clara y objetiva de cuánto se ha logrado testear.
Obj. 2	Aumentar la posibilidad de realizar mejores casos de prueba ya que sólo se va a considerar aquellos que ayuden a alcanzar el criterio establecido.
Obj. 3	Disminuir las fallas. Esto se logra porque al elegir mejores casos de prueba se encuentran más errores.

Estas son todas las actividades que se definen para la disciplina de Métricas y Medidas.

5. Niveles de Calidad

EL Framework propuesto agrupa las actividades en Niveles de Calidad. Los niveles van del uno al seis y a medida que se aumenta de Nivel de Calidad aumenta la probabilidad de que el producto a construir sea mejor en ciertas propiedades de calidad. Se recomienda como mínimo el Nivel de Calidad tres, ya que el uno y el dos se definen únicamente para ordenar ideas y aseguran poco respecto a la calidad del producto final.

Cada Nivel tiene asociado el tipo de prototipo que se espera conseguir al ejecutar las actividades. Para el Nivel 1 simplemente se espera conocer qué es lo que se tiene que construir. El Nivel 2 permite obtener un prototipo funcional. Un prototipo con fallas conocidas se puede obtener en el Nivel 3. En el Nivel 4 el prototipo está controlado; los cambios no deberían provocar que dejen de ejecutar correctamente las funcionalidades que antes estaban correctas. El Nivel 5 tiene controlado el prototipo también a nivel de componentes y se conocen las fallas de cada una. Este prototipo es extensible, modificable y sus componentes pueden ser reutilizadas en otros desarrollos. En el Nivel 6 el prototipo que se espera debe tener menor cantidad de defectos que si se desarrollara en el nivel anterior. Además, se conoce el comportamiento del mismo a nivel de cada clase que compone el sistema.

Cada Nivel se describe de la siguiente forma: Se presentan las actividades que se deben realizar para ese Nivel. Se numeran restricciones sobre las actividades propuestas. Se describe un conjunto de observaciones para el Nivel. Se listan los documentos que se deben realizar obligatoriamente y que no son salida de ninguna de las actividades propuestas.[2] Por último se menciona qué se debería conseguir[3] si se desarrolla un producto en el Nivel de Calidad descrito.

A continuación se presentan los seis Niveles de Calidad.

[2]Estos documentos se producen en actividades triviales y por eso no se describen en el Framework. Un ejemplo de estos documentos es el Manual de Usuario.

[3]Lo que se debería conseguir no es algo que se pueda asegurar solamente por desarrollar en un Nivel ya que existen múltiples factores que influyen en la calidad de un producto.

Nivel de Calidad 1

Descripción
En este nivel se desconoce el funcionamiento del producto de software final. Se tiene especificado de forma correcta qué es lo que desea el cliente. No se recomienda elegir este nivel.

Actividades
- Relevar requerimientos y priorizarlos
- Validar requerimientos
- Generar casos de uso y priorizarlos
- Validar casos de uso

Restricciones sobre las actividades
No aplica

Observaciones
No aplica

Documentos fuera de las actividades propuestas
- Manual de usuario.
- Manual técnico.

En este Nivel se debe conseguir
- El producto especificado es el que el cliente necesita.
- El producto está especificado de forma que facilita el mantenimiento y la modificabilidad del mismo.
- El producto está especificado de forma tal de poder brindar las funcionalidades más importantes ya que estas fueron priorizadas.

Nivel de Calidad 2

Descripción
En este nivel se conoce el funcionamiento del producto final en casos determinados por el cliente. Variar los datos respecto a los probados o cambiar el orden de ejecución puede dar resultados inesperados. Es un nivel de alto riesgo pero permite realizar demos (usando únicamente los casos especificados en la prueba de aceptación).

Actividades
- Relevar requerimientos y priorizarlos
- Validar requerimientos
- Generar casos de uso y priorizarlos
- Validar casos de uso
- Testing de los casos de uso
- Testing funcional del sistema
- Test de aceptación

Restricciones sobre las actividades
- En la ejecución de los test de aceptación no se deben detectar fallas.

Observaciones
No aplica

Documentos fuera de las actividades propuestas
- Manual de usuario.
- Manual técnico.

En este Nivel se debe conseguir
- Mayor probabilidad de construir el producto deseado que en el nivel 1.
- Mayor facilidad para mantener y/o modificar el producto que en el nivel 1.
- Menor cantidad de fallas que en el nivel 1 ya que obliga a tener casos de prueba que han sido ejecutados correctamente.
- Un producto en el cual determinadas funcionalidades, aplicando determinados datos, se pueden ejecutar correctamente.
- Todo lo que se consigue en el Nivel 1.

Nivel de Calidad 3

Descripción

En este nivel se conocen fallas del sistema y cuánta funcionalidad del producto ha sido cubierta por los casos de prueba. Entonces, se pueden preparar futuras versiones sabiendo de antemano qué es lo que hay que corregir. También se tiene una idea de qué queda sin testear. Este nivel permite realizar demos con cierta confianza ya que se conoce las funcionalidades que son correctas y aquellas que aún no se pueden ejecutar. Esto permite *moverse* por el sistema con mayor flexibilidad que en el nivel 2.

Actividades

- Relevar requerimientos y priorizarlos
- Validar requerimientos
- Generar casos de uso y priorizarlos
- Validar casos de uso
- Testing de los casos de uso
- Testing funcional del sistema
- Test de aceptación
- Establecimiento de cubrimiento (de requerimientos y/o casos de uso)
- Cantidad de fallas/Casos de prueba (de requerimientos y/o casos de uso)
- Midiendo el cubrimiento (de requerimientos y/o casos de uso)

Restricciones sobre las actividades

- En la ejecución de los test de aceptación no se deben detectar fallas.
- Se debe llevar un registro de las fallas encontradas a nivel de sistema y estas deben ser reproducibles. Restricciones sobre las actividades Testing de los Casos de Uso y Testing Funcional del Sistema.

Observaciones

No aplica

Documentos fuera de las actividades propuestas

- Manual de usuario.
- Manual técnico.
- Estándar de codificación (se debe definir y seguir).

Continúa en la página siguiente...

Nivel de Calidad 3 - (Continuación)

En este Nivel se debe conseguir

- Conocer las fallas del sistema y poder reproducirlas.
- Un aumento en la capacidad de corrección de las faltas en el código si las pruebas están automatizadas. Esto aumenta la modificabilidad y la mantenibilidad.
- Conocer la cantidad de fallas relacionada con la cantidad de casos de prueba funcionales ejecutados.
- Tener una idea acerca de la confiabilidad del sistema. Esta, entre otras cosas, depende del criterio de cubrimiento establecido y el nivel de adecuación que se alcanzó del mismo.
- Todo lo que se consigue en el Nivel 2.

Nivel de Calidad 4

Descripción

Este nivel es el mínimo recomendado para realizar demos y para establecer un prototipo que asegure un cumplimiento satisfactorio de las funcionalidades. Esto se obtiene debido a la exigencia sobre la cantidad de fallas máximas que el sistema puede tener en la ejecución de los casos de prueba y a la exigencia en el criterio de cubrimiento. El Plan de Pruebas permite que el criterio de cubrimiento sea flexible y pueda ser más *liviano* para funcionalidades de poca criticidad o más *pesado* en caso contrario. Con esto último se logra no perder productividad.

Actividades

- Relevar requerimientos y priorizarlos
- Validar requerimientos
- Generar casos de uso y priorizarlos
- Validar casos de uso
- Establecimiento de cubrimiento (de requerimientos y/o casos de uso)
- Planificación del testing (de requerimientos y/o casos de uso)
- Testing de los casos de uso
- Testing funcional del sistema
- Test de aceptación
- Test de regresión
- Cantidad de fallas/Casos de prueba (de requerimientos y/o casos de uso)
- Midiendo el cubrimiento (de requerimientos y/o casos de uso)

Continúa en la página siguiente...

Nivel de Calidad 4 - (Continuación)

Restricciones sobre las actividades

- En la ejecución de los test de aceptación no se deben detectar fallas.
- Se debe llevar un registro de las fallas encontradas a nivel de sistema y estas deben ser reproducibles. Restricciones sobre las actividades Testing de los Casos de Uso y Testing Funcional del Sistema.
- La medida Cantidad de Fallas/Casos de Prueba debe ser $\leq$ 0,1 a nivel de requerimientos y casos de uso.
- La medida que surge de Midiendo el Cubrimiento debe ser de 100 % a nivel de requerimientos y casos de uso.
- El Criterio de Cubrimiento de Especificación de Requerimientos debe tener al menos un caso de prueba para cada requerimiento en el alcance y se debe utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Casos de Uso debe tener al menos un caso de prueba para cada escenario posible y utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.

Observaciones

- Es válido determinar distintos cubrimientos para requerimientos o casos de uso dependiendo de la prioridad y/o riesgo de cada uno; esto se especifica en el plan de test.
- Se mantienen bajo control de configuración tanto el código como los casos de prueba.

Documentos fuera de las actividades propuestas

- Manual de usuario.
- Manual técnico.
- Estándar de codificación (se debe definir y seguir).

En este Nivel se debe conseguir

- Las funcionalidades del sistema se comportan como se espera en situaciones determinadas por el criterio de adecuación.
- Un control sobre la inyección de nuevos errores (a nivel de la funcionalidad del sistema) usando tests de regresión.
- Estabilizar el sistema a nivel funcional.
- Testear mejor las partes criticas del sistema debido a la Planificación del testing.
- Todo lo que se consigue en el Nivel 3.

Nivel de Calidad 5

Descripción
Se recomienda desarrollar como mínimo en este nivel si el producto se va a continuar en años subsiguientes o si se requiere un número bajo de defectos. La cantidad de defectos disminuye considerablemente respecto al nivel anterior debido al control a nivel de componentes de software. En este nivel se da la posibilidad de que el sistema y sus componentes sean reutilizables.

Actividades

- Relevar requerimientos y priorizarlos
- Validar requerimientos
- Generar casos de uso y priorizarlos
- Validar casos de uso
- Especificación de componentes
- Establecimiento de cubrimiento (de requerimientos, casos de uso y componentes)
- Planificación del testing (de requerimientos, casos de uso y componentes)
- Testing de los casos de uso
- Testing funcional del sistema
- Test de aceptación
- Test de regresión
- Testing de componentes
- Cantidad de fallas/Casos de prueba (de requerimientos, casos de uso y componentes)
- Midiendo el cubrimiento (de requerimientos, casos de uso y componentes)
- Midiendo el diseño

Restricciones sobre las actividades

- En la ejecución de los test de aceptación no se deben detectar fallas.
- La medida Cantidad de Fallas/Casos de Prueba debe ser $\leq 0{,}1$ a nivel de requerimientos y casos de uso.
- La medida Cantidad de Fallas/Casos de Prueba debe ser $\leq 0{,}05$ a nivel de componentes.
- La medida que surge de Midiendo el Cubrimiento debe ser de 100 % a nivel de requerimientos, casos de uso y componentes.
- El Criterio de Cubrimiento de Especificación de Requerimientos debe tener al menos un caso de prueba para cada requerimiento en el alcance y se debe utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Casos de Uso debe tener al menos un caso de prueba para cada escenario posible y utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.

Continúa en la página siguiente...

Nivel de Calidad 5 - (Continuación)

Restricciones sobre las actividades - (Continuación)

- El Criterio de Cubrimiento de Componentes debe tener al menos un caso de prueba para cada funcionalidad que brindan las componentes y utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Componentes debe cumplir con el criterio *All round-trip path* si la componente es estado-dependiente.
- Las pruebas que surgen de las actividades Testing Funcional del Sistema y Testing de los Casos de Uso deben estar automatizadas de alguna forma; las pruebas a nivel del sistema quedan automatizadas.
- Se debe llevar un registro de todas las fallas encontradas y su estado. Se recomienda el uso de una herramienta CASE.
- El diseño de la aplicación se debe realizar con una herramienta CASE.

Observaciones

- Es válido determinar distintos cubrimientos para requerimientos o casos de uso dependiendo de la prioridad y/o riesgo de cada uno; esto se especifica en el plan de test.
- Se mantienen bajo control de configuración el documento de requerimientos, el documento de diseño, el código y los casos de prueba.
- Es válido determinar distintos cubrimientos para las componentes dependiendo de la prioridad y/o riesgo de cada una; esto se especifica en el plan de test.

Documentos fuera de las actividades propuestas

- Manual de usuario.
- Manual técnico.
- Estándar de codificación (se debe definir y seguir).

En este Nivel se debe conseguir

- Lograr un aumento en la calidad global del producto.
- Mayor posibilidad de reutilización de componentes y del sistema entero.
- Lograr que el producto tenga una mayor modificabilidad y mantenibilidad.
- Menor cantidad de fallas que en el nivel anterior.
- Todo lo que se consigue en el Nivel 4.

