



Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería

Escuela de Ingeniería Civil en Informática

**“EXTREME PROGRAMMING APLICADA AL
DESARROLLO DE UN PROTOTIPO DE SISTEMA
PARA LA GESTIÓN DE TRABAJOS DE
CONTRATISTAS”**

Proyecto para optar al título de

Ingeniero Civil en Informática

PATROCINANTE:

Renato Arturo Boegeholz Castillo

Ingeniero Civil en Informática

PROFESOR CO-PATROCINANTE

Jorge Mauricio Ruiz-Tagle Molina

Ingeniero Civil en Informática

PROFESOR INFORMANTE

Tania Denisse Letelier Santibáñez

Ingeniero Civil en Informática

Luis Felipe Álvarez Burgos

VALDIVIA – CHILE

2010

AGRADECIMIENTOS

A mi mamá y a mi papá, por la formación que me entregaron, por ser soporte cuando necesité ayuda y por ser incentivo y guía cuando necesite avanzar.

A Vanesa por su confianza, cariño y amor incondicionales.

A Loreto y Catalina por siempre subir el ánimo de toda la familia.

A Renato y Mauricio por guiar este Proyecto de Título.

A Perceptum por darme la oportunidad.

A todos mis amigos por su compañía y consejo.

ÍNDICE DE CONTENIDOS

ÍNDICE DE CONTENIDOS	i
ÍNDICE DE FIGURAS	iii
ÍNDICE DE TABLAS	v
RESUMEN.....	vi
1. INTRODUCCIÓN	1
1.1. Metodología	1
1.2. Aplicación	6
1.3. Objetivos	9
2. REVISION DE SISTEMAS SOFTWARE DE GESTIÓN DE PROYECTOS	10
2.1. Clasificación de los Sistemas Software de Gestión de Proyectos.....	10
2.2. Revisión de Sistemas de Código Cerrado.....	11
2.3. Descripción de Sistemas FLOSS	14
2.4. Conclusiones y Propuesta	17
3. EXTREME PROGRAMMING.....	19
3.1. Valores	19
3.2. Principios	22
3.3. Prácticas	25
4. DISEÑO EVOLUTIVO	30
4.1. Principios SOLID.....	31
4.2. Prácticas de Diseño Propuestas por <i>eXtreme Programming</i>	35
5. HISTORIAS DE USUARIO Y ARQUITECTURA PRELIMINARES.....	42
5.1. Historias de Usuario Preliminares	43
5.2. Arquitectura Preliminar	46
5.2.1. Selección de un Patrón de Arquitectura.....	46
5.2.2. Selección de un <i>Framework</i> de Desarrollo	49
6. ADAPTACIÓN DE LAS PRÁCTICAS DE <i>EXTREME PROGRAMMING</i>	53

6.1.	Prácticas	53
6.2.	Métricas.....	58
7.	PROTOTIPO.....	61
7.1.	Proceso de Desarrollo	61
7.2.	Resultado de las Métricas	70
8.	EVALUACIÓN.....	73
8.1.	Evaluación del Prototipo.....	73
8.2.	Evaluación de la Aplicación de la Metodología	74
9.	CONCLUSIONES Y TRABAJO A FUTURO.....	76
9.1.	Conclusiones	76
9.2.	Trabajo Futuro	77
10.	REFERENCIAS.....	78
10.1.	Libros y Publicaciones	78
10.2.	Material en Internet	80

ÍNDICE DE FIGURAS

Figura 1.1. Porcentaje de uso real de funcionalidades entregadas de un proyecto exitoso.....	3
Figura 1.2. Costo de cambio en un proyecto de software tradicional.	5
Figura 1.3. Curva de costo de cambio ágil	5
Figura 3.1. Valores, principios y prácticas de <i>eXtreme Programming</i>	19
Figura 4.1. Diseño acorde a SRP.....	31
Figura 4.2. Diseño acorde a OCP.....	32
Figura 4.3. Diagrama de clases que no conforman LSP.	33
Figura 4.4. Diseño acorde a ISP.....	34
Figura 4.5. Diseño acorde a DIP.	35
Figura 4.6 Ciclo del desarrollo guiado por pruebas	36
Figura 4.7. Prueba unitaria pasada.	37
Figura 4.8. Refactorización técnica Renombrar.....	38
Figura 4.9. Refactorización técnica PullUp.	39
Figura 4.10. Refactorización técnica PushDown.	39
Figura 4.11. Integración continua, configuración de red.	40
Figura 5.1. Esquema para materializar las historias de usuario.	42
Figura 5.2. Diagrama de dominio.....	44
Figura 5.3. <i>Wireframe</i> de la planificación de una orden de trabajo.	45
Figura 5.4. Flujo de información en MVC.....	47
Figura 5.5. Patrón de arquitectura propuesto por DDD.	48
Figura 6.1. Historias de usuario y tareas.	54
Figura 6.2. Panel informativo.....	55
Figura 6.3. <i>Fixtures</i> de la actividad y sus puntos de control.	57
Figura 6.4. Ejecución de pruebas unitarias.	57
Figura 6.5. Cobertura de código.....	59
Figura 6.6. Cobertura de código completa para el modelo Punto de Control.	60

Figura 7.1. Menú lateral	61
Figura 7.2. Dos órdenes de trabajo desplegadas.	62
Figura 7.3. Modelo de dominio en la primera iteración.....	62
Figura 7.4. Diagrama de modelos en la cuarta iteración.....	63
Figura 7.5. Componentes de una orden de trabajo en la tercera iteración.	64
Figura 7.6. Nuevas ventanas en la quinta iteración.....	66
Figura 7.7. Ventana de avisos.	66
Figura 7.8. Selección de una herramienta.	67
Figura 7.9. Modelo de dominio al finalizar la séptima iteración.	67
Figura 7.10. Calendario.....	68
Figura 7.11. Ingreso de usuarios.	68
Figura 7.12. Asignar un responsable a un punto de control.....	68
Figura 7.13. Planificación de una orden de trabajo en Excel.....	70
Figura 7.14. Velocidad.....	71
Figura 7.15. Precisión de la estimación.....	71

ÍNDICE DE TABLAS

Tabla 2.1. Lista de sistemas de gestión de proyecto de código cerrado.....	12
Tabla 2.2. Descripción detallada de las alternativas comerciales seleccionadas.	13
Tabla 2.3. Lista de sistemas de gestión de proyecto de código abierto.....	15
Tabla 2.4. Descripción detallada de las alternativas FLOSS seleccionadas.	16
Tabla 5.1. Estimación de las historias de usuario preliminares.	46
Tabla 5.2. Comparativa de patrones de arquitectura.	49
Tabla 5.3. Características y funcionalidades de CodeIgniter.	50
Tabla 5.4. Características y funcionalidades de CakePHP.....	51
Tabla 5.5. Sistema de puntaje para evaluar <i>frameworks</i>	51
Tabla 5.6. Comparación entre <i>frameworks</i> de desarrollo.....	52
Tabla 6.1. Velocidad y precisión de la estimación en la quinta iteración.	59
Tabla 7.1. Pruebas unitarias de la primera iteración.	62
Tabla 7.2. Pruebas unitarias nuevas en la cuarta iteración.....	64
Tabla 7.3. Pruebas unitarias nuevas en la quinta iteración.....	65
Tabla 7.4. Pruebas unitarias nuevas al finalizar la novena iteración.....	69
Tabla 7.5. Pruebas unitarias nuevas en la décima iteración.	69
Tabla 7.6. Horas trabajadas, velocidad y precisión por cada una de las iteraciones.....	70
Tabla 7.7. Cobertura de código del desarrollo.	72

RESUMEN

Los mercados actuales son altamente cambiantes y la capacidad de adaptación es una de las características más apreciadas por las organizaciones. Este planteamiento es compartido por las empresas de desarrollo de software, las que en respuesta proponen la aplicación de metodologías enfocadas en la adaptación y entrega constante de valor hacia los clientes; metodologías ágiles para el desarrollo de software.