Nivel de Calidad 6

Descripción
En este nivel el producto que se desarrolla puede ser continuado en años posteriores con facilidad. Se recomienda este nivel para productos multi-versión. Si este nivel se usa para el desarrollo de una única versión la disminución en la productividad es notoria.

Actividades

- Relevar requerimientos y priorizarlos
- Validar requerimientos
- Generar casos de uso y priorizarlos
- Validar casos de uso
- Especificación de componentes
- Especificación de clases
- Especificación de métodos
- Establecimiento de cubrimiento (de requerimientos, casos de uso, componentes, clases y métodos)
- Planificación del testing (de requerimientos, casos de uso, componentes y clases)
- Testing de los casos de uso
- Testing funcional del sistema
- Test de aceptación
- Test de regresión
- Testing de componentes
- Testing intraclases
- Testing de métodos
- Cantidad de fallas/Casos de prueba (de requerimientos, casos de uso, componentes, clases y métodos)
- Midiendo el cubrimiento (de requerimientos, casos de uso, componentes, clases y métodos)
- Midiendo el diseño

Restricciones sobre las actividades

- En la ejecución de los test de aceptación no se deben detectar fallas.
- La medida Cantidad de Fallas/Casos de Prueba debe ser $\leq$ 0,1 a nivel de requerimientos y casos de uso.
- La medida Cantidad de Fallas/Casos de Prueba debe ser $\leq$ 0,05 a nivel de componentes.
- La medida Cantidad de Fallas/Casos de Prueba debe ser = 0 a nivel de Clases y Métodos.
- La medida que surge de Midiendo el Cubrimiento debe ser de 100 % a nivel de requerimientos y casos de uso, componentes, clases y métodos..

Continúa en la página siguiente...

Nivel de Calidad 6 - (Continuación)

Restricciones sobre las actividades - (Continuación)

- El Criterio de Cubrimiento de Especificación de Requerimientos debe tener al menos un caso de prueba para cada requerimiento en el alcance y se debe utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Casos de Uso debe tener al menos un caso de prueba para cada escenario posible y utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Componentes (Clases) debe tener al menos un caso de prueba para cada funcionalidad que brindan las componentes y utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Componentes (Clases) debe cumplir con el criterio *All round-trip path* si la componente (Clase) es estado-dependiente.
- El Criterio de Cubrimiento de Métodos debe tener al menos un caso de prueba para cada funcionalidad que brindan los métodos y utilizar técnicas de partición en clases de equivalencia para la obtención de los datos de prueba.
- El Criterio de Cubrimiento de Métodos debe cumplir como mínimo con el criterio de decisión.
- Todos los casos de prueba que surgen de las distintas actividades de testing y que se pueden automatizar deben estar automatizados.
- Se debe llevar un registro de todas las fallas encontradas y su estado. Se recomienda el uso de una herramienta CASE.
- El diseño de la aplicación se debe realizar con una herramienta CASE.

Observaciones

- Es válido determinar distintos cubrimientos para requerimientos o casos de uso dependiendo de la prioridad y/o riesgo de cada uno; esto se especifica en el plan de test.
- Se mantienen bajo control de configuración el documento de requerimientos, el documento de diseño, el código y los casos de prueba.
- Es válido determinar distintos cubrimientos para las componentes dependiendo de la prioridad y/o riesgo de cada una; esto se especifica en el plan de test.

Documentos fuera de las actividades propuestas

- Manual de usuario.
- Manual técnico.
- Estándar de codificación (se debe definir y seguir).

Continúa en la página siguiente...

Nivel de Calidad 6 - (Continuación)
En este Nivel se debe conseguir
- Mayor posibilidad de reuso a todo nivel de las partes del software. - Lograr que el producto tenga una mayor modificabilidad y mantenibilidad. - Reducir considerablemente la cantidad de fallas respecto al nivel anterior. - Un producto sencillo de continuar en una siguiente versión. - Todo lo que se consigue en el Nivel 5.

6. Conclusiones

SE propuso un Framework para el desarrollo académico de software que consta de diecinueve Actividades, seis Niveles de Calidad y un Responsable de Calidad. Dicho Framework contempla las características particulares del desarrollo académico de software que fueron derivadas a partir de la experiencia de desarrollo de software en el grupo CSI.

En la sección 1 del capítulo *"Estado del Arte"* se estudiaron distintos modelos, procesos y propuestas para el desarrollo de productos de software de calidad. Estas propuestas tienen como centro de atención el desarrollo de software en la industria. En este tipo de desarrollo el mayor problema detectado radica en aspectos de gestión; por ejemplo, CMM busca primero, mediante las KPA's del Nivel 2, solucionar los problemas de gestión de los proyectos, y luego atiende otro tipo de problemas.

Por otro lado, también detectamos en el estado del arte de las disciplinas consideradas que la preocupación es el software para la industria y no el software desarrollado en un ambiente académico. Entonces, la mayoría de las actividades propuestas parecen exageradas para el desarrollo académico de software; por ejemplo, actividades relacionadas con la ingeniería de requerimientos orientada a *goals*, actividades de uso de la etnografía para la ingeniería de requerimientos y cubrimientos basados en el flujo de datos. Las disciplinas que se consideraron son: Ingeniería de Requerimientos, Verificación y Validación, y Métricas y Medidas de Diseño. Estas se presentan y analizan en las secciones 2, 3 y 4 del capítulo *"Estado del Arte"*

Debido a esto, nuestra propuesta descarta diversas actividades, estudia los problemas reales del desarrollo en el CSI y propone actividades puramente técnicas y realizables en el contexto de estudio de forma de controlar el desarrollo de software. Este control busca solucionar los problemas específicos detectados en el CSI.

Varias de las actividades propuestas pueden ser comparables con las actividades de algunos procesos de software en las disciplinas de requerimientos y verificación. Sin embargo, en estos procesos, por ejemplo el RUP, estas disciplinas son sub-procesos dentro del proceso de desarrollo. En nuestro caso, no presentamos procesos para las disciplinas sino

que preferimos presentar actividades *sueltas*. Entendemos que en el desarrollo académico de software no es bueno seguir procesos predefinidos por lo que el proceso de desarrollo a seguir queda librado a los desarrolladores.

Considerando el conjunto completo de actividades propuestas, es notorio que la gran diferencia con las propuestas estudiadas radica en haber eliminado todo aspecto de gestión. Esta diferencia es enorme en el momento de desarrollar software con el Framework o con alguna de las propuestas estudiadas. Entendemos que con nuestra propuesta se logra ganar productividad en el contexto académico. Creemos que en el desarrollo académico de software, con una propuesta que contemple actividades de gestión, se perdería en productividad y no cambiaría la calidad del producto desarrollado.

Un punto oscuro del Framework son las restricciones sobre la actividad *Cantidad de Fallas/Casos de Prueba*. A partir de cierto Nivel la misma tiene que ser menor que 0,1 para las pruebas de sistema y menor que 0,05 para las pruebas de componentes. Estos números fueron elegidos sin mucho criterio y se tomaron básicamente para que los grupos de desarrollo tuvieran una meta a alcanzar. Corresponde ajustar estos valores con datos que se puedan obtener a partir de la aplicación del Framework.

Cabe aclarar que la restricción que aparece en el nivel 6, *La medida Cantidad de Fallas/Casos de Prueba debe ser igual a cero a nivel de Clases y Métodos*, es razonable y no se tiene que ajustar. Al realizar test de clases y de métodos se espera que no se encuentren fallas o, en caso contrario, se corrigen los defectos y se vuelve a realizar el test. Esto se debe a lo pequeña y controlable que es la unidad bajo test.

Consideramos que el Framework propuesto es adecuado para el desarrollo académico de software y en particular para el desarrollo de software dentro del grupo CSI.

Estudio de Campo 4

Desde Abril de 2004 a Junio de 2005 se ejecutan once proyectos que ponen en práctica el Framework. Todos estos proyectos son Proyectos de Grado de la Facultad de Ingeniería. Este estudio de campo es necesario para validar y ajustar nuestra propuesta para la mejora de la calidad en el desarrollo académico de software.

Todos los grupos de desarrollo son de dos o tres personas y es usado el lenguaje de programación Java. Cada proyecto tiene al menos un cliente y en total se cuenta con cuatro clientes para los once proyectos. Estos tienen distinta duración. Uno de los proyectos termina en Diciembre de 2004 y los diez restantes finalizan entre Abril y Junio de 2005. Siete de los proyectos trabajan en el Nivel de Calidad 5 y los cuatro restantes trabajan en el Nivel de Calidad 3.

Mediante el estudio de la documentación entregada y reuniones, tanto con los desarrolladores como con los clientes, se analizan los resultados de cada una de las actividades propuestas. Entendemos que, en general, las actividades se realizaron de forma correcta y que estas contribuyeron a mejorar la calidad del producto final.

Se realiza una encuesta a los desarrolladores y a los clientes de forma de conocer su opinión sobre la aplicación del Framework. El resultado es alentador, todos los desarrolladores y todos los clientes opinan que el Framework logró la mejora de la calidad de los productos desarrollados.

Contenido

1. Proyectos y Productos

El Framework se aplica en once proyectos desde Abril de 2004 hasta Junio de 2005. Los mismos son Proyectos de Grado de la Facultad de Ingeniería. Esta sección busca presentar los distintos proyectos, describiendo la cantidad de personas involucradas en cada uno y el producto que se espera como resultado.

Este estudio de campo es necesario para validar y ajustar nuestra propuesta. Durante el mismo, los grupos de desarrollo cuentan con la especificación completa del Framework en un sito web diseñado para tal fin y tienen a disposición al Responsable de Calidad.

Todos los grupos de desarrollo son de dos o tres personas y es usado el lenguaje de programación Java. Los proyectos tienen distinta duración. Uno de los proyectos termina en Diciembre de 2004 y los diez restantes finalizan entre Abril y Junio de 2005. Siete de los proyectos trabajan en el Nivel de Calidad 5 y los cuatro restantes trabajan en el Nivel de Calidad 3.

Cada proyecto tiene al menos un cliente y en total se cuenta con cuatro clientes para los once proyectos. Nombraremos a los clientes *"Cliente n"*, siendo n un número de 1 a 4. Nombraremos a los proyectos *"Proyecto n"*, siendo n un número de 1 a 11. El Cuadro 4.1 muestra en qué proyectos trabajaron los clientes.

CLIENTES	**PROYECTOS**										
	P. 1	**P. 2**	**P. 3**	**P. 4**	**P. 5**	**P. 6**	**P. 7**	**P. 8**	**P. 9**	**P. 10**	**P. 11**
Cliente 1	X										
Cliente 2		X	X	X	X	X					
Cliente 3							X	X	X	X	X
Cliente 4				X							

Tabla 4.1: Relación de Clientes y Proyectos

A continuación se describen los once proyectos:[1]

En el proyecto número 1, *Análisis e Implementación de un Toolkit para Testing de Performance*[AG05], se estudia el desarrollo de una herramienta para un laboratorio de testing independiente[CES] que funciona dentro de la Facultad de Ingeniería. La herramienta que se construye en este proyecto es un Toolkit para testing de performance que permite a los usuarios realizar la actividad del testing de forma más eficaz, eficiente y simple. El producto resultado de este proyecto se llama Performance Center. Este permite tener un único repositorio de datos que facilite el análisis de los mismos y la posibilidad de tener un control centralizado en escenarios de testing en los que participen varias herramientas diferentes. Este producto se maneja a través de una interfaz web y utiliza una base de datos relacional para almacenar la información recolectada. Esta herramienta permite la creación y ejecución de pruebas en diferentes herramientas así como la trans-

[1]Estas descripciones son tomadas de los informes finales de cada uno de los proyectos.

formación de los resultados obtenidos a un formato estándar. Este proyecto es realizado por dos personas y tiene como exigencia un Nivel de Calidad 3.

El proyecto número 2, *Gestión de la Producción Intelectual de una Universidad*[BK05], busca desarrollar un sistema que ayude en la tarea de gestionar y administrar la producción intelectual de una universidad. Este proyecto está integrado con el Proyecto Fénix de la Universidad de Lisboa y comprende la implementación de un producto fácilmente extensible que cumpla con los siguientes objetivos: centralizar el conocimiento aportado por las diferentes producciones, mejorar la gestión relacionada con los formularios administrativos y facilitar la consulta de material producido dentro de la Universidad. El producto de software se basa en una arquitectura en capas y está construido sobre una ontología especificada en OWL que describe la realidad del Instituto de Computación de la Facultad de Ingeniería. Se usa el API Jena para el acceso a ontologías, OJB para el acceso a base de datos y STRUTS como framework de presentación. Este proyecto es desarrollado por dos personas y el Nivel de Calidad que se exige es 5.

El proyecto *Estudio de la Transformación de Contenido Educacional Representado Mediante LOM, hacia SCORM*[CP05], proyecto número 3, estudia la posibilidad de transformar contenido existente en LOM a SCORM. Analiza posibles soluciones a los distintos problemas detectados y hace hincapié en los puntos críticos de decisión. Presenta métodos para transformar contenido LOM genérico, así como un análisis detallado de una transformación específica a partir de un perfil de aplicación de LOM. Con esto se sientan las bases para que las distintas plataformas que tomen a SCORM como modelo base, no desaprovechen el contenido existente. Este proyecto es desarrollado por dos personas y el Nivel de Calidad exigido es 3.

En el proyecto número 4, *Extracción e Integración de Información en una Arquitectura de Web Warehousing*[GGV04], se desarrolla una herramienta para la extracción de información desde la Web. Luego esa información es integrada y almacenada en un Data Warehouse. La extracción se basa en la estructura de la página Web, considerando algunas de las distintas formas de presentación. El método de integración de la información se basa en la confiabilidad de la fuente de la cual fue extraída. El proyecto es llevado adelante por un grupo de tres personas y tiene como exigencia un Nivel de Calidad 3.

El proyecto número 5, *Nivel de Calidad de la Adaptación de Cursos en Ambientes de Aprendizaje Electrónico*[PSS05], tiene como propósito construir un prototipo capaz de permitir adaptar en forma automática, para diferentes estudiantes, el contenido de cursos a distancia, abarcando aspectos concernientes a la presentación así como también a las características conceptuales del curso. El trabajo se realiza en el contexto del proyecto AdaptWeb Multicultural y del Proyecto Red EduCa. Tanto el perfil de estudiante como el contenido de curso son descritos mediante ontologías expresadas en los lenguajes DAML + OIL y OWL respectivamente. Para el procesamiento de estas ontologías se utilizó el framework Jena. Este proyecto fue desarrollado por un grupo de tres personas y usaron el Nivel de Calidad 3.

El proyecto número 6, *Codificación de Conocimientos Médicos*[BG05], se encarga de desarrollar una componente para el producto EviDoctor de la empresa EviMed. Este componente busca documentos en el Servicio de Información del producto EviDoctor

y devuelve los documentos relevantes para un escenario clínico dado. Este escenario se define por la historia clínica de un paciente y el médico tratante. Para lograr el objetivo de la búsqueda es necesario clasificar documentos según su contenido. Este producto es desarrollado por dos personas. El Nivel de Calidad exigido es 5 en todo lo que involucra al motor de búsqueda y 3 en el resto de las componentes.

Otro de los proyectos desarrollado en el marco de esta investigación, el proyecto número 7, se llama *Mecanismo de Ayuda Contextual para Sistemas de Información Federados*[GRS05]. El objetivo de este proyecto es desarrollar utilitarios y componentes para proveer funcionalidades de ayuda en línea a aplicaciones que integran Sistemas de Información Federados a través de la Web, en particular a las aplicaciones del sistema Link-All[LA]. Dicha ayuda debe ser sensible al contexto de ejecución, el cual depende entre otras cosas, de las características del usuario y de la aplicación y funcionalidad que se están ejecutando. Entre el Sistema de Ayuda y las aplicaciones debe existir muy bajo acoplamiento, debiendo ser esta situación totalmente transparente para los usuarios, ya que ellos podrían manipular la ayuda de todas las aplicaciones como si fuera una sola, permitiendo así que los usuarios puedan conocer y trabajar en todas las posibilidades que le ofrece el Sistema de Información. La solución final se divide en tres grandes partes. En primer lugar, un framework, que es la parte central del sistema ya que contiene toda la lógica del negocio, luego, una implementación de este para el sistema Link-All, y por último, un conjunto de utilitarios que proveen una serie de servicios fundamentales para el correcto funcionamiento del framework. Se buscó respetar fuertemente las características de extensibilidad, flexibilidad y adaptabilidad, logrando así un Mecanismo de Ayuda Contextual configurable también a otros Sistemas de Información Federados. En este proyecto trabajan tres personas. El Nivel de Calidad que se usa en este proyecto es 3 para algunas funcionalidades y 5 para otras.

Dos grupos diferentes de desarrollo realizaron proyectos muy similares. Uno fue el proyecto número 8, *Utilitarios para Deployment de Aplicaciones en una Plataforma Basada en Servicios sobre J2EE*[DM05]. El producto desarrollado es un framework de deployment genérico, basado en dos premisas fundamentales: proveer mecanismos de extensión mediante el uso de plugins y simplificar al máximo su personalización. Definida una plataforma de aplicación, el framework provee la infraestructura sobre la cual elaborar la solución de deployment para la misma. Es este proyecto trabajan dos personas. Los niveles de calidad son 3 y 5 y dependen de la funcionalidad.

El otro fue el proyecto número 9, *Distribución y Deployment de Aplicaciones en una Plataforma Basada en Servicios*[CCD05]. El mismo se enmarca en el contexto del proyecto Link-All. El objetivo específico de este proyecto es el de definir e implementar un mecanismo genérico para la distribución y deployment de las aplicaciones que forman parte de la plataforma Link-All. El mayor desafío que presenta es el de definir un mecanismo lo suficientemente flexible y extensible capaz de instalar aplicaciones implementadas en distintas tecnologías. Además, debe contar con una interface Web simple y fácil de utilizar por usuarios con pocos conocimientos técnicos. La solución a la cual se llegó permite extender la plataforma desarrollando pequeños componentes llamados deployers. Estos deployers son los encargados de agregar nuevas funcionalidades capaces de satisfacer nuevos requerimientos. La implementación está basada en estándares de la industria como

XML y SOAP. También se utiliza un conjunto de frameworks de bajo costo de licenciamiento que solucionan problemas específicos en la implementación. En este proyecto trabajan tres personas y al igual que el anterior tiene Nivel de Calidad 3 para algunas funcionalidades y 5 para otras.

Otros dos grupos atacan un mismo problema. El primero de ellos es el proyecto número 10, *Federador Link-All*[AA05]. Este proyecto brinda los mecanismos de federación necesarios a un proyecto de mayor porte denominado Link-All, el cual consiste en Sistemas de Información Federados a través de la Web. La solución propuesta comprende una aplicación integrada a la plataforma Link-All, que mediante la comunicación entre servidores utilizando Web Services, posibilita la gestión y monitoreo de la federación de nodos, servicios y datos. Entre sus funcionalidades se destaca la capacidad de administrar la configuración de cada servidor de la red Link-All, rutear el acceso a datos y servicios en base a una identificación lógica y visualizar el estado actual de la federación tanto de forma local como remota. Esta solución se desarrolla sobre la plataforma J2EE y se basa en estándares abiertos de la industria. Fue construida basándose en patrones de diseño y utilizando herramientas de código abierto para así lograr una aplicación flexible, fácil de mantener y extender. Este proyecto es llevado a cabo por dos personas. Para algunas funcionalidades el Nivel de Calidad es 5 y para otras es 3.

El otro proyecto es el número 11, *Utilitarios para Federar Servidores Link-All Basado en J2EE*[lR05]. Este proyecto está enmarcado en el proyecto Link-All. Esta es una plataforma basada en servicios y datos federados a través del web. Se busca desarrollar utilitarios y componentes para el ambiente Link-All que permitan federar varios servidores con dicha plataforma, dando transparencia sobre la ubicación física de los mismos. En el dominio del problema se maneja el concepto de Nodo Link-All (servidor de aplicaciones J2EE). Existe una jerarquización entre los nodos de la federación, distinguiéndose varios Nodos y un Nodo Soporte. El Nodo Soporte es el que se encuentra actualizado en todo momento acerca del estado de los recursos de la federación (topología), tomándoselo como referente para la actualización de la información en el resto de los nodos. En cada nodo se alojan una o más comunidades de usuarios, organizados en grupos. Los servicios brindados en la federación se corresponden con aplicaciones deployadas desde el Nodo Soporte. Los mismos son provistos en tres categorías distintas: pública, privada y compartida. A los servicios privados acceden únicamente los usuarios de la comunidad proveedora; a los compartidos tienen acceso los usuarios de la comunidad proveedora y todo aquel grupo extra comunidad que tenga permiso de uso, y a los públicos se accede libremente dentro de la red. En lo que respecta a los datos, si bien todas las comunidades tienen acceso a una base de datos (con esquema común a la federación) que se encuentra en el propio nodo, también pueden consultar datos de comunidades remotas con previa obtención de permiso de acceso para lectura. La solución propone un mecanismo para rutear las diferentes solicitudes de servicios y datos a través de la federación de servidores de forma que el usuario esté ajeno a la ubicación de dichos recursos. Para implementar el sistema de manejo de permisos sobre los recursos compartidos se diseñó un protocolo de comunicación entre el Nodo Soporte y el resto de los nodos. Para sustentar esta propuesta se brindan funcionalidades administrativas para la federación de comunidades y servicios, así como mecanismos para mantener actualizada la información de la federación. Las tecnologías

empleadas en la solución comprenden las utilizadas comúnmente en plataformas J2EE: EJB (Enterprise Java Beans) para la comunicación remota, Servlets y JSP (Java Server Pages) para la construcción de las interfaces gráficas y JMS (Java Message Service) para el envío de solicitudes de acceso a recursos. Este proyecto es llevado a cabo por dos personas. Para algunas funcionalidades el Nivel de Calidad es 5 y para otras es 3.

2. Resultados por Actividad

En esta sección se estudia el resultado obtenido de la ejecución de cada actividad por los grupos de desarrollo que siguieron el Framework. Los resultados son similares para todos los proyectos estudiados por lo que se pueden sacar conclusiones de validez general.

Mediante el estudio de la documentación entregada y reuniones tanto con los desarrolladores como con los clientes se analizan los resultados de cada una de las actividades propuestas. En general entendemos que las actividades se realizaron de forma correcta y que estas contribuyeron a mejorar la calidad del producto final. A continuación se presentan los resultados divididos por actividad del Framework.

Relevar Requerimientos y Priorizarlos

El relevamiento de los requerimientos se realizó de forma correcta por los grupos. Según los clientes los documentos de especificación de requerimientos, en general, están completos y son adecuados.

La priorización de los requerimientos, en la mayoría de los casos, contó con un conjunto de requerimientos de prioridad alta, otro de prioridad media y otro de baja prioridad. Se puede entender el nivel de alta prioridad como aquellos requerimientos que son imprescindibles para el producto y/o que determinan la arquitectura del mismo. El nivel medio indica requerimientos que sería muy bueno tener en el producto pero que no son imprescindibles. La prioridad baja se usa para requerimientos que surgen del proceso de obtención de requerimientos pero que no deben ser necesariamente implementados en la versión que va a ser desarrollada del producto. Estas tres clasificaciones se usaron de forma correcta.

La definición del alcance fue acertada en casi todos los proyectos. Por *acertada* entendemos dos cosas: Primero, el cliente queda conforme con un producto que contenga todas las funcionalidades definidas en el alcance. Segundo, el grupo de desarrollo puede implementar, en tiempo y forma, un producto que abarque todo el alcance.

Validar Requerimientos

Las validaciones se realizaron generalmente de dos formas: lectura de los documentos

de requerimientos (la menos frecuente), y lectura conjunta con los desarrolladores normalmente apoyados por casos de uso (se vuelve sobre este punto en la actividad *Validar Casos de Uso*). La validación de los requerimientos por parte de los clientes no resultó una tarea fácil. Los clientes dentro del desarrollo académico de software son, normalmente, investigadores que no han trabajado como clientes en proyectos de la industria. A pesar de esto, las validaciones resultaron provechosas y sirvieron tanto a los clientes como a los desarrolladores para aclarar ideas y poder entenderse.

Generar Casos de Uso y Priorizarlos

La generación y prorización de los casos de uso tuvo el mismo tipo de comportamiento que la actividad *Relevar los Requerimientos y Priorizarlos* por lo que no se agrega ningún comentario.

Validar Casos de Uso

La validación de los casos de uso fue sustancialmente más provechosa que la actividad *Validación de los Requerimientos*. Fue más fácil trabajar con los casos de uso debido a que estos y los escenarios que los componen representan formas en las cuales el sistema es usado. Los requerimientos aislados y a veces descontextualizados generaron problemas para ser validados, en cambio, los casos de uso resultaron más propicios para lograr su validación. Esta validación contribuyó fuertemente al entendimiento del sistema, a lograr definir el alcance y a establecer un acuerdo entre desarrolladores y clientes.