El objetivo general del presente Proyecto de Título es aplicar una metodología ágil, más específicamente *eXtreme Programming*, al desarrollo de un sistema prototipo para la gestión de trabajos de contratistas.

Para definir algunas de las características con las que debería contar este prototipo, se revisaron y describieron diversos sistemas software existentes para la gestión de proyectos.

Se describieron las bases teóricas y técnicas que sustentan *eXtreme Programming*, las que finalmente se ven materializadas en prácticas, que fueron adaptadas y aplicadas por el equipo de desarrollo.

Finalmente, se evaluó la calidad del prototipo obtenido y el impacto que tuvo *eXtreme Programming* en el desarrollo y su resultado.

SUMMARY

In present days the markets are highly changing and adaptation is one of the most valued characteristic of an organization. This is shared by software development companies, who propose the use of methodologies focusing on adaptation and a steady flow of value towards the client; agile software development methodologies.

The main objective of this Project is to apply an agile methodology, more specifically eXtreme Programming, to the development of a prototype system for the management of contractor's works.

To define the features that the prototype system should count with, several existing project management systems were reviewed and described.

The eXtreme programming supporting theories and techniques are described; these theories and techniques are adapted and later applied by the development team.

Finally, the quality of the resulting prototype and the impact of eXtreme Programming in the development and its result were evaluated.

1. INTRODUCCIÓN

"Lo importante es no dejar de hacerse preguntas"

Albert Einstein 1879-1955.

Este Proyecto de Titulación está enfocado en la aplicación de la metodología *eXtreme Programming* al desarrollo de software en la empresa Perceptum¹, materializada en la producción de un prototipo de sistema de gestión de trabajos de contratistas.

La metodología seleccionada fue *eXtreme Programming*, por ser una metodología centrada en las personas, que proporciona un marco de prácticas para el desarrollo de software que permiten lograr aplicaciones de calidad, proporcionando valor a los clientes de manera temprana al enfocarse en aquellas funcionalidades de mayor importancia.

1.1. Metodología

Una metodología es un sistema de métodos, seguidos por una disciplina particular². Para el caso particular de la producción de software, recibe como denominación metodología de desarrollo de software.

Las organizaciones hacen grandes esfuerzos por ubicarse entre los primeros lugares en sus respectivos mercados y diferenciarse de su competencia. Lo que se intenta lograr es maximizar las utilidades, optimizando su producción y brindando un mejor servicio a sus usuarios.

Las empresas de desarrollo de software no son ajenas a este planteamiento; también buscan mejorar sus procesos para lograr un mayor grado de satisfacción en sus clientes.

Metodologías de desarrollo de software

Desde la década de 1970 existen modelos formales para explicar cómo debe ocurrir el proceso de software, como el conocido Modelo en Cascada. Los procesos que siguen el Modelo en Cascada están divididos en distintas etapas, las cuales no pueden estar traslapadas. Normalmente las etapas son [Roy70]:

- Requisitos de sistema.
- Requisitos de software.
- Análisis.

¹ <http://www.perceptum.cl>

² <http://wordnetweb.princeton.edu/perl/webwn?s=methodology>

- Diseño del programa.
- Codificación.
- Pruebas.
- Puesta en marcha.

Este enfoque tiene algunas desventajas [McC04]:

- Falta de flexibilidad.
- Es difícil predecir todos los problemas con anterioridad.
- Se pierde conocimiento importante entre etapas.
- Falta de cohesión en el equipo.
- Defectos que no son detectados hasta la etapa de pruebas, entre otras.

Por otro lado, el desarrollo iterativo e incremental ha estado presente por mucho tiempo en la aplicación de la ingeniería [Lar03]. Específicamente en la Ingeniería de Software, en 1976 Tom Gilbs en su libro *Software Metrics* en el cual se discute la práctica de la “gestión de proyectos evolutiva”, introduce por primera vez el término “evolución”:

“‘Evolución’ es una técnica para producir la apariencia de estabilidad. Un sistema complejo tendrá más éxito si es implementado en pequeños pasos, y si cada paso tiene una clara medida de logro, así como la posibilidad de "retiro" a un paso exitoso anterior, en caso de fallo. Además existe la oportunidad de recibir alguna retroalimentación del mundo real antes de entregar todos los recursos destinados a un sistema, y corregir posibles errores de diseño...” [Gil76].

Además, en 1976 Harlan D. Mills refuerza esta idea y establece que:

“El desarrollo de software se debe hacer gradualmente, en etapas con participación continua de los usuarios, re-planificación y con un diseño relativo a los costos de programación en cada etapa” [Mil76].

A medida que maduraba la Ingeniería de Software, se establecieron metodologías más formales de aplicación del diseño iterativo e incremental. Por ejemplo, en el artículo *The Spiral Model of Software Development and Enhancement* de 1988 por Barry Boehm [Boe88], se plantea que cada iteración representa un conjunto de actividades, las cuales no están definidas a priori sino que están dadas por el análisis de riesgo en la iteración anterior.

Durante la década de 1990 el Desarrollo Iterativo e Incremental o IID ganó amplia aceptación en la comunidad de software, a través de varias formas, por ejemplo, el prototipado rápido, Desarrollo Rápido de Aplicaciones o RAD [Mar91] y el Proceso Unificado de Rational o RUP.

La necesidad de un nuevo enfoque

Desde 1994 Standish Group³ edita cada dos años el informe *Chaos Report*, cuyo objetivo es medir la efectividad en los proyectos de software. Este estudio revisa miles de proyectos de software, abarcando muchos dominios y negocios. *Chaos Report* del año 2006 establece que un 15% de los proyectos resultan cancelados, un 51% de los proyectos resultan con problemas (entregados pero exceden en tiempo y/o presupuesto) y un 34% de proyectos que resultan exitosos, es decir, que logran las características con las que fueron concebidos inicialmente.

Además *Chaos Report* incluye la siguiente pregunta a aquellos proyectos exitosos: De las funcionalidades entregadas ¿Cuántas son realmente utilizadas? Frente a esta interrogante las respuestas a esta pregunta están resumidas en la Figura 1.1. donde podemos observar que un 45% de las funcionalidades no son utilizadas (nunca), y un 19% son utilizadas raramente [Sta06].

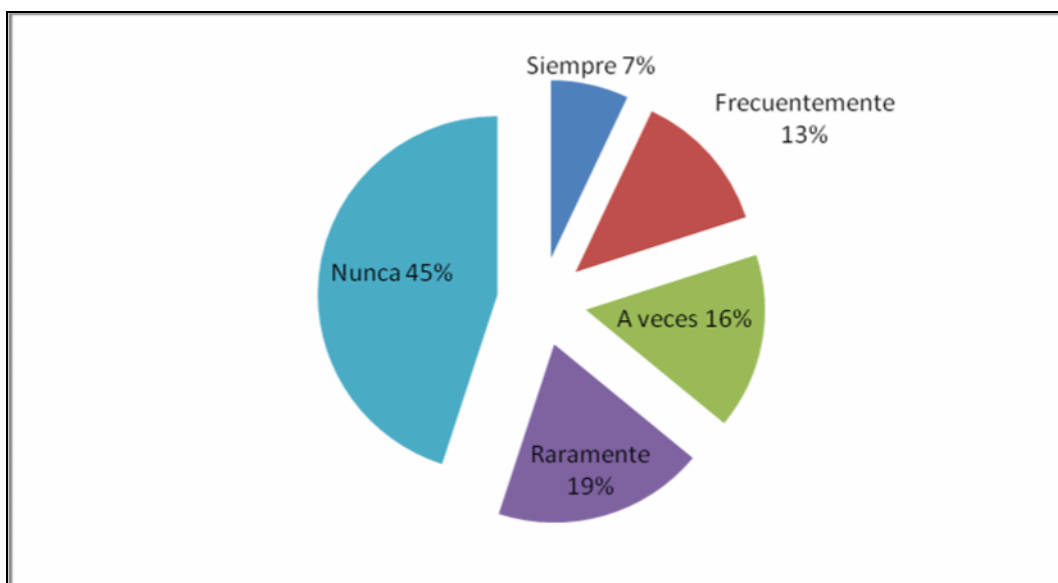


Figura 1.1. Porcentaje de uso real de funcionalidades entregadas de un proyecto exitoso (Fuente [Sta06]).