Especificación de Componentes

La especificación de componentes normalmente se realizó mediante un diagrama de estados cuyas transiciones son los métodos que pueden ser invocados desde fuera de la componente. Este tipo de especificación no fue fácil de realizar para los desarrolladores. Sin embargo, los resultados fueron muy buenos, consistiendo estos en especificaciones completas de componentes centrales del producto desarrollado.

Planificación del Testing

Esta actividad exige definir qué testear, cuándo testear y definir la relación entre las partes del software y los tipos de cubrimiento que se encuentran en el Documento de Cubrimientos. Para el nivel 3 esta actividad no está definida. Para el nivel 5, el otro Nivel en el cual trabajaron grupos de desarrollo, se debe planificar el testing a nivel de requerimientos, casos de uso y componentes. La correcta ejecución de esta actividad solamente fue llevada a cabo por un grupo de desarrollo. La particularidad de este grupo, y motivo por el cual creemos que llevó correctamente a cabo esta actividad, es que su producto era más sencillo de implementar que los otros productos. No realizar esta actividad provocó una baja en la productividad en momentos de ejecutar el testing ya que no se tenía el foco correcto de qué, cuándo y con qué extensión testear. También trajo aparejado un testing igualitario para todas las partes del software construido, lo cual no es conveniente si tenemos en cuenta que distintas partes tienen distinta criticidad.

Establecimiento de Adecuación

Esta actividad propone generar distintos criterios de adecuación según el tipo de testing, a saber, de requerimientos, de casos de uso, de componentes, de clases y de métodos. Esta fue sólo llevada a cabo correctamente por el grupo que realizó de forma correcta la actividad *Planificación del Testing*. Los grupos que no realizaron esta actividad no tenían criterios para terminar el testing, por lo cual surgieron problemas menores que pudo resolver el Responsable de Calidad.

Test de Aceptación

Esta actividad es llevada a cabo por los clientes. En el Framework se entiende que la misma debe ser realizada varias veces, junto con la liberación de versiones, a lo largo del proyecto. Lamentablemente esto no fue así, y en la mayoría de los proyectos se realizó una única vez al final del proyecto. De todas formas esta actividad sirvió para notar la buena calidad de los productos construidos.

Testing Funcional del Sistema

Esta actividad y la actividad *Testing de Casos de Uso* fueron las actividades de testing que mejor se realizaron durante el estudio de campo. Debido a esto y a pesar de no tener correctamente establecidos los criterios de adecuación entendemos que los productos fueron correctamente verificados a nivel del sistema.

Testing de los Casos de Uso

Esta actividad se realizó de forma correcta. El conocimiento previo que se tiene en este tipo de testing[2] fue fundamental en el momento de realizar los casos de prueba para los Casos de Uso.

Testing de Componentes

Esta actividad dio buenos resultados para dos de los siete proyectos que desarrollaron en el Nivel de Calidad 5. Estos proyectos tienen la particularidad de tener muy bien definida la arquitectura del sistema. Además, se detecta que los integrantes de estos dos grupos son conscientes de los beneficios que puede reportar el testing a nivel de componentes. Debido a que sólo dos grupos, de los siete que tenían que realizar esta actividad, obtuvieron buenos resultados, es que entendemos que este testing no fue realizado de forma adecuada.

Test de Regresión

El test de regresión a nivel de sistema (test de requerimientos y test de casos de uso) funcionó bien. No fueron automatizados los test pero de todas maneras se logró que las cosas que funcionaban bien en una liberación siguieran funcionando bien en las siguientes. Aún mejor funcionó a nivel de componentes para los dos grupos que realizaron de forma

[2]Este conocimiento se adquiere en asignaturas que deben ser cursadas antes de poder realizar el Proyecto de Grado.

correcta la actividad *Testing de Componentes*. Estos test fueron automatizados con la herramienta JUnit y esto fue muy provechoso al ejecutar los test de regresión.

Cantidad de Fallas/Casos de Prueba

Esta actividad permitió tener una idea de si el proceso de testing estaba mejorando en la detección de defectos y si aumentaba la confiabilidad del sistema. La actividad fue realizada por pocos grupos.

Midiendo el Cubrimiento

Esta actividad fue realizada correctamente por el único grupo que realizó la actividad *Establecimiento de Adecuación*. A este grupo le sirvió para determinar, de forma confiable, cuándo se alcanzó el objetivo establecido.

Midiendo el Diseño

Todos los grupos que desarrollaron en el Nivel de Calidad 5 realizaron esta actividad. Los grupos supieron aprovechar de distintas maneras los resultados obtenidos al ejecutar esta actividad. Algunos grupos usaron, además de las medidas sugeridas, las métricas MOOSE [CK94]. Entendemos que esta actividad produjo mejoras importantes en los diseños.

Otras Actividades

Las actividades Especificación de Clases, Especificación de Métodos, Testing Intraclases y Testing de Métodos corresponden al sexto Nivel de Calidad. Ninguno de los proyectos trabajó en ese Nivel de Calidad por lo que no se realizó ninguna de estas actividades en el estudio de campo.

En general, creemos que las actividades propuestas se adecuaron al contexto del desarrollo académico de software. Estas fueron aplicadas por los distintos grupos sin tener mayores dificultades. Las mismas son lo suficientemente flexibles como para que los desarrolladores puedan usar distintos métodos al aplicarlas. A la vez, están lo suficientemente definidas como para lograr los objetivos de calidad que cada una pretende alcanzar. Estas dos cualidades de las actividades permitieron que los desarrolladores se sintieran cómodos al aplicarlas y que distintos grupos de desarrolladores pudieran alcanzar los mismos objetivos de calidad aplicando métodos distintos.

3. Evaluación

ESTA sección presenta la evaluación del uso del Framework en los proyectos del grupo CSI que comenzaron en el año 2004. La misma se realiza mediante encuestas a los clientes y a los desarrolladores.

3.1. Evaluación de los Clientes

La pregunta más relevante para conocer el resultado de la experiencia de usar el Framework es: ¿Mejoró la calidad del software desarrollado respecto a proyectos de años anteriores? Esta pregunta tiene una alta subjetividad y sólo la pueden contestar los clientes debido a que los desarrolladores desconocen los proyectos de años anteriores. La encuesta que se realiza a los clientes se presenta en el Cuadro 4.2.

1.	¿Cree que las actividades planteadas ayudaron a obtener un producto de mejor calidad que los obtenidos en años anteriores? Justifique su respuesta. Intente indicar en qué propiedades de calidad mejoró el producto (menos defectos y menos severos, mayor mantenibilidad adaptativa y correctiva, facilidad de instalación, facilidad de comprensión del producto obtenido, etc.)
2.	¿Cree que las actividades planteadas pueden servir para mejorar la calidad de cualquier proyecto de grado propuesto por el CSI (que sea de desarrollo de un producto de software)?
3.	¿Cree que las actividades planteadas hicieron que el alcance del producto fuera menor? En caso de ser así, ¿podría estimar cuánto disminuyó el alcance del proyecto siendo 100 % el alcance que se pensaba obtener?
4.	¿El sitio web le fue útil? ¿Lo usó?
5.	¿Necesitó ayuda del Responsable de Calidad?
6.	¿Es necesario que esta persona exista o sólo con el sitio web (mejorado) sería suficiente?
7.	¿Está conforme con la calidad del producto obtenido?

Tabla 4.2: Encuesta Realizada a los Clientes

Las personas que fueron clientes de varios proyectos no encontraron diferencias en los resultados de la aplicación del Framework en cada uno de estos. Por lo tanto se presenta una única encuesta por cada cliente. Los Cuadros 4.3, 4.4 y 4.5 muestran las respuestas de los clientes para cada pregunta.

Pregunta	Cliente Uno	Cliente Dos	Cliente Tres	Cliente Cuatro
¿Cree que las actividades planteadas ayudaron a obtener un producto de mejor calidad que los obtenidos en años anteriores?	Si. La diferencia es notoria.	Si	Si	Si
¿En qué propiedades de calidad mejoró el producto?	Sobre todo se tienen menos defectos y menor severidad de los mismos.	Se mejoró en todas las áreas. Destaco que se notaron menos defectos y mayor facilidad de adaptación y corrección.	No especifica.	Menos errores. Se conocen qué cosas están bien porque fueron metódicamente controladas.
¿Cree que las actividades planteadas pueden servir para mejorar la calidad de cualquier proyecto de grado propuesto por el CSI?	Si	Si	Si	Si. Tal vez en algunos casos hay que hacer un poco de esfuerzo para adaptar las actividades a tipos de proyectos particulares.
¿Cree que las actividades planteadas hicieron que el alcance del producto fuera menor?	Si. Sin embargo en el proyecto del cual fui cliente no fue el mayor causante del cambio en el alcance.	No. Si existió *overhead* fue en horas de desarrollo.	No	No tengo seguridad. Puede haber pasado, pero la influencia fue menor.
¿Podría estimar cuánto disminuyó el alcance del proyecto siendo 100 % el alcance que se pensaba obtener?	15. No se debe sólo al Nivel de Calidad.	0	0	5
¿El sitio web le fue útil?	Si	Si	Si	Si
¿Necesitó ayuda del Responsable de Calidad?	En ocasiones	Si	No	Si

Tabla 4.3: Las Preguntas y Respuestas de los Clientes - 1

Pregunta	Cliente Uno	Cliente Dos	Cliente Tres	Cliente Cuatro
¿Es necesario que esta persona exista?	Si	Si. Al menos se necesita contar con una entrevista en el transcurso del proyecto.	Si	El sitio es suficiente para realizar las actividades. Si queremos ser muy estrictos en la aplicación de la propuesta es necesaria la participación del experto.
¿Está conforme con la calidad del producto obtenido?	Si	Mucho	Si	Si

Tabla 4.4: Las Preguntas y Respuestas de los Clientes - 2

Cliente Uno	Creo que, comparados con los proyectos de años anteriores, en estos proyectos es clara la mayor rigurosidad en el desarrollo y en el conocimiento de las características de los productos. Además, es independiente de las funcionalidades del producto y la calidad del proyecto en sí mismo. En la historia han habido proyectos excelentes, con un resultado mucho mejor, pero el software entregado siempre era algo misterioso. En estos casos, aun no siendo en algunos casos resultados excepcionales, la calidad del producto resulta más clara. Por mi parte creo que fue una muy buena experiencia, tanto por la experiencia como por los resultados.
Cliente Dos	Creo que del lado de los desarrolladores fue una tarea más que se debió sumar al esfuerzo total del proyecto pero que no fue tan pesada como todos suponían al principio. Además, ¡el resultado para todos los proyectos que siguieron esto fue muy exitoso! Para el lado del cliente significó poco esfuerzo extra.
Cliente Tres	El hecho de que los desarrolladores tengan que pensar en la calidad logra que desde el inicio del proyecto se tenga buena calidad. Además, las actividades del Framework dan un buen soporte, uno formal, a la hora de tener una idea de qué "tan bueno" es el producto obtenido.
Cliente Cuatro	Me pareció una experiencia excelente, pero como todo, exige un poco mas de esfuerzo de todas las partes y no siempre se dan las condiciones para realizarlo. Sería bueno que nos propusiéramos aplicarlo siempre.

Tabla 4.5: Los Comentarios de los Clientes

A partir de la encuesta se puede ver que los clientes:

- Piensan que la calidad de los productos es mejor al aplicar el Framework.
- Tienen un mayor conocimiento técnico del producto desarrollado.
- Destacan que los productos tienen menos defectos.
- Creen que el Framework puede servir para cualquier proyecto de grado de desarrollo de software dentro del CSI.
- Usan y les es útil el sitio web.
- En su mayoría necesitan ayuda del responsable de calidad.
- Están conformes con la calidad del prototipo entregado.

El alcance del prototipo sufre variaciones para algunos proyectos, sin embargo, esto sucede solo para 2 de los 11 proyectos estudiados. En uno de los proyectos se estima un 5 %. En el otro el alcance varía aproximadamente en un 15 %, pero el factor preponderante por el cual el alcance cambia no son las actividades de calidad. En este último caso no se tiene estimado cuánto de este 15 % corresponde a las actividades propuestas. Dados estos valores entendemos que el cambio en el alcance, desde el punto de vista de los clientes, es despreciable.

Por último, se observa que para los clientes debe existir el rol Responsable de Calidad. Sin embargo, dos de los cuatro clientes entienden que la participación del mismo puede ser menor a la que tuvo en el transcurso de estos proyectos. Se vuelve sobre este punto en la sección 3 del capítulo 5.

3.2. Evaluación de los Desarrolladores

Para obtener más información y poder evaluar con más datos los resultados de aplicar el Framework se realiza una encuesta a los desarrolladores. La misma se presenta en el Cuadro 4.6

1.	¿Cree que las actividades planteadas ayudaron a obtener un producto de mejor calidad?
2.	¿Cree que las actividades planteadas aumentaron el tiempo total del desarrollo? En caso de ser así, ¿podría estimar cuánto insumió siendo 100 % el total del desarrollo?
3.	¿El sitio web le fue útil?
4.	¿Necesitó ayuda del Responsable de Calidad?
5.	¿Es necesario que esta persona exista o sólo con el sitio web (mejorado) sería suficiente?

Tabla 4.6: Encuesta Realizada a los Desarrolladores

Las encuestas a los desarrolladores fueron realizadas de forma individual y hay dos maneras que parecen razonables para su procesamiento. Una es considerar cada encuesta por separado y que cada una tenga el mismo peso. La otra es ponderar por proyecto asumiendo que estos tienen características que los diferencian entre sí. En los cuadros 4.7, 4.8 y 4.9 se presentan las respuestas de los desarrolladores. Se muestran los números de proyecto únicamente con el propósito de conocer qué desarrolladores trabajaron en un mismo proyecto.

Pregunta	**Des. 1**	**Des. 2**	**Des. 3**	**Des. 4**	**Des. 5**	**Des. 6**	**Des. 7**	**Des. 8**
Proyecto	1	1	2	2	3	3	4	4
Nivel de Calidad	3	3	5	5	3	3	5	5
¿Cree que las actividades ayudaron a obtener un producto de mejor calidad?	Si	Si	Si	Si	NC	NC	Si	Si
¿Cree que las actividades planteadas aumentaron el tiempo total del desarrollo?	Si	Si	Si	Si	NC	NC	No	Si
¿En caso de ser así podría estimar cuánto insumió?	25	20	25	10	NC	NC	0	35
¿El sitio web le fue útil?	Si	Si	Si	Si	NC	NC	Si	Si
¿Necesitaron ayuda del Responsable de Calidad?	No	No	Si	Algo	NC	NC	Si	No
¿Es necesario que esta persona exista...?	Si	Si	Si	Si	NC	NC	Si	Si

Tabla 4.7: Preguntas y Respuestas a los Desarrolladores - 1

Pregunta	Des. 9	Des. 10	Des. 11	Des. 12	Des. 13	Des. 14	Des. 15	Des. 16
Proyecto	5	5	5	6	6	7	7	7
Nivel de Calidad	3	3	3	5	5	5	5	5
¿Cree que las actividades ayudaron a obtener un producto de mejor calidad?	Si	Si	Si	Si	Si	Si	Si	Si
¿Cree que las actividades planteadas aumentaron el tiempo total del desarrollo?	Si	Si	Si	Si	Si	Si	Si	Si
¿En caso de ser así podría estimar cuánto insumió?	20	30	15	35	10	10	5	10
¿El sitio web le fue útil?	Si	Si	Si	Si	Si	Si	Si	Si
¿Necesitaron ayuda del Responsable de Calidad?	No	Si	Algo	Si	Algo	Si	Si	Si
¿Es necesario que esta persona exista...?	Si	Si	Si	Si	Si	Si	Poco	Poco

Tabla 4.8: Preguntas y Respuestas a los Desarrolladores - 2

Pregunta	**Des. 17**	**Des. 18**	**Des. 19**	**Des. 20**	**Des. 21**	**Des. 22**	**Des. 23**	**Des. 24**	**Des. 25**	**Des. 26**
Proyecto	8	8	8	9	9	10	10	10	11	11
Nivel de Calidad	3	3	3	5	5	5	5	5	5	5
¿Cree que las actividades ayudaron a obtener un producto de mejor calidad?	Si	Si	Si	NC	Si	Si	Si	Si	Si	Si
¿Cree que las actividades planteadas aumentaron el tiempo total del desarrollo?	Si	Si	Si	NC	Si	Si	Si	Si	Si	Si
¿En caso de ser así podría estimar cuánto insumió?	30	29	20	NC	30	13	13	25	40	NC
¿El sitio web le fue útil?	Si	Si	Si	NC	Si	Si	Si	Si	Si	Si
¿Necesitaron ayuda del Responsable de Calidad?	Si	Si	Si	NC	Si	Si	Si	Si	Si	Si
¿Es necesario que esta persona exista. . . ?	Si	Si	Si	NC	Si	Si	Si	Si	Si	Si

Tabla 4.9: Preguntas y Respuestas a los Desarrolladores - 3

Los dos integrantes del proyecto número 3 y un integrante del proyecto número 9 no contestaron la encuesta por lo cual se considera un total de 23 desarrolladores para la evaluación.

A partir de los resultados obtenidos en las dos primeras preguntas se deduce que para los desarrolladores las actividades del Framework aumentan la calidad de los productos y también aumentan el tiempo total de desarrollo. Los resultados muestran que ambas consideraciones son independientes del Nivel en el cual se desarrolla el proyecto. El Cuadro 4.10 presenta el promedio general considerando que la respuesta *"Si"* da un puntaje 1 y la respuesta *"No"* da un puntaje 0. Se suman los puntajes de las respuestas y se divide entre 23, luego este número se multiplica por 100 para obtener el promedio.

Pregunta	**Promedio**
¿Cree que las actividades ayudaron a obtener un producto de mejor calidad?	100 %
¿Cree que las actividades planteadas aumentaron el tiempo total del desarrollo?	96,6 %

Tabla 4.10: Promedio de las Dos Primeras Preguntas a los Desarrolladores

El Cuadro 4.11 muestra el tiempo que insumieron las actividades propuestas respecto al tiempo total del desarrollo. Se divide el cálculo por Nivel de Calidad ya que entendemos que el tiempo que lleva realizar las actividades en Niveles más altos es mayor que en Niveles menores. También se realiza un corte por Nivel-Proyecto, entendiendo que los tipos de proyectos también provocan variaciones en el tiempo de realización de las actividades.

Forma de Cálculo	**Promedio**	**Mínimo**	**Máximo**
Nivel de Calidad 3 - Por Desarrollador	23,6 %	15 %	30 %
Nivel de Calidad 5 - Por Desarrollador	18,6 %	0 %	40 %
Nivel de Calidad 3 - Por Proyecto	23,5 %	21,7 %	26,3 %
Nivel de Calidad 5 - Por Proyecto	21,8 %	8,3 %	40 %

Tabla 4.11: Tiempo de las Actividades Propuestas Sobre Total de Desarrollo

Se hace difícil obtener conclusiones acerca del tiempo total que es usado en las actividades propuestas. Las respuestas fueron muy dispares, incluso las de integrantes de un mismo grupo de desarrollo. Se ve también que no existe mucha diferencia entre el tiempo usado en el Nivel 3 y en el Nivel 5, incluso los promedios muestran que en el Nivel 3 las actividades insumen más tiempo en el total del desarrollo. Se necesitan más estudios para razonar acerca de cuánto puede variar este valor en distintos proyectos.

Es importante notar que no se tiene en cuenta el tiempo (o costo) de la NO calidad; o lo que es lo mismo en este caso, el costo de no aplicar el Framework. Con esto queremos decir dos cosas: Primero, no tenemos cálculos sobre el re-trabajo que se evita por el hecho de realizar estas actividades. Segundo, al obtener un producto de mejor calidad, los proyectos de años posteriores que continúen dicho producto deben tener una mayor productividad que los proyectos que continúan productos que no fueron desarrollados usando

el Framework. Entonces, el *verdadero* tiempo que consumen las actividades de calidad es el consumido por la ejecución de dichas actividades menos el re-trabajo que se ahorra, menos el tiempo ahorrado al continuar el producto. Estamos convencidos de que este cálculo termina arrojando un número negativo, por lo que se ganaría en productividad.

El sitio web fue útil para el 100 % de los desarrolladores que respondieron a la encuesta. Esto indica que el armado del sitio fue correcto y que sirve para los objetivos planteados.

Para la pregunta *¿necesitaron ayuda del Responsable de Calidad?* se considera un puntaje de cero a las respuestas *"No"*, un puntaje de cero con cinco (0,5) a las respuestas *"Algo"* y un puntaje de uno a las respuestas *"Si"*. Se suman los puntajes y se dividen entre 23 obteniendo un puntaje de 0,76. A partir de este resultado se entiende que fue necesaria la ayuda del Responsable de Calidad. Separando por proyectos se ve que existe un proyecto para el cual la ayuda no fue necesaria, cinco proyectos en que sus desarrolladores sí pensaron que fue necesaria y en los otros cuatro proyectos sus integrantes tienen opiniones encontradas. Estos datos indican que la ayuda del Responsable de Calidad fue necesaria para casi todos los proyectos.

La última pregunta indaga sobre la necesidad de la existencia del rol de Responsable de Calidad. En la misma 21 desarrolladores contestaron *"Si"* y 2 contestaron *"Poco"*. Queda claro que para estos desarrolladores debe existir el rol. Más adelante se vuelve sobre este punto.

Los desarrolladores también realizaron comentarios en la encuesta, estos se muestran en los cuadros 4.12 y 4.13.

Desarrollador 1	El Responsable de Calidad es necesario porque existen dudas que pueden requerir la consulta a esta persona.
Desarrollador 2	El rol de Responsable de Calidad debe existir porque debe haber una persona a quien dirigir las dudas que se presenten
Desarrollador 3	Las actividades que más ayudaron a mejorar la calidad fueron las de testing. Fueron importantes las métricas de diseño para captar aspectos importantes a la hora de diseñar.
Desarrollador 4	Las actividades de calidad brindaron un camino marcado que hizo que el tema de calidad no fuera improvisado
Desarrollador 5	No contesta
Desarrollador 6	No contesta
Desarrollador 7	El Responsable de Calidad debería tener un rol más cercano al grupo de desarrollo. Debería brindarse capacitación antes o durante el proyecto para poder aplicar correctamente las actividades de calidad.
Desarrollador 8	Las actividades de verificación y de testing fueron las que más tiempo llevaron. Fue difícil cumplir con las actividades de calidad debido a que no estamos empapados en estos temas.
Desarrollador 9	El sitio web puede quitarte algunas dudas, pero otras sólo se aclaran con una persona capacitada. Pienso que gracias a la exigencia de ciertos niveles de calidad se pueden lograr mejores productos que sin esta exigencia.
Desarrollador 10	Se requiere de una persona con experiencia en el área, capaz de asistir en la aplicación de las distintas actividades del nivel asignado. Esto se debe a que las tareas a aplicar y cómo llevarlas a cabo no sólo dependen del Nivel de Calidad sino que también dependen mucho de la naturaleza del proyecto.
Desarrollador 11	No sé si todos son conscientes de que estas actividades agregan esfuerzo al desarrollo, pero no en vano, porque ¡se obtiene un producto mejor!
Desarrollador 12	Las actividades que me parecen más interesantes son: test unitario usando herramientas que lo automaticen, tracking de errores y evolución de las componentes y por último el uso de herramientas para el diagnóstico del código.
Desarrollador 13	Sin Comentarios

Tabla 4.12: Los Comentarios de los Desarrolladores - 1

Desarrollador 14	Sería bueno tener *templates* para algunos de los documentos que se piden, creo que ayudarían bastante. Otra cosa podría ser plantear un proyecto de ejemplo y en base a este describir las actividades y dar ejemplos de los documentos.
Desarrollador 15	El sitio web puede aportar más en cuanto a herramientas a utilizar para algunas de las actividades.
Desarrollador 16	Más allá del tiempo que puedan llevar las tareas de calidad, considero que vale la pena tenerlas en cuenta, sin duda influyen en la calidad del proyecto.
Desarrollador 17	En nuestro caso el testing no sólo llevó a corregir errores del sistema sino también a mejorar los algoritmos heurísticos que utilizamos.
Desarrollador 18	Las actividades (y especialmente documentarlas) aumentaron el tiempo de desarrollo, por otro lado no medimos cuánto tiempo disminuyeron al permitir detectar y corregir errores de forma temprana.
Desarrollador 19	Sin comentarios
Desarrollador 20	No contesta
Desarrollador 21	El tener un control de calidad explícito contribuyó a mejorar la calidad del producto ya que planteó una exigencia en las actividades a realizar y a no dejarlas de lado. El tener los niveles de calidad, las actividades especificadas para cada nivel y ayuda en cómo realizar cada actividad facilita la tarea de aplicarlas y que se hagan realmente efectivas.
Desarrollador 22	Creo que en general la aplicación de esta metodología beneficio el producto final. Un lado negativo es que llevó mucho mas tiempo hacer la documentación sobre los casos de prueba que correr los mismos.
Desarrollador 23	Para facilitar el trabajo serían necesarias plantillas para cada documento a realizar.
Desarrollador 24	Sería bueno que en la página web hubiera ejemplos de todos los tipos de documentos que se deben realizar, para poder ver el formato y aprender más fácilmente a realizar estos. También explicaciones de qué herramientas en particular ayudan a realizar cada cosa.
Desarrollador 25	Me parece que cualquier actividad de testing puede ayudar en la obtención de calidad. En particular las de nivel 5, que requieren sobre todo cubrimiento a nivel de componentes, me parecen importantes.
Desarrollador 26	Para aprovechar al máximo las actividades de calidad, en el sentido de seguir aprendiendo técnicas para lograr una buena calidad en el software, es necesario que el alcance del proyecto así lo permita.