Estos casi dos tercios de funcionalidades representan -al no ser de directa utilidad y no generar valor para el cliente- una fuente de pérdida de velocidad y flexibilidad, dos características esenciales en el competitivo mercado de la generación de un nuevo producto. Este inconveniente puede deberse a muchas razones entre las que contamos:

³ <http://www.standishgroup.com/>

- Los requerimientos varían: En el período que ocurre entre la toma de requisitos y que el software es efectivamente puesto en funcionamiento, pueden ocurrir muchos cambios, entre ellos cambios en el negocio o cambios en la legislación.
- El entendimiento de los requerimientos varía: Muchas veces lo que es considerado como un cambio en los requerimientos, en realidad es que los clientes han mejorado su entendimiento de los mismos.
- Los clientes inventan requerimientos: Bajo el enfoque clásico, cambios en los requerimientos resultan muy caros de modificar en etapas tardías del desarrollo, esto obliga a enumerar al comienzo requerimientos que posiblemente pudiesen ser necesarios, pero que finalmente resultan en funcionalidades que no son utilizadas.

Dado estos costos extras, es necesario cambiar el enfoque donde el cliente percibe valor sólo al final del desarrollo, por uno donde el cliente reciba las funcionalidades de manera iterativa a medida que avanza el desarrollo del producto final.

Metodologías ágiles para el desarrollo de software

El enfoque “ágil” para el desarrollo de software propone una descripción inicial de los requerimientos de alto nivel; los detalles son desarrollados en un enfoque iterativo e incremental, comparable al método conocido como *Just In Time* o JIT⁴.

Para conseguir que este enfoque logre entregar valor de manera constante, el diseño debe mantenerse adaptable y los costos de cambio aceptables. Para ello las metodologías ágiles proponen la revisión de la curva de costo de cambio propuesta por Barry Boehm (ver Figura 1.2), de acuerdo a la cual un cambio en la etapa de análisis puede costar un dólar, mientras que este mismo cambio cuesta miles en producción [Boe81].

⁴ http://www.strategosinc.com/just-in-time_production.htm

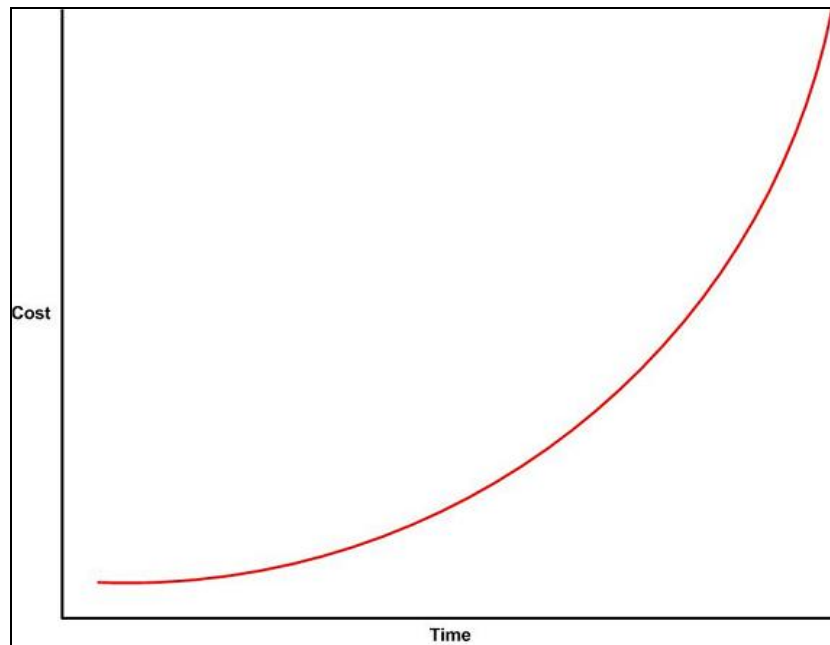


Figura 1.2. Costo de cambio en un proyecto de software tradicional. Fuente [Boe81].
 Esta revisión concluye que la curva de costo de cambio en un proyecto de software con una metodología ágil puede ser “aplanado” (ver Figura 1.3).

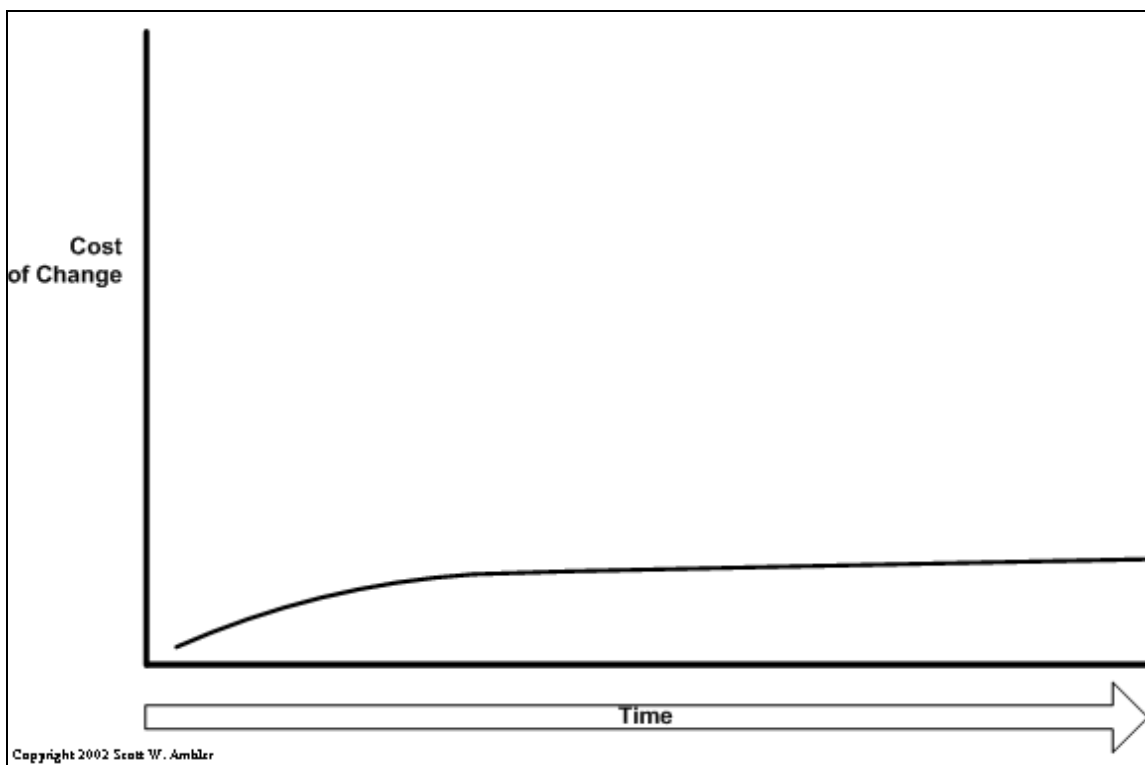


Figura 1.3. Curva de costo de cambio ágil. Fuente [Bec99].
 Este “aplanamiento” de la curva de costo habilita el diseño evolutivo, lo cual es a la vez permitido y explotado por las metodologías ágiles, en especial por *eXtreme Programming* o XP, como es expresado por Martin Fowler en el artículo *Is Design Dead?* (¿Ha muerto el diseño?) [Fow02]. Esto permite a los desarrolladores centrarse en aquellas funcionalidades que

aportan valor inmediato al software y no en aquellas que son percibidas como un aporte a futuro [Jef98].

Las primeras experiencias del uso de métodos ágiles como tales se encuentran en desarrollos de magnitudes más bien grandes. Por ejemplo, la metodología XP se inició en Chrysler Corporation en 1996 en un proyecto que utilizaba IID y en cuyo equipo se encontraban Ron Jeffries y Kent Beck [C3T98], mientras que *Feature Driven Design* comenzó a aplicarse en United Overseas Bank en Singapur, uno de los bancos más grandes de Asia [And04].

Al finalizar la década de 1990, la proliferación de las metodologías “livianas” (hoy conocidas como “ágiles”) que hacían uso de IID se hizo patente tomando formas como *Scrum*⁵ o *Crystal*⁶. Junto a su expansión, surgió la necesidad de coordinar los esfuerzos en torno a las metodologías.

El resultado más destacable de esta etapa se produce cuando algunos de los líderes responsables de la teoría y aplicación de estas metodologías se reunieron para escribir el *Manifiesto for Agile Software Development* (Manifiesto para el Desarrollo Ágil de Software), donde se establecen los principios del movimiento ágil⁷, que corresponden a:

- Personas y sus interacciones por sobre procesos y herramientas.
- Software funcional por sobre documentación exhaustiva.
- Colaboración del cliente por sobre la negociación con contrato.
- Responder al cambio por sobre seguir un plan.

El presente Proyecto de Título documenta la aplicación de este enfoque, más específicamente de *eXtreme Programming*, al desarrollo de un sistema prototipo cuyo objetivo es apoyar la gestión de trabajos a PYME contratistas chilenas.

1.2. Aplicación

Las empresas contratistas y la gestión de proyectos

Los contratistas son personas naturales o empresas que contando con la capacidad técnica y económica adecuada realizan un trabajo para una empresa principal o mandante. Este trabajo se encuentra regulado por un contrato entre el contratista y el mandante, y que especifica las

⁵<http://www.controlchaos.com/about/>

⁶ <http://alistair.cockburn.us/Crystal+Clear+distilled>

⁷ <http://agilemanifesto.org/>

condiciones en las cuales será realizado este trabajo. Un contratista provee de todos los materiales y herramientas para la ejecución de este trabajo⁸.

Actualmente, la mayoría de las empresas contratistas terminan sus trabajos en forma efectiva, pero -debido a desorden, falta o incumplimiento de documentación requerida por los mandantes- los trabajos terminados son facturados y cobrados mucho después de su finalización. Es por esto que, resulta fundamental para la oportuna facturación de los trabajos, la gestión de los mismos como proyectos.

De acuerdo al libro *A Guide to the Project Management Body of Knowledge* (Guía de los Fundamentos para la Dirección de Proyectos) el ciclo de vida de cualquier proyecto puede ser descrito a través de las siguientes etapas [Pro08].

- Definición de objetivos e inicio del proyecto.
- Organización y preparación del proyecto.
- Ejecución y control del proyecto.
- Finalización y cierre del proyecto.

La gestión de proyectos es la disciplina que aplica conocimiento, habilidad y técnica a la planificación, organización y gestión de recursos para lograr alcanzar o superar las expectativas del cliente [Pro08].

La gestión de proyectos eficiente requiere más que solamente una buena planificación, requiere que la información relevante sea obtenida, analizada y entendida oportunamente. Esto puede proveer de alertas tempranas sobre problemas pendientes y los impactos que estos podrían tener sobre las otras tareas, lo que podría generar modificaciones a la planificación o acciones de gestión [Ker09]. Es este el escenario donde el software para la gestión de proyectos se convierte en una herramienta importante para las organizaciones.

Sin embargo, en Chile, las PYME no han integrado TIC de manera masiva a sus actividades de negocio más específicas [Sub06], lo que es recogido en el informe *Business and Information Technologies (BIT) 2005* [PUC05], en el cual se expresa que la diferencia en el uso de TIC por parte de grandes empresas y PYME se ha estado acortando, no obstante, a medida que las tecnologías se vuelven más sofisticadas o modernas las diferencias se tornan más evidentes. Esto se ve reflejado en la existencia de software de gestión de proyectos muy evolucionado en el mercado, pero el costo, complejidad o nivel de especialización de estos

⁸ <http://es.wikipedia.org/wiki/Contratista>

sistemas los convierten en alcanzables sólo por las empresas mandantes, y no por las PYME que les entregan servicios.

Un sistema de gestión de proyectos para contratistas

Para definir las características y funcionalidades de un sistema de gestión de proyectos para contratistas, la empresa patrocinante se apoyó en un asesor con 15 años de experiencia en proyectos de infraestructura de empresas de telecomunicaciones.

El asesor, el encargado del proyecto por parte de la empresa patrocinante y el estudiante tesista constituyen el equipo de desarrollo.

En un primer análisis fueron distinguidas como fundamentales dentro del proceso de gestión de proyectos las etapas descritas a continuación:

- **Ingreso y valorización de trabajos:** El ingreso y valorización de un trabajo, es un proceso que es iniciado con la llegada de un requerimiento por parte de algún mandante, el cual es expresado como una **orden de trabajo**, la que contiene permisos, actividades y recursos requeridos. Esta orden de trabajo es “presupuestada” utilizando valores previamente acordados por contrato, una vez que este presupuesto es aceptado por el cliente, pasa a ser considerada “aceptada”. Los permisos, son pre-requisitos a la ejecución del trabajo, los cuales son gestionados por la empresa contratista (ejemplo permisos municipales). Las actividades son acciones que deben ser llevadas a cabo para la compleción del trabajo (ejemplo cablear un edificio). Los recursos son los materiales y herramientas requeridos por las actividades para su realización.
- **Planificación de trabajos:** La planificación de un trabajo es la etapa donde se calendarizan los permisos y actividades. Además por cada actividad se asignan recursos y se definen puntos de control. Una vez terminado este proceso, la orden de trabajo es considerada como “planificada”.
- **Seguimiento y control de trabajos:** El seguimiento y control es la etapa, en la que se encuentran aquellos trabajos que han sido planificados y que se encuentran en el estado “en ejecución”. Aquí es necesario completar los puntos de control, cada uno de los cuales deberá incluir documentación establecida en la etapa anterior. Cuando todas las actividades de una orden de trabajo han sido terminadas, esta es considerada “completada”.
- **Facturación y cierre de trabajos:** La facturación y cierre es la etapa donde se encuentran aquellos trabajos que se han completado. En esta fase, los documentos recopilados durante las etapas anteriores y el rebaje de materiales (informe de material

sobrante), son enviados al mandante. Este da su conformidad y a partir de este momento la orden de trabajo, pasa a la etapa de facturación. Una vez facturada la orden de trabajo es considerada “cerrada”.

Con estas definiciones se planteará una solución acorde a la realidad chilena; concretada en un prototipo de sistema de gestión de trabajos de PYME contratistas. Para el desarrollo del prototipo, se revisarán sistemas software existentes, de esta revisión se seleccionarán las características y funcionalidades más apropiadas para ser implementadas de manera iterativa e incremental, utilizando las prácticas propuestas por *eXtreme Programming*.

1.3. Objetivos

Objetivo General

Aplicar *eXtreme Programming* al desarrollo de un sistema prototipo para la gestión de trabajos de PYME contratistas.

Objetivos específicos

- Describir y explicar la problemática de la gestión de proyectos en PYME, en particular, la de trabajos de empresas contratistas.
- Identificar y comparar software existente relacionado con la problemática.
- Describir los valores, principios y prácticas de *eXtreme Programming*.
- Desarrollar un prototipo de gestor de trabajos para contratistas adaptando prácticas de *eXtreme Programming*.
- Evaluar la adopción de la metodología y su impacto en el producto obtenido.

2. REVISION DE SISTEMAS SOFTWARE DE GESTIÓN DE PROYECTOS

*"-Parece que tomamos el tren equivocado - le dije a mi hermana.
-Te te te- me contestó y se rió. Eso bueno tiene, que ni es miedosa ni acomplejada"*

Marcela Paz 1902-1985.