Tabla 4.13: Los Comentarios de los Desarrolladores - 2

Los comentarios de los desarrolladores que nosotros entendemos más interesantes para este trabajo se pueden sintetizar en lo que sigue. En general los desarrolladores entienden que:

- Debe existir el Responsable de Calidad ya que es necesario que esté para aclarar dudas del Framework y ayudar en la forma de llevar a cabo las actividades para algunos proyectos particulares.
- El testing fue lo que más ayudó para conseguir productos de calidad.
- Las actividades marcaron un camino y esto hizo que no se improvisaran los temas de calidad.
- Se necesita capacitación antes o durante el proyecto para aplicar de forma correcta las actividades de calidad.
- Fue difícil cumplir con las actividades de calidad por no "estar empapados" en estos temas.
- Sería bueno contar con *templates* para los documentos que son salidas de las actividades.
- Contar con un proyecto de ejemplo y mostrar cómo se aplica cada una de las actividades para ese caso.
- Para cada actividad, en caso de ser posible, sugerir herramientas que faciliten y mejoren la ejecución la misma.

3.3. Nuestra Visión

Como se vio, la evaluación realizada tanto por los desarrolladores como por los clientes es muy positiva. Los desarrolladores entienden que gracias al Framework se mejoró la calidad de los productos construidos. Reafirmando esta impresión, los clientes están plenamente satisfechos con la calidad de todos los productos desarrollados y creen que estos son de mejor calidad que los construidos en años anteriores.

Nosotros coincidimos totalmente con estas opiniones. Entendemos que el Framework produjo la mejora de la calidad durante el desarrollo y también creemos que estos productos son mejores que productos de años anteriores en varias propiedades de calidad del producto de software. En particular, estos productos se desarrollan de forma más controlada, por lo que tienen ***menos defectos*** y estos son ***menos graves***. También creemos que los productos desarrollados en el Nivel 5 son claramente más ***fáciles de modificar***, ***extender***, ***reutilizar*** e ***integrar*** que prototipos de años anteriores. Además, el control que se tuvo en el desarrollo también permitió realizar un producto de ***fácil instalación***. Estas son todas las propiedades de calidad deseadas por los integrantes del CSI. Al mejorar en todas estas propiedades de calidad se ha cumplido con gran parte del objetivo específico de esta tesis.

Nuestra propuesta, como ya se mostró anteriormente, ***no redujo la productividad*** de los grupos de desarrollo,[3] y ***el aumento en la carga de trabajo de los investigadores del CSI no es significativo***. Estas dos situaciones cumplen con otra parte del objetivo específico.

El Framework debía respetar todas las características particulares de los proyectos del CSI que fueron mencionadas en el capítulo 1. Por lo cual, de cierta manera, el Framework queda *atado* a las mismas. Por esto es importante realizar una discusión centrada en estas características.

Los prototipos son sistemas de información o similares

Entendemos que el Framework sirve para otros tipos de producto además de sistemas de información. Sin embargo, no sirve para cualquier producto. Por ejemplo, sistemas de tiempo real. En este tipo de sistemas el tiempo es lo más importante, por lo que se deben realizar pruebas de performance. Este tipo de pruebas no son consideradas de forma explícita en el Framework. Queda fuera del alcance de esta tesis un estudio exhaustivo de los tipos de producto que son soportados por el desarrollo basado en el Framework.

La duración de los proyectos es normalmente corta

El Framework fue pensado para proyectos de corta duración. En proyectos de larga duración se hace necesaria la gestión de distintos aspectos del software que no son considerados en nuestra propuesta. Si bien no tenemos datos exactos de cuánto puede durar un proyecto que usa el Framework recomendamos que este no sea usado para proyectos de más de 18 meses de duración.

Los grupos de desarrollo son de dos o tres personas

En grupos de gran cantidad de personas comienza a ser fundamental la gestión de los recursos y del proyecto. Como ya se mencionó, el Framework, de forma intencional, no contempla la gestión del proyecto. La propuesta funcionó de forma adecuada para grupos de desarrollo de dos y tres personas. Creemos que el Framework puede soportar grupos de desarrollo de hasta cinco personas.

Los clientes son una o dos personas

Ninguna de las actividades del Framework prevé resolución de conflictos de requerimientos. Entendemos que con más de dos clientes pueden surgir distintos problemas que requieren de la realización de actividades particulares que no son provistas por el Framework, por ejemplo, resolución de conflictos, manejo de múltiples vistas, recolección de requerimientos de múltiples fuentes, definición de distintos tipos de clientes y cómo contemplar a cada uno. El Framework se ha probado con éxito con uno y dos clientes y pensamos que podría no responder bien si aumenta esta cantidad.

[3]Esto se desprende de que el alcance logrado no varió respecto del que se pensaba obtener.

El cliente es un investigador del grupo CSI

En el estudio de campo todos los clientes son investigadores del grupo CSI. Creemos que el cliente puede ser cualquier persona siempre y cuando entienda y se ajuste a las características que deben tener los proyectos que usan el Framework para el desarrollo.

Los desarrolladores no son desarrolladores consolidados

En nuestro caso todos los desarrolladores son estudiantes de grado y ninguno es un desarrollador consolidado. Entendemos que el Framework respondería aún mejor con desarrolladores consolidados. Esto se debe a principalmente a dos factores: primero, un desarrollador consolidado tiene una mayor productividad, segundo, un desarrollador consolidado conoce mejor las formas de mejorar la calidad del producto que está desarrollando.[4]

El alcance de los proyectos está dado por el tiempo que duran los mismos

El Framework no se ve limitado por esta característica, por lo que el mismo puede ser usado en situaciones *"comunes"*.

El lenguaje de programación es Java

Muchas de las actividades propuestas están fuertemente orientadas a lenguajes orientados a objetos. Ninguna de las actividades considera de forma específica al lenguaje Java. Entonces, creemos que el Framework funciona correctamente para cualquier lenguaje orientado a objetos. Además, las actividades que están pensadas para lenguajes orientados a objetos son adaptables a lenguajes procedurales, por lo que, con cierto trabajo de adaptación el Framework puede ser usado también para estos lenguajes.

A partir de lo presentado en esta subsección se deduce que se cumple con el objetivo específico de esta tesis. Se construyó un Framework que logra la mejora de la calidad de los productos, respeta el contexto de trabajo del CSI, no reduce la productividad en el desarrollo y no aumenta significativamente la carga de trabajo de los investigadores.

A continuación se presentan de forma compacta los beneficios y el contexto en el cual se puede ejecutar el Framework.

[4]Se debe recordar que en la encuesta varios desarrolladores manifiestan no saber mucho acerca de los temas de calidad y mencionan la necesidad de tener algún tipo de capacitación para lograr aplicar de forma correcta las actividades de calidad.

El *Framework para la mejora de la calidad* en su Nivel de Calidad 3 ha demostrado que:

- Reduce la cantidad de defectos del producto de software.
- Reduce la severidad de los defectos que quedan en el producto de software.
- Permite conocer un gran porcentaje[5] de las fallas que provocan los defectos remanentes.
- Produce documentos asociados al producto de software que son de suma utilidad: documento de requerimientos, casos de prueba y diseño de la aplicación, entre otros.

Al ejecutar en el Nivel de Calidad 5 se ha podido comprobar que los productos de software obtenidos son:

- Modificables.
- Extensibles.
- Reusables.
- Integrables a otros productos de software.

El Framework funciona en el siguiente contexto:

- Proyectos de menos de 18 meses de duración.
- Grupos de desarrollo de hasta 5 personas.
- Máximo de dos clientes relacionados con el grupo de desarrollo.
- Lenguaje de desarrollo orientado a objetos. *Se pueden realizar ajustes de manera que el Framework sirva para lenguajes procedurales.*

[5]Lamentablemente no se puede conocer este porcentaje. Para conocerlo se necesita tener a los productos en producción durante algunos meses. Esto no se pudo realizar en el estudio de campo.

5 Conclusiones

Vimos que el desarrollo académico de software no ha tenido la atención que merece desde el punto de vista de la Ingeniería de Software. Debido a esto, creamos un Framework para mejorar la calidad de los productos desarrollados en un ámbito académico.

El principal aporte de este trabajo es tener especificado y probado con éxito dicho Framework. Este respeta las características únicas del desarrollo académico de software y, a la vez, consigue mejorar varias propiedades de calidad de los productos.

Contenido

1. Conclusiones

Vimos que el desarrollo académico de software no ha tenido la atención que merece desde el punto de vista de la Ingeniería de Software. Debido a esto, los prototipos carecen de muchas de las propiedades de calidad que normalmente tienen los productos de software del mercado.

Conociendo esta situación, estudiamos las propiedades de calidad que son necesarias, y que hasta el momento no se cumplían, en los productos de software que desarrolla el CSI. Habiendo entendido las necesidades del grupo y el contexto de desarrollo dentro del mismo, creamos un Framework para mejorar la calidad de los productos.

Dicho Framework cuenta con 19 actividades a ejecutar por desarrolladores y clientes para lograr la mejora de las propiedades de calidad que son necesarias en los productos del CSI. Para que el mismo sea sencillo, se seleccionan únicamente tres disciplinas para componerlo: Especificaciones, Verificación y Validación, y Métricas y Medidas. Todas las actividades propuestas corresponden a partes técnicas de estas disciplinas, dejando de lado aspectos de gestión.

Como aplicación de la propuesta, se puso en práctica el Framework en once proyectos durante 2004 y parte de 2005. Siete de estos proyectos trabajaron en el Nivel de Calidad 5 y los cuatro restantes trabajaron en el Nivel de Calidad 3.

Realizamos un análisis de los resultados de los proyectos en cada una de las actividades que componen el Framework, y vimos que las mismas se realizaron de forma correcta y contribuyeron a mejorar la calidad del producto final. En general, creemos que las actividades propuestas se adecuaron al contexto del desarrollo académico de software. Salvo algunas excepciones, estas fueron aplicadas por los distintos grupos sin tener mayores dificultades. Las mismas son lo suficientemente flexibles como para que los desarrolladores puedan usar distintos métodos al aplicarlas. A la vez, están lo suficientemente definidas como para lograr los objetivos de calidad que cada una pretende alcanzar. Estas dos cualidades de las actividades permitieron que los desarrolladores se sintieran cómodos al aplicarlas y que distintos proyectos pudieran alcanzar los mismos objetivos de calidad aplicando métodos distintos.

Encuestamos tanto a los desarrolladores como a los clientes de forma de conocer su opinión acerca del funcionamiento del Framework. Todos los desarrolladores y todos los clientes opinaron que el Framework logró la mejora de la calidad de los productos desarrollados. Además, los clientes afirmaron que están satisfechos con la calidad lograda en los productos, y ambos, clientes y desarrolladores, quedaron muy conformes con la experiencia realizada.

Nosotros coincidimos totalmente con estas opiniones. Entendemos que el Framework produjo la mejora de la calidad durante el desarrollo, y también creemos que estos productos son mejores que los de años anteriores en las propiedades de calidad de productos de software que eran necesarias para el grupo CSI.

En el Cuadro 5.1 se presentan de forma compacta los beneficios y el contexto en el

cual se puede ejecutar el Framework.

El *Framework para la mejora de la calidad* en su Nivel de Calidad 3 ha demostrado que:

- Reduce la cantidad de defectos del producto de software.
- Reduce la severidad de los defectos que quedan en el producto de software.
- Permite conocer un gran porcentaje[1] de las fallas que provocan los defectos remanentes.
- Produce documentos asociados al producto de software que son de suma utilidad: documento de requerimientos, casos de prueba y diseño de la aplicación, entre otros.

Al ejecutar en el Nivel de Calidad 5 se ha podido comprobar que los productos de software obtenidos son:

- Modificables.
- Extensibles.
- Reusables.
- Integrables a otros productos de software.

El Framework funciona en el siguiente contexto:

- Proyectos de menos de 18 meses de duración.
- Grupos de desarrollo de hasta 5 personas.
- Máximo de dos clientes relacionados con el grupo de desarrollo.
- Lenguaje de desarrollo orientado a objetos. *Se pueden realizar ajustes de manera que el Framework sirva para lenguajes procedurales.*

Tabla 5.1: Beneficios y Contexto de Uso del Framework

Debe quedar claro que el Framework no está pensado para el desarrollo de productos *finales* de mercado. Sin embargo, el mismo sirve para desarrollar prototipos **no** desechables, por lo que estos se pueden modificar hasta conseguir un producto final; sobre todo, si se desarrolla en el Nivel de Calidad 5 o 6.