Papelucho perdido. 1960.

En el presente capítulo se clasifican, describen y comparan algunas alternativas existentes de sistemas software que ayudan a llevar a cabo proyectos a través de su ciclo de vida.

2.1. Clasificación de los Sistemas Software de Gestión de Proyectos

Se han seleccionado dos clasificaciones de sistemas software, según la cantidad de funcionalidades que poseen y según el acceso a la modificación de su código fuente.

Según la cantidad de funcionalidades:

- **Gama simple:** Este tipo de software son los más usados en el mercado, las herramientas más comunes en este grupo son hojas de cálculo, e-mail, entre otras. Entre las características comunes de los sistemas en este grupo están: manejan un solo proyecto, un solo usuario tiene acceso a un proyecto a la vez, lo que produce que la información esté dispersa a través de la organización.
- **Herramientas colaborativas:** Este tipo de software se ha desarrollado muy fuertemente en los últimos diez años, lo que distingue a estos sistemas de aquellos en la gama simple, principalmente es el hecho de que la información está centralizada y que son basados en Web. Esta clase de herramientas esta esencialmente orientada a la colaboración, por lo tanto carecen de algunas características específicas cuyo objetivo es planificar actividades, controlar costos o gestionar incidencias.
- **Gama media:** Esta categoría ha tenido un gran auge en los últimos diez años gracias a la masificación de Internet y de las aplicaciones Web. Entre sus principales características tenemos: son más sofisticados que los encontrados en las categorías anteriores al proveer una buena cantidad de funcionalidades, están diseñados para un moderado número de usuarios, son de fácil instalación y en algunos casos no se requiere ya que son provistos como un servicio o SaaS (del inglés *Software as a Service*).
- **Herramientas de alto nivel:** Estas herramientas existen desde hace mucho tiempo y son utilizadas por grandes organizaciones, por lo que manejan una gran cantidad de

usuarios y complejos requerimientos, tienen muchas funcionalidades, son altamente adaptables, normalmente su utilización tiene una mayor complejidad y requiere una alta inversión de tiempo y dinero para lograr su completo funcionamiento.

- Administración de portafolio de proyectos: La gestión de portafolio de proyectos, está centrada en determinar la mezcla y ordenamiento correcto de un conjunto de proyectos, para lograr los objetivos finales de una organización. Esta categoría es un poco más transversal y es posible encontrar herramientas de nivel medio o alto con estas características.

Según su acceso a la modificación del código fuente:

- De código cerrado: Es aquella clase de software cuyo código fuente no puede ser modificado.
- De código abierto: Es aquella clase de software cuyo código fuente puede ser modificado por el usuario final, estos son llamados generalmente FLOSS⁹.

De acuerdo a estas dos clasificaciones, se procede a analizar sistemas de gestión existentes.

2.2.Revisión de Sistemas de Código Cerrado

De aquellos sistemas cuyo código es cerrado, existen muchas alternativas. La Tabla 2.1 presenta las alternativas más importantes con una descripción básica:

⁹ Free/Libre Open Source Software

Tabla 2.1. Lista de sistemas de gestión de proyecto de código cerrado.

Sistema	Clasificación según funcionalidades	Plataforma	En español	Modelo de distribución	Costo	URL
Attask	Gama alta	Web	Sí	Web Instalado en cada organización	USD \$395 /usuario	http://www.attask.com/
Basecamp	Gama media	Web	No	SaaS	Desde USD \$24 mensuales	http://basecamphq.com/
Clarizen	Gama media	Web	Sí	SaaS	USD \$24.95 /usuario /mes	http://www.clarizen.com/
easyprojects.net	Gama alta	Web	No	SaaS e instalado en cada organización	Como SaaS USD\$24.99/usuario e instalado en la empresa USD\$190/usuario	http://easyprojects.net/
Enterplicity	Gama media	Web	No	SaaS	Desde USD\$24 a USD\$49 por usuario por mes	http://www.teaminteractions.com/
Microsoft Project	Gama básica	Microsoft Windows	Sí	Archivo instalable	Desde USD \$149.95	http://office.microsoft.com/es-es/project/
Primavera Project management	Administración de portafolios de proyectos	Windows, Linux y Web	Sí	Instalado en cada organización	USD \$2500/usuario	http://www.oracle.com/primavera
TeamworkPM	Gama alta	Web	Sí	SaaS	Desde USD \$12/mes hasta USD \$149/mes	http://www.teamworkpm.net/

Para un estudio más detallado se seleccionaron los sistemas **Basecamp**, **Clarizen** y **Enterplicity**, utilizando los siguientes criterios:

- Corresponden a la Gama Media: Las funcionalidades requeridas por las PYME en Chile están bien cubiertas por los sistemas de esta categoría y no conllevan una gran inversión en implantación.
- Están basados en Web, es decir los usuarios, pueden acceder desde cualquier parte utilizando un navegador Web.
- Son provistos como un servicio (SaaS), es decir, el servicio de mantención está incluido en la adquisición del software.

A continuación en la Tabla 2.2 se describen con detalle las alternativas seleccionadas.

Tabla 2.2. Descripción detallada de las alternativas comerciales seleccionadas.

Información Básica			
Nombre	Basecamp	Clarizen	Enterplicity
Desarrollador	37Signals ¹⁰	Clarizen Inc. ¹¹	Team Interactions, Inc. ¹²
Funcionalidades Básicas			
Carta Gantt	No	Sí	Sí
Sub - tareas	No	Sí	Sí
Planificación con dependencias	No	Sí	Sí
Manejo de Portafolios	Sí	Sí	Sí
Gestión de Recursos	Sí	Sí	Sí
Gestión Documental	Sí	Sí	Sí
Notificaciones de e-mail	Sí	Sí	Sí
Colaborativo	Sí	Sí	Sí
En español	No	Sí, pero los manuales están en inglés	No
Personalización	No(solo los colores)	Sí en los reportes	Modificación de los distintos módulos del Sistema.
Manejo de Activos	No	Manejo de gastos	No
Soporte			
Teléfono	No	No	No
e-mail	Sí	Sí	Sí
Presencial	No	No	No
Foros	Sí	Sí	Sí

¹⁰ <http://37signals.com/>

¹¹ <http://www.clarizen.com/AboutUs/CompanyOverview.aspx>

¹² <http://www.teaminteractions.com/aboutus.aspx>

2.3.Descripción de Sistemas FLOSS

Existen muchas alternativas cuyo código es abierto, en la Tabla 2.4 se presentan las más importantes, con una descripción básica:

Tabla 2.3. Lista de sistemas de gestión de proyecto de código abierto.

Sistema	ClaSificación según funcionalidades	Plataforma	En español	Lenguaje y requerimientos	URL
Collabtive	Herramienta colaborativa	Web	Sí	PHP5 y mysql 4 +	http://collabtive.o-dyn.de/
dotProject	Gama media-alta	Web	Sí	PHP5 y mysql 4 +	http://www.dotproject.net/
eGroupWare	Gama media-alta	Web	Sí	PHP5 y mysql 4 +	http://www.egroupware.org/
OpenGoo	Gama media	Web	Sí	PHP5 y mysql 4 +	http://www.opengoo.org
Project.net	Administración de portafolios de proyectos	Web - Escritorio (todas las plataformas)	Sí	Java Runtime Environment, Oracle	http://www.project.net/
project-Open	Gama alta - ERP	Web	Sí	OpenACS, AOLserver, Postgresql y alternativamente Oracle	http://www.project-open.com/
qdPM	Gama media	Web	No	PHP5 (symfony) y mysql 5+	http://qdpm.net/
Redmine	Gama media	Web	Sí	RubyOnRails, mysql 4+ o PostgreSql o Sqlite3	http://qdpm.net/
TaskJuggler	Gama básica	Escritorio (linux, mac)	No	C++, Qt	http://www.taskjuggler.org/

Para un estudio más detallado se seleccionaron tres sistemas utilizando los criterios explicados en el punto 2.2, a los que se añade:

- Se encuentran programados utilizando PHP y Mysql, esto debido a que son las tecnologías de mayor dominio en la empresa patrocinante.

Los sistemas seleccionados son **dotProject**, **eGroupWare** y **OpenGoo**, los que son descritos de manera detallada en la Tabla 2.4.

Tabla 2.4. Descripción detallada de las alternativas FLOSS seleccionadas.

Información Básica			
Nombre	dotProject	eGroupWare	OpenGoo
Desarrollador	Voluntarios	Comunidad eGroupWare	Feng Office Inc.
Funcionalidades Básicas			
Carta Gantt	Sí	Sí	No
Sub - tareas	No	No	No
Planificación con dependencias	Sí	No	No
Manejo de Portafolios	No	No	No
Gestión de Recursos	Recursos humanos	Sí	Sí
Gestión Documental	Sí	Sí	Sí
Notificaciones de e-mail	Sí	Sí	Sí
Colaborativo	Sí	Sí	Sí
En español	Sí	Sí	Sí
Manejo de Activos	No	Sí	Sí
Documentación del código fuente			
Documentación escrita	No	Sí	No
Pruebas unitarias	No	No	No
Soporte			
Foros	Sí	Sí	Sí
Otros		Listas de correos, IRC	
Coste			
Costo	No tiene costo asociado	No tiene costo asociado	No tiene costo asociado
Licencia	GPL	GPL	Affero GPL 3

2.4. Conclusiones y Propuesta

Sobre los sistemas de código cerrado

Los sistemas de código cerrado presentan algunas dificultades para su utilización, entre las cuales se pueden contar:

- **Modificación limitada:** Al ser de código cerrado las posibilidades de extensión son limitadas.
- **Sistemas genéricos:** Los sistemas seleccionados están diseñados para la gestión de proyectos en general no centrándose en los contratistas, lo que se traduce en funcionalidades faltantes o sobrantes.
- **Carecen de un sistema para la gestión de inventarios.**
- **Soporte limitado:** En su totalidad los proveedores de estos sistemas se encuentran en el extranjero.

Dado lo anterior, se concluye que no es recomendable el uso de software de código es cerrado al problema de la gestión de trabajos de PYME contratistas.

Sobre los sistemas FLOSS

Por otro lado los sistemas de código abierto, tienen restricciones similares a los de código cerrado. Sin embargo, dada la naturaleza modificable del código, muchos de estos problemas se convierten en solucionables, es decir, es posible:

- **Agregar o quitar funcionalidades y acomodar esta aplicación a las necesidades de las empresas contratistas.**
- **Integrar un sistema para la gestión de inventarios existente o construir uno.**
- **Proveer mantenimiento en Chile, a través de la empresa patrocinante.**

No obstante, la carencia de documentación del código en las alternativas analizadas, en especial de **pruebas unitarias**, las convierte en difíciles de mantener y extender.

De lo anterior, se concluye que no es recomendable la utilización de los sistemas FLOSS analizados como punto de partida para el desarrollo de un sistema para la gestión de trabajos de PYME contratistas.

Propuesta

Atendiendo a las necesidades de las PYME contratistas y en base a los sistemas estudiados, se propone la creación de un sistema para la gestión de proyectos centrado en la realidad de las PYME contratistas chilenas, que permita llevar a cabo un trabajo a través de las fases descritas anteriormente y que además contenga las siguientes características:

- **Gestión Documental:** La gestión documental, en este contexto se entiende como la administración de los documentos de cada trabajo y cada actividad de manera ordenada, los cuales deben poder ser descargados como “paquetes” de documentación.
- **Colaborativo:** Por colaborativo se entiende que múltiples usuarios puedan trabajar de manera concurrente en el mismo proyecto.
- **Personalización:** Por personalización se entenderá la capacidad de adaptar el sistema a la realidad de distintas empresas contratistas, incluyendo distintos rubros.
- **Manejo de inventarios:** El manejo de inventarios, corresponde a la gestión eficiente de materiales, los cuales podrían ser entregados por las empresas mandantes o adquiridos por la propia empresa contratista.
- **Notificaciones por e-mail.**
- **Tener una interfaz de usuario en español.**

3. EXTREME PROGRAMMING

"Los que se enamoran de la práctica sin la teoría son como los pilotos sin timón ni brújula, que nunca podrán saber a dónde van"

Leonardo da Vinci 1452-1519.

eXtreme Programming o XP, para muchos es una metodología de Ingeniería de Software, sin embargo, Jeffries, Anderson y Hendrickson al inicio de su libro *Extreme Programming Installed* la definen como:

“Una disciplina de desarrollo de software con los valores de la comunicación, la simplicidad, la retroalimentación, el coraje y el respeto”
[And01].

El presente capítulo está basado en la segunda edición del libro *Extreme Programming Explained* [Bec05], en el cual se indica que las prácticas de desarrollo están basada en cinco valores fundamentales.

Por otra parte, los valores se relacionan con las prácticas a través de los principios. Esto gráficamente se muestra en la Figura 3.1.



Figura 3.1. Valores, principios y prácticas de *eXtreme Programming*.

3.1. Valores

Los valores son aquellos aspectos del desarrollo que son considerados de especial relevancia para la obtención de software que permita cumplir con las expectativas de los clientes.

Los valores en XP se interpretan como criterios que juzgan lo que un equipo ve, piensa y hace. En XP estos valores son: comunicación, simplicidad, retroalimentación, coraje y respeto.

Comunicación

Lo más importante para un equipo de desarrollo de software, es asegurar las buenas relaciones entre los miembros del equipo.

Martin Fowler en su artículo *Who needs an architect?* (¿Quién necesita un arquitecto?), expresa que el software es una construcción social [Fow03a]. Por su parte, Alistair Cockburn en su publicación *Characterizing people as non-linear, first-order components in software development* (Caracterización de las personas como componentes no lineales de primer orden en el desarrollo de software), explica que las personas involucradas en el desarrollo de software influyen en el mismo, más que cualquier método o práctica. [Coc99]. Finalmente, Tom Demarco y Timothy Lister afirman que:

“...las interacciones humanas son complicadas, difíciles y poco claras en sus efectos, pero importan más que cualquier otro aspecto del trabajo” [Dem99].

La mayoría de las veces la aparición de problemas durante el desarrollo se debe a la existencia de conocimiento que no llega a las personas que son capaces de realizar los cambios [Bec05a].

Retroalimentación

Entre los dos problemas más comunes que tienen los proyectos fallidos de software se encuentran una incompleta definición de requisitos y los requerimientos cambiantes [Sta06].

La definición en detalle de los requisitos, funciona bien en aquellos ambientes donde los mismos no cambiarán y son estables en el tiempo. Sin embargo, en la mayoría de los proyectos, los requerimientos sí varían en el tiempo, y el cambio crea la necesidad de retroalimentación.

Los equipos que utilizan XP, buscan generar la mayor cantidad de retroalimentación, lo más rápido posible. Mientras más temprano se conozcan los cambios, más temprana resulta su adaptación.

Simplicidad

La simplicidad en XP corresponde al que probablemente sea el más complejo de los valores de XP. Esto es por la dificultad que representa crear un sistema lo suficientemente simple para solucionar sólo los problemas de hoy. En este sentido una de las frases más representativas de

este valor es “¿Qué es lo más simple que posiblemente pudiese funcionar?”¹³, este valor, tiene por objetivo la eliminación de la sobre-ingeniería, causante de funcionalidades no utilizadas.

Según Kent Beck [Bec99], para que un código sea simple debe cumplir con las cuatro siguientes propiedades:

1. Pasar todas las pruebas.
2. Expresar intencionalidad.
3. No contener partes duplicadas. Análogamente, maximiza la cohesión.
4. Utilizar la mínima cantidad de clases y métodos. Análogamente, minimiza el acoplamiento.