Se desprende del Cuadro 5.1 que se ha cumplido con los objetivos específicos que nos hemos planteado para esta tesis. También entendemos que se ha cumplido con el objetivo general. Creemos que el Framework es aplicable en el contexto que se menciona

[1]Lamentablemente, no se puede conocer este porcentaje. Para conocerlo, se necesita tener a los productos en producción durante algunos meses. Esto no se pudo realizar en el estudio de campo.

en el Cuadro 5.1; este contexto es el normal dentro del desarrollo académico de software. Además, las propiedades de calidad que se logran mejorar usando el Framework coinciden con las que normalmente son dejadas de lado en este tipo de desarrollo. De todas maneras, parece razonable aplicar y observar el funcionamiento del Framework fuera del CSI y fuera de Proyectos de Grado de estudiantes de forma de confirmar que el Framework es adecuado para todo el desarrollo académico de software.

El éxito de la propuesta se debe principalmente a los siguientes factores:

- las actividades son relativamente sencillas de ejecutar.
- las actividades pertenecen a disciplinas técnicas y medulares dentro de la construcción de software.
- no se proponen actividades de gestión.
- no se requieren documentos *pesados* que entorpezcan la realización del proyecto.
- el proceso de desarrollo de software es elegido por el grupo de desarrollo.

2. Aportes y Limitaciones

Uno de los aportes de este trabajo es haber realizado un análisis de un problema poco analizado y para el cual no hay trabajos específicos: la mejora de la calidad en el desarrollo académico de software. También se ha estudiado el estado del arte en temas más generales y se han discutido distintas propuestas desde el punto de vista del desarrollo académico de software.

Para nosotros, el principal aporte de este trabajo es tener especificado y probado con éxito un Framework para el desarrollo académico de software. Este Framework respeta las características únicas de este tipo de desarrollo y, a la vez, consigue mejorar varias propiedades de calidad de los productos.

En la Ingeniería de Software este tipo de desarrollo no ha sido considerado como se merece. Nuestra propuesta es un primer paso para cubrir un espacio existente entre el desarrollo de software de la industria y el desarrollo académico de software.

Otro de los aportes de este trabajo es haber logrado controlar el desarrollo de software y conseguido satisfacer las necesidades de los clientes usando actividades puramente técnicas. Hoy en día, los procesos pesados, RUP y otros, y los modelos de mejora de procesos, CMMI y otros, están en boga. Esto ha provocado la creencia de que no se puede desarrollar software si no se genera gran cantidad de documentación, no se realiza gran cantidad de actividades de gestión y no se usan procesos de desarrollo engorrosos y detallistas. Sin embargo, nosotros creemos que esto depende, entre otras cosas, del contexto

de trabajo y del tipo de desarrollo.[2] Vimos que, para el desarrollo académico de software, las actividades técnicas tienen mayor importancia que las de gestión.

Las limitaciones de nuestro trabajo están dadas, por un lado, por el Framework y, por otro, por el estudio de campo. El Framework se limita a un cierto contexto y no abarca todos los tipos existentes de desarrollo académico. En este sentido, entendemos que las mayores limitaciones son: el tipo de producto que se puede desarrollar y el lenguaje de programación a usar para el desarrollo basado en el Framework. El otro tipo de limitación está dado por el estudio de campo; este fue restringido al desarrollo de software por parte de estudiantes de grado y dentro de un único grupo de investigación de una Facultad.

3. Trabajo a Futuro

Esta sección presenta el trabajo que se puede realizar a futuro. La forma de continuar queda bastante clara, y se dan varias opciones: realizar más estudios de campo, mejorar la descripción de las actividades y mejorar las actividades que componen el Framework, generalizarlo para otros lenguajes y estudiar la posibilidad de que sea usado en todos los desarrollos de software del Instituto de Computación.

Entendemos que es necesario realizar otro estudio de campo. Sería interesante que este estudio se llevara a cabo en otros grupos del Instituto de Computación e incluso en grupos de desarrollo académico de software que estén fuera de la Facultad de Ingeniería. Estos estudios podrían confirmar nuestra experiencia y nuestras conclusiones. Este nuevo estudio serviría para mejorar el Framework en su conjunto y en cada una de las actividades propuestas, refinando las mismas y ajustándolas, en caso de ser necesario, a estos nuevos contextos.

Parece interesante contar con *templates* para cada uno de los documentos que son salida de las actividades propuestas. Esto provocaría una estandarización de todos los documentos en distintos proyectos, lo cual es favorable para los clientes. Además, al contar con esta ayuda, los desarrolladores no tendrían que utilizar tiempo en elegir documentos y formato para los mismos. Por último, contar con estos *templates* brindaría una ayuda más para entender mejor cada actividad y lo que estas se proponen conseguir.

Varias, si no todas, las actividades pueden tener herramientas asociadas que ayuden a su correcta ejecución. Parece interesante tener un conjunto de herramientas que colaboren entre sí y que sirvan de soporte al Framework. Las herramientas que parecen más interesantes son las de automatización y gestión del testing, obtención de forma automática de valores de diseño para distintas medidas y seguimiento de incidentes. Se debe tener cuidado y considerar cuáles de estas herramientas son las más adecuadas, ya que no se debería interferir con la productividad del grupo de desarrollo.

[2] También creen esto muchos de los defensores de los procesos *livianos*.

Algunos desarrolladores no sabían cómo ejecutar de forma correcta algunas actividades. Este hecho provocó que el Responsable de Calidad tuviera que colaborar varias veces con los grupos de desarrollo. Entendemos que con una capacitación previa en las distintas actividades, y sobre todo en las de testing, se puede lograr que el Responsable de Calidad sólo esté en el momento de la capacitación, que los grupos sean más productivos y que actividades que no fueron llevadas a cabo por la mayoría de los grupos, por ejemplo, *Planificación del Testing*, sean realizadas de forma apropiada por todos los grupos.

Es necesaria la mejora del sitio Web que contiene la especificación del Framework. Debe ser mejorada tanto la presentación del sitio como el contenido. Este se puede mejorar fuertemente mostrando los distintos documentos generados en las actividades para un proyecto particular. Este ejemplo permitirá a los grupos aclarar dudas rápidamente y, sobre todo, estudiar las mejores formas de realizar cada actividad.

Un punto interesante de trabajo a futuro es el de intentar generalizar el Framework para distintos tipos de lenguajes de desarrollo. Una primera aproximación puede ser considerar lenguajes procedurales e investigar qué cambios hay que realizar. Luego, parece interesante considerar otros paradigmas de programación que son usados en el Instituto de Computación, tales como la programación lógica y la programación funcional.

Entendemos razonable eliminar algunos Niveles de Calidad, ya que no parecen tener ninguna aplicación. Se debería estudiar si es razonable tener únicamente los niveles 3, 5 y 6.

Se nos ocurre un planteo muy ambicioso, pero que puede llegar a tener un impacto muy positivo en los proyectos desarrollados dentro del InCo. Si se considera el InCo como una organización de desarrollo, nos podemos permitir lo siguiente:

- Tener un grupo similar al SEPG (Software Engineering Software Group) de CMM. Este grupo estaría encargado de mantener y mejorar el Framework, teniendo retroalimentación desde todos los proyectos de grado del InCo.
- Tener pequeños grupos de apoyo al desarrollo, tales como Software Quality Assurance Group, Software Configuration Management Group. Estos grupos serían generales a todos los desarrollos del InCo, por lo que serían transversales a los proyectos.
- Recabar información de los tipos de defectos que se introducen durante el desarrollo académico de software. Si bien en este caso los grupos de desarrollo cambian constantemente, puede ser razonable suponer que, como los desarrolladores tienen una formación similar[3], los tipos de defectos que se inyectan son similares. En caso de que esto aconteciera, parece relativamente sencillo obtener formas de evitar y remover los defectos.

Como se mencionó, este último punto es muy ambicioso pero, en caso de lograrlo, la mejora de los productos de software desarrollados en el InCo sería muy notoria.

[3]Estar cursando la carrera de Ingeniería en Computación de la Facultad de Ingeniería o ser Ingeniero recién recibido de la misma Facultad.

Comentarios de Integrantes del Grupo CSI

A

En este apéndice se presentan algunos de los comentarios recibidos por los integrantes del grupo CSI. La pregunta formulada fue ¿En qué le aporta al CSI tener productos de mejor calidad? A continuación se presentan algunos de los comentarios recibidos.

Comentario uno

A mi me parece que la forma mas fácil de llevar adelante distintos proyectos en forma eficaz y poder alcanzar objetivos a veces muy ambiciosos, es minimizar el esfuerzo en cosas que no son directamente lo que uno desea desarrollar o investigar.

La misma definición de grupo implica que uno no trabaja aislado sino que colabora y se ve beneficiado de aportes de otros integrantes. Esta colaboración puede ser en muchos niveles, pero claramente me parece cuanto mas se acerque a parecer que en vez de colaborar, uno interactúa y es parte de un esfuerzo coordinado, los resultados en muchos niveles van a ser mejores.

En definitiva, mejorar la calidad de los prototipos, analizando el problema desde el punto de vista que vos lo haces tiene que hacer al grupo no solo sea mas productivo en el sentido de calidad de los prototipos, sino que en el volumen de cosas que se puedan investigar / prototipar.

Comentario dos

Los diferentes trabajos que hace CSI tienen una fuerte componente experimental, en el sentido que se proponen cosas (metodologías, lenguajes, etc.) que es necesario validar si cumplen la función para la que fueron desarrollados. El prototipo es un medio para eso. Por otro lado, también se espera (creo que si bien no es demasiado explícito en algunos casos) que tenga un mínimo (en realidad máximo) impacto industrial... Para esto también los prototipos son importantes...

Sin embargo, la calidad de los prototipos actuales, tanto los realizados por los estudiantes como los realizados por nosotros, es objetivamente, baja. Esta afirmación se basa en los argumentos manejados en la charla del otro día... (no comunicación entre los diferentes prototipos, bugs, no es posible reutilizarlos...etc.) Tener los prototipos con mejor calidad implica realizar mejores validaciones y levantar la expectativa de la construcciń posterior (o no) de un producto.

Comentario tres

En que eso va a permitir mejor construir herramientas donde sus partes se fueron construyendo incrementalmente en distintos proyectos.

En que vamos a tener mayores garantías de que las propuestas de investigación que estemos probando en esos prototipos realmente "funcionan"

En que nos da mayores garantías acerca de lo que construyen los proyectos de grado.

Comentario cuatro

Pienso que aporta en:

- la prototipación en si de ideas
- la facilidad para la continuidad cuando el grupo de gente cambia
- abrir la posibilidad de presentar demos en secciones de este tipo que existen en algunas conferencias
- si se trata de un prototipo de "base", tener un prototipo que funcione correctamente permite ser usado en trabajos donde se pueden "aplicar"
- comunicación hacia las empresas de software (me parece que en general que hay un acercamiento si les mostrás algo funcionando; a partir de ahí creo que te pueden escuchar los "fundamentos" de lo que "vieron").

Bibliografía

[AA05] F. Alvarez and R. Amador. Federador Link-All. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[AEG+98] M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, and K. Weidenhaupt. Survey on the scenario use in twelve selected industrial projects. Technical report, AIB 98-07, RWTH Aachen, 1998.

[AG05] I. Abel and P. Giampedraglia. Análisis e implementación de un toolkit para testing de performance. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[AHH04] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In Springer Berlin / Heidelberg, editor, *Genetic and Evolutionary Computation – GECCO 2004*, volume 3103 of *Lectures Notes in Computer Science*, pages 1338–1349, 2004.

[Ant96] A. I. Antón. Goal-based requirements analysis. In *Proceedings of the Second International Conference on Requirements Engineering*, pages 136–144, 1996.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, November 2004.

[Bas97] V. R. Basili. Evolving and packaging reading technologies. In Elsevier Science Inc., editor, *J. System Software*, volume 38, pages 3–12, 1997.

[BBK+78] B. W. Bohem, J. R. Brown, H. Kaspar, M. Lipow, G. Muleod, and M. Merritt. Characteristics of software quality. North Holland, 1978.

[BBM95] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. Technical report, University of Maryland - Department of Computer Science, 1995.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BD02] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[BDL+06] B. Blanc, G. Durrieu, A. Lakehal, O. Laurent, B. Marre, I. Parissis, C. Seguin, and V. Wiels. *Automated functional test case generation from data flow specifications using structural coverage criteria.* 3ème Congrès Européen Embedded Real Time Software 2006, 2006.