Coraje

El desarrollo de software es una actividad que involucra muchos riesgos. Las personas involucradas tienen muchos miedos respecto a lo que podría salir mal. Algunos de los miedos percibidos por los clientes son: pagar mucho y obtener muy poco o desconocer que está sucediendo con el desarrollo. Por otro lado, algunos de los miedos percibidos por los desarrolladores son: carecer de una clara definición de los requerimientos, sacrificar calidad para poder cumplir las metas, necesitar más tiempo para lograr terminar con éxito el proyecto, entre otros.

El coraje es la habilidad de sobreponerse a los miedos y perseverar en la realización de una acción¹⁴.

El coraje, sin el correcto contrapeso, puede resultar peligroso. Sin embargo, en la compañía de los otros valores puede ser muy poderoso. El coraje para decir la verdad, fomenta la comunicación y la confianza. El coraje para descartar una solución fallida y buscar una nueva, promueve la simplicidad. El coraje para buscar soluciones concretas y reales crea retroalimentación.

Respeto

El respeto, es un valor que está en el centro de XP. La adopción de los cuatro valores anteriores conduce a ganar el respeto de los demás en el equipo. Nadie debe sentirse despreciado o ignorado. Esto garantiza un alto nivel de motivación y fomenta la lealtad hacia el equipo, y el objetivo del proyecto. Este valor es muy dependiente de los demás, y está muy orientado hacia las personas del equipo. Referente a este valor, Kent Beck declara:

¹³ <http://c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork>

¹⁴ <http://es.wikipedia.org/wiki/Coraje>

“Si los miembros de un equipo no se preocupan unos de otros y lo que están haciendo, XP no funcionará. Si los miembros de un equipo no se preocupan de un proyecto, nada puede salvarlo.” [Bec05a].

3.2.Principios

Un principio puede ser entendido como una norma para ser aplicada en la acción¹⁵. Es comúnmente aceptado, que los principios son derivados de los valores, pero también es común que las personas tengan valores debidos a ciertos principios.

Dado que los valores tienden a ser demasiado abstractos y las prácticas son acciones concretas para desarrollar software; para disminuir esta distancia, los principios establecen técnicas intelectuales que actúan como un puente entre los valores y las prácticas. Los principios son usados para entender mejor las prácticas y para improvisar prácticas complementarias [Bec05b].

En XP, los principios son: humanidad, economía, beneficio mutuo, auto-similitud, mejoramiento, diversidad, reflexión, flujo, oportunidad, redundancia, fracaso, calidad, pasos pequeños, responsabilidad aceptada.

Humanidad

Son personas las que desarrollan y necesitan del software. Es por esto que el proceso de desarrollo de software, debe tomar en cuenta la humanidad de las personas que giran en torno a este, poniendo atención a sus necesidades, debilidades y fortalezas.

Entre las necesidades que deben ser llenadas para que las personas puedan producir buen software podemos encontrar: seguridad básica, sentido de logro, sentimiento de pertenencia, oportunidad de crecimiento, ser entendido y entender a otros.

Parte del desafío de un equipo, es balancear las necesidades del equipo, con las necesidades de los individuos.

Economía

Es necesario que el desarrollo, cumpla con los propósitos y objetivos del negocio al que se pretende ayudar agregando siempre el máximo valor. Los equipos XP buscan priorizar los aspectos más importantes del negocio y atenderlos rápidamente, dado que esto maximiza el valor del proyecto.

¹⁵ <http://dictionary.reference.com/browse/principle>

Por otro lado, dos aspectos importantes a considerar por el desarrollo de software son, el valor temporal del dinero y el valor de la opción del sistema y del equipo. El valor temporal del dinero dice que un peso hoy vale más que un peso mañana. El desarrollo del software es más valioso cuando gana dinero tempranamente y lo gasta de manera tardía. El diseño incremental explícitamente aplaza la inversión en diseño hasta el último momento responsable en un esfuerzo de gastar el dinero lo más tarde posible.

Otra fuente de valor económico de un desarrollo de software es su valor, como una opción para el futuro. Un sistema es mucho más valioso si tiene la opción de ser utilizado para más propósitos de los cuales fue inicialmente concebido.

Beneficio mutuo

El beneficio mutuo en XP es la búsqueda de prácticas que producen un beneficio ahora, en el futuro y también para el cliente. La comunicación con futuros desarrolladores se establece principalmente de la siguiente forma:

- Se escriben pruebas automáticas que permiten diseñar e implementar mejor hoy. Estas pruebas unitarias quedarán para beneficiar a un futuro desarrollador.
- La refactorización¹⁶ cuidadosa permite remover complejidad, esto ayuda a cometer menores defectos y hacer el código más fácil de entender a futuro.

Se toman nombres de una metáfora coherente y explícita, la cual mejora la velocidad de desarrollo y hace el código más claro para nuevos programadores.

Auto-similitud

Cuando se ha encontrado una estructura o método que funciona en un contexto, copiar esta solución en un nuevo contexto, incluso en escalas diferentes, puede resultar beneficioso. Por ejemplo: el ritmo básico para la generación de código es, escribir una prueba unitaria que falla, luego se hace una pequeña modificación en el código que permite pasar esta prueba. De la misma manera, al principio de una iteración se escriben historias de usuario, las que son traducidas en pruebas de aceptación y que serán aprobadas durante la iteración. Esto no quiere decir que se deban utilizar siempre los mismos patrones para cada problema. Algunas veces se deberá utilizar una solución única.

Mejoramiento

La perfección en el desarrollo de software no existe. No existen procesos perfectos, diseños perfectos, ni historias perfectas. Sin embargo, si se pueden perfeccionar. El objetivo de XP es

¹⁶ Modificación del código fuente sin cambiar su comportamiento. Ver <http://www.refactoring.com>

lograr excelencia a través de la mejora continua. La idea es hacer siempre lo mejor posible hoy, aprendiendo y entendiendo lo suficiente, para hacerlo mejor mañana. En este sentido, tener ciclos semanales, es una expresión de la posibilidad de mejorar los planes a largo plazo, a la luz de una mayor experiencia. El diseño evolutivo, hace uso de este principio al refinar de manera constante el diseño del sistema.

El principio del mejoramiento comienza, no esperando por la perfección. Para esto es necesario encontrar un punto de partida y mejorar desde allí.

Diversidad

Los equipos necesitan de una variedad de habilidades, actitudes y perspectivas que permitan visualizar los problemas, y detectar múltiples soluciones. En este sentido, dos ideas sobre un diseño presentan una oportunidad y este principio propone que todas las opiniones deben ser valoradas.

Reflexión

Los buenos equipos no solamente hacen su trabajo, piensan en cómo y por qué lo están haciendo. Analizan las causas del éxito o del fracaso. El objetivo es aprender de los errores cometidos. Estas reflexiones pueden tomar lugar al finalizar una iteración, durante la programación de a pares, o la integración continua.

Flujo

El flujo en el desarrollo de software se puede entender como la constante entrega de funcionalidades. Esto se logra mediante la realización; de manera simultánea; de todas las actividades del desarrollo (toma de requerimientos, análisis, diseño, codificación y pruebas).

El principio del flujo sugiere que para lograr un mejoramiento, se deben entregar pequeños incrementos de software de manera muy frecuente.

Oportunidad

Los problemas son oportunidades para el cambio. La actitud de sobrevivencia, lleva solamente a la solución del problema. Para alcanzar excelencia, los problemas deben ser convertidos en oportunidades de crecimiento personal, de mejorar las relaciones y de mejorar el software.

Redundancia

Los problemas críticos o muy difíciles, deben ser resueltos en más de una manera, así, si una solución falla, la otra prevendrá el desastre.

Si bien es cierto, las redundancias pueden ser fuente de residuo; remover aquellas que sirven a un propósito válido puede ser peligroso. Por ejemplo, tener una etapa de pruebas al final de

una iteración puede ser redundante. Sin embargo, puede ser eliminada sólo cuando se ha probado que no existen defectos en varios ciclos seguidos.