[BFD04] C. Bunse, R. L. Feldmann, and J. Dorr. Agile methods in software engineering education. In *Proceedings of XP 2004: The Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering*, June 2004.

[BG05] J. Briosso and P. Gahn. Codificación de conocimientos médicos. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[BHR+92] R. Bentley, J. A. Hughes, D. Randall, T. Rodden, P. Sawyer, D. Shapiro, and I. Sommerville. Ethnographically-informed systems design for air traffic control. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 123–129. ACM Press, 1992.

[Bin00] R. V. Binder. *Testing Object-Oriented Systems. Models, Patterns and Tools.* Addison-Wesley, 2000.

[BK05] S. Benasús and A. Kind. Gestión de la producción intelectual de una universidad. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[BLF99] K. Breitman, J. C. S. P. Leite, and A. Finkelstein. The world's a stage: A survey on requirements engineering using a real-life case study. In *Journal of the Brazilian Computer Society*, volume 6, pages 13–37, 1999.

[Boe79] B. W. Boehm. Software engineering; r&d trends and defense needs. In MIT Press, editor, *Research Directions in Sofware Technology*, 1979.

[Boe81] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[BRH97] Steve Blythin, Mark Rouncefield, and John A. Hughes. Never mind the ethno' stuff, what does all this mean and what do we do now: ethnography in the commercial world. *interactions*, 4(3):38–47, 1997.

[Bro87] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–9, 1987.

[CCD+01] A. Cavarra, C. Crichton, J. Davies, A. Hartman, T. Jeron, and L. Mounier. *Using UML for Automatic Test Generation.* IBM - White Paper, 2001.

[CCD05] J. P. Campot, J. C. Campot, and G. Dodera. Distribución y deployment de aplicaciones en una plataforma basada en servicios. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[CES] CES. Centro de ensayos de software - www.ces.com.uy, última visita junio 2005.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. In *IEEE Transactions on Software Engineering*, volume 20, pages 476–493. IEEE Press, June 1994.

[CP05] Martin Cabrera and Sergio Perez. Estudio de la transformación de contenido educacional representado mediante LOM, hacia SCORM. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[CSM99] R. Conradi, B. Skatevik, and A. Marjara. An empirical study of inspection and testing data at ericsson. In *Proceedings of the 24th Annual Software Engineering Workshop*, 1999.

[DBT01] E. Díaz, R. Blanco, and J. Tuya. Automated software testing with metaheuristic techniques. In *ERCIM news*, number 58, 2001.

[DM05] N. Doroskevich and S. Moreira. Utilitarios para deployment de aplicaciones en una plataforma basada en servicios sobre j2ee. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[DRT01] A. Durán, A. Ruiz, and M. Toro. An automated approach for verification of software requirements. In *JIRA'2001 Proceeedings*, 2001.

[DT97] M. Dorfman and R. Thayer. *Software Requirements Engineering, 2nd Edition.* Wiley-IEEE Computer Society Pr, 1997.

[eAC94] F. Brito e Abreu and R. Carapuça. Object oriented software engineering: Measuring and controlling the development process, 1994.

[EWEBBF04] M. El-Wakil, A. El-Bastawisi, M. Boshra, and A. Fahmy. Object-oriented design quality models a survey and comparision. In *Second International Conference on Informatics and Systems*, March 2004.

[Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. In IBM, editor, *IBM Systems Journal*, volume 15, pages 258–287, 1976.

[FFGL00] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. Software requirements verification by natural language analysis: A cnr initiative for italian sme's. In *ERCIM News*, number 40, January 2000.

[FHK+97] P. Ferguson, W.S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya. Introducing the personal software process: Three industry case studies. In *IEEE Computer*, volume 30, pages 24–31, 1997.

[Fin93] A. Finkelstein. Report of the inquiry into the london ambulance service. electronic version prepared by prof. Finkelstein, avaiable at ftp://cs.ucl.ac.uk/acwf/info/lascase0.9.pdf with permission from the communications directorate, South West Thames Regional Health Authority, Original ISBN: 0 905133 70 6, 1993.

[Fre84] R. E. Freeman. *Strategic Management: A Stakeholder Approach*. Pitman Publishing, 1984.

[GGV04] V. Giaudrone, M. Guerra, and M. Vaccaro. Extracción e integración de información en una arquitectura de web warehousing. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2004.

[GL93] J.A. Goguen and C. Linde. Techniques for requirements elicitation. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 152–164, 1993.

[Gli00] M. Glinz. Improving the quality of requirements with scenarios. In *Proceedings of the Second World Congress for Software Quality (2WCSQ)*, pages 55–60, 2000.

[Gro95] The Standish Group. Software chaos, 1995.

[Gro99] The Standish Group. Chaos: A recipe for success, 1999.

[GRS05] L. Gonzalez, G. Roldos, and F. Serra. Mecanismo de ayuda contextual para sistemas de información federados. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[Heu01] J. Heumann. Generating test cases from use cases. Technical report, The Rational Edge, June 2001.

[HL01] H. F. Hofmann and F. Lehner. Requirements engineering as a success factor in software projects. *IEEE Software*, 18(4):58–66, 2001.

[Hof00] H. F. Hofmann. *Requirements Engineering: A Situated Discovery Process*. DUV Gabler Edition Wissenschaft, 2000.

[HPW98] P. Haumer, K. Pohl, and K. Weidenhaupt. Requirements elicitation and validation with real world scenes. In *IEEE Transactions on Software Engineering*, volume 24, Issue 12, pages 1036–1054, 1998.

[Hum95] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[Hum00] W. S. Humphrey. The personal software process (PSP). Technical Report CMU/SEI-2000-TR-022, Software Engineering Institute, Carnegie Mellon University, November 2000.

[IBM] IBM. Ibm rational unified process. http://www-306.ibm.com/software/awdtools/rup/.

[IEE90] IEEE. *Standard Glossary of Software Engineering Technology*. ANSI/IEEE Std. 610.12, 1990.

[IEE97] IEEE. *IEEE Standar for Software Reviews*. IEEE Press, 1997.

[IEE98] IEEE. *IEEE Guide to Sofware Requirements Specification*. IEEE Press, 1998.

[Ins96] European Software Institute. European user survey analysis. *Report USV-EUR 2.1, ESPITI Project*, 1996.

[ISOa] ISO. Iso 9001:2000. http://www.iso.org.

[ISOb] ISO. Iso 9126. http://www.iso.org.

[ISO95] ISO. *Gestión de la Calidad y Aseguramiento de Calidad. Vocabulario*. UNE-EN ISO 8402, 1995.

[JMS02] N. Juristo, A. Moreno, and A. Silva. Is the european industry moving toward solving requirements engineering problems? *IEEE Software*, 19(6):70–77, 2002.

[Jon91] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, Inc., New York, NY, USA, 1991.

[KMKT91] S. KUSUMOTO, K. MATSUMOTO, T. KIKUNO, and K. TORII. A new metric for cost effectiveness of software reviews. In *IEICE Transactions on Information and Systems*, volume E75-D, pages 674–680, 1991.

[LA] LINK-ALL. Local insertion network para américa latina - www.link-all.org, última visita febrero 2006.

[LF91] J. C. S. P. Leite and P. A. Freeman. Requirements validation through viewpoint resolution. In *IEEE Transactions on Software Engineering*, volume 17, pages 1253–1269. IEEE Press, December 1991.

[LNJ02] P. Laplante, C.Ñeill, and C. Jacobs. Software requirements practices: Some real data. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 121–128, 2002.

[lR05] K. Álvarez and I. Reolón. Utilitarios para federar servidores link-all basado en j2ee. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[Mar94] R. Martin. O.o. design quality metrics: An analysis of dependencies. Technical report, October 1994. las metricas que se presentan en mi tesis.

[MG05] N. Moha and Y. G. Guéhéneuc. On the automatic detection and correction of software architectural defects in object oriented designs. In Serge Demeyer, Kim Mens, Roel Wuyts, and Stéphane Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. Springer-Verlag, July 2005.

[MRW77] J. A. McCall, P. K. Richards, and G. F. Waters. Factors in software quality. volume 1, 2 y 3. Natl's Tech Information Services, 1977.

[Mye79] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE - Future of SE Track*, pages 35–46, 2000.

[NL03] C.Ñeill and P. Laplante. Requirements engineering: The state of the practice. In *Software, IEEE*, volume 20, pages 40–45, 2003.

[NSK00] U.Ñikula, J. Sajeniemi, and H. Kalvianen. A state-of-the-practice survey on requirements engineering in small- and medium-sized enterprises. Technical report, Telecom Business Research Ctr., Lappeenrata University of Technology, 2000.

[OP97] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification & Reliability*, 7(3):165–192, 1997.

[PCCW93] M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, February 1993.

[Per01] D.E. Perry. An empirical approach to design metrics and judgments. *New Vision for Software Design and Production Workshop*, Diciembre 2001.

[PPA05] A. Percy, S. Pari, and B. K. Aichernig. Automatic test case generation for ocl: a mutation approach. Technical Report UNU-IIST Report No. 321, International Institute for Software Technology, May 2005.

[PSS05] M. Panario, G. Signorele, and F. Sorbías. Nivel de calidad de la adaptación de cursos en ambientes de aprendizaje electrónico. In *Universidad de la República - Facultad de Ingeniería - Instituto de Computación*, Proyecto de Grado, 2005.

[PV03] S. Purao and V. Vaishnavi. Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221, 2003.

[PW97] A. Pouloudi and E. A. Whitley. Stakeholder identification in inter-organizational systems: Gaining insights for drug use management systems. In *European Journal of Information Systems*, volume 6, pages 1–14, 1997.

[PWG+93] M.C. Paulk, C.V. Weber, S. Garcia, M.B. Chrissis, and M. Bush. Key practices of the capability maturity model, version 1.1. Technical Report CMU/SEI-93-TR-25, Software Engineering Institute, Carnegie Mellon University, February 1993.

[RBE98] B. Regnell, P. Beremark, and O. Eklundh. A market-driven requirements engineering process: Results from an industrial process improvement programme, 1998.

[SBD98] U. Schroeder, M. Brunner, and M. Deneke. Constructionist learning in software engineering projects. In *P. Klint, J. Nawrocki: International Software Engineering Education Symposium*, 1998.

[SFG99] H. Sharp, A. Finkelstein, and G. Galal. Stakeholder identification in the requirements engineering process. In *Proceedings. Tenth International Workshop on Database and Expert Systems Applications*, pages 387–391, 1999.

[Som04] I. Sommerville. *Software Engineering (7th Edition)*. Addison Wesley, 2004.

[SS98] I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1998.

[SSV98] I. Sommerville, P. Sawyer, and S. Viller. Viewpoints for requirements elicitation: a practical approach. In *Proceedings. Third International Conference on Requirements Engineering*, pages 74–81, 1998.

[Tri04] J. Triñanes. Construcción de un banco de pruebas de modelos de proceso. In *4 Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento*, 2004.

[TSK+95] T. Tanaka, K. Sakamoto, S. Kusumoto, K. Matsumoto, and T. Kikuno. Improvement of software process by process description and benefit estimation. In *Proc. of the 17th International Conference on Software Engineering*, pages 123–132, 1995.

[Ude06] UdeS. Universidad de sydney. proyecto de desarrollo de software. http://www.ug.it.usyd.edu.au/∼soft3300/index.cgi?Home, 2006.

[Val05] D. Vallespir. Generación automática de casos de prueba. investigación en variables enteras. Technical Report 05-11, InCo PEDECIBA-Informática, 2005.

[VCBC04] J. Verner, K. Cox, S. Bleistein, and N. Cerpa. Requirements engineering and software project success: An industrial survey in australia and the u.s. In *Proceedings 9th Australian Workshop on Requirements Engineering (AWRE'04)*, pages 10.1–10.10, 2004.

[vL00] Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19, New York, NY, USA, 2000. ACM Press.

[VS99] S. Viller and I. Sommerville. Social analysis in the requirements engineering process: From ethnography to method. In *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 6–13, Washington, DC, USA, 1999. IEEE Computer Society.

[WPJH98] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. In *IEEE Software*, volume 15, Issue 2, pages 34–45, 1998.

[XSZC00] M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis. Object-oriented metrics - a survey, 2000.

[YC95] M. C. K. Yang and A. Chao. Reliability-estimation & stopping-rules for software testing, based on repeated appearances of bugs. In *IEEE Transactions on Reliability*, volume 44, pages 315–321. IEEE Press, June 1995.

[Zav97] P. Zave. Classification of research efforts in requirements engineering. *ACM Comput. Surv.*, 29(4):315–321, 1997.