Fracaso

Ante el desconocimiento de la solución correcta a un problema, la manera más rápida de encontrar una solución es explorar, fallar y aprender de los errores. En XP el fracaso que imparte conocimiento, no representa un gasto.

Calidad

La calidad no debe ser una variable de control. Los proyectos no avanzan más rápido por aceptar una calidad más baja, ni más lento por exigir una mejor calidad. Generalmente, elevar los estándares de calidad produce entregas más rápidas, mientras que bajarlos significa incrementos tardíos y con menor predictibilidad. La calidad puede ser medida en muchos aspectos, entre ellos defectos, calidad del diseño, y en la experiencia del desarrollo. Cada mejora en la calidad produce mejoras en otros aspectos del proyecto, por ejemplo, productividad y efectividad.

Pasos pequeños

Cambios fundamentales realizados en un período corto de tiempo, pueden resultar peligroso. Una de las preguntas que un equipo debe hacerse es: “¿Cuál es el avance más pequeño y que esté visiblemente en la dirección correcta?”. El objetivo es realizar progresos pequeños, tan seguidos, que aparezcan un flujo. Este principio se ve reflejado en prácticas como escribir las pruebas primero y en la integración continua.

Responsabilidad aceptada

La responsabilidad no puede ser asignada, solamente puede ser aceptada. Este principio se ve reflejado en prácticas como: la persona que elige un trabajo también lo estima, la persona encargada de una historia es en última instancia la responsable de su diseño, implementación y pruebas.

3.3.Prácticas

Las prácticas son aquellas acciones concretas ejecutadas de manera diaria por los equipos XP. Estas son expresiones de los valores. Las prácticas de XP tienden a funcionar mejor en conjunto dado que entre ellas suelen amplificar su efecto.

Las prácticas son dependientes de las situaciones, si las condiciones cambian, las prácticas también deben cambiar para cumplir con los requisitos de esta nueva situación. Los valores no

se modifican de acuerdo a la situación, sin embargo, algunos principios podrían aparecer o desaparecer.

En XP estas prácticas son: sentarse juntos, todo el equipo, espacio informativo, trabajo energético, programación de pares, historias de usuario, ciclo semanal, ciclo trimestral, holgura, construcción rápida del sistema, integración continua, programación guiada por pruebas y diseño incremental.

Sentarse juntos

Todo el equipo trabaja en un espacio abierto, donde además cada integrante pueda tener un espacio para la privacidad. Esta práctica sostiene que el trabajo presencial, genera un proyecto más productivo y más humano.

Equipo completo

Reunir todas las habilidades y perspectivas necesarias para que un equipo pueda tener éxito. En este sentido las personas que interactúan con el sistema deben identificarse con el equipo y no con la función que cumplen.

Espacio informativo

El espacio compartido por el equipo desarrollador, está ambientado de manera que, cualquier persona interesada en el proyecto pueda hacerse una idea general del avance del proyecto en unos pocos momentos.

Trabajar con energía

Trabajar solamente la cantidad de horas en las que se es productivo. La base para esta práctica, es que el desarrollo de software, es una actividad que requiere creatividad y perspicacia, y estas características están más presentes en aquellas mentes que se encuentran relajadas y descansadas. El cansancio, no solo ralentiza el trabajo, sino que también, ayuda a disminuir la calidad de este.

Programación de a pares

La programación de a pares se refiere a que el código es producido por dos personas sentadas en un solo computador. La programación de pares es un diálogo entre dos personas que simultáneamente se encuentran programando, analizando, diseñando, probando y tratando de mejorar su trabajo.

La programación de pares permite a los desarrolladores:

- Mantenerse mutuamente concentrados en el trabajo.

- Mejorar el sistema.
- Aclarar ideas.
- Tomar la iniciativa cuando alguno de los desarrolladores se encuentra atrapado lo que disminuye la frustración.

Mantenerse en las prácticas y estándares del equipo.

Historias de usuario

Una historia de usuario es una forma de expresar los requerimientos a través de una descripción de funcionalidad, la que es escrita desde la perspectiva del cliente o usuario. Sin embargo, existe una diferencia clave entre las historias de usuario y los requerimientos, y es que las historias de usuario son muy tempranamente estimadas. Esto entrega a las personas relacionadas con el negocio, perspectivas sobre las funcionalidades que crean un mayor valor y que necesitan ser entregadas más tempranamente.

Las historias de usuario constan de un nombre, una descripción corta y una estimación.

Ciclo semanal

Planificar el trabajo de una semana cada vez, marcando el inicio de este ciclo con una reunión, durante esta sesión se realizan las siguientes actividades:

- Se analiza el progreso hasta la fecha. Incluyendo las estimaciones para la semana recién terminada con el avance real.
- El cliente elige las historias de usuario que serán realizadas durante la siguiente semana, basándose en sus estimados.
- Las historias son divididas en tareas. Los miembros del equipo las eligen y estiman.

Al iniciar la semana, se escriben pruebas automáticas, que pasarán cuando las historias estén completas. El resto de la semana se utilizará implementando las historias de usuario y logrando hacer pasar las pruebas automáticas.

Ciclo trimestral

Planificar el trabajo trimestralmente. Esta práctica busca alinear los equipos y sus proyectos con las metas de largo plazo.

Durante una planificación trimestral, se realizan las siguientes actividades:

- Identificar cuellos de botella.

- Iniciar reparaciones.
- Planificar los tópicos para el siguiente trimestre.

Elegir la forma en que serán abordados aquellos tópicos.

Holgura

En cualquier plan, se incluyen algunas tareas que puedan ser potencialmente dejadas de lado. Siempre es posible agregar una mayor cantidad de funcionalidades y entregar una mayor cantidad de historias que las prometidas.

Construcción rápida del sistema

Automáticamente construir todo el sistema y correr todas las pruebas en diez minutos. El objetivo de que este proceso no sobrepase los diez minutos es aumentar la frecuencia uso y así mejorar la retroalimentación.

Integración continua

Integrar y probar los cambios al sistema, con una periodicidad no mayor a dos horas. La programación en equipos es un problema donde se debe dividir, conquistar e integrar para vencer.

El proceso de integración puede ser impredecible y fácilmente tomar más que el tiempo estimado. Estas características se acrecientan en directa relación al tiempo que transcurre entre integraciones.

Desarrollo guiado por pruebas

Escribir una prueba automática que falla antes de cualquier cambio en el código. Esta práctica se ocupa de varios problemas a la vez:

- De alcance: al declarar explícitamente el objetivo del código se logra enfocar más definidamente el trabajo.
- De cohesión y acoplamiento: la dificultad al escribir las pruebas, es señal de un problema de diseño más que de las pruebas.
- De confianza: escribir código claro, que funciona y que expresa su intencionalidad a través de las pruebas automáticas, logra generar confianza entre los miembros del equipo.
- De ritmo: no es extraño perderse cuando se está codificando. Cuando se programa escribiendo una prueba primero, existen dos posibles caminos: escribir una nueva prueba o pasar alguna prueba que falla. Esto define el

ritmo natural de codificación; probar, codificar, refactorizar; nuevamente probar, codificar, refactorizar.

Diseño incremental

Invertir en el diseño del sistema todos los días. Hacer que el diseño se adapte a las necesidades de ese día.

Las pruebas automáticas, la refactorización, la integración continua y el explícito proceso social contribuyen a mantener los costos de cambio lo más bajo posible. Esto entrega a los equipos la confianza suficiente para adaptar el diseño a nuevos requerimientos.

El diseño incremental propone que el momento más efectivo para diseñar es a la luz de la experiencia. Esto significa que el diseño realizado lo más cercano a cuando es usado es más eficiente.

4. DISEÑO EVOLUTIVO

"De la independencia de los individuos, depende la grandeza de los pueblos."

José Martí 1853-1895.

¿Qué es el diseño?

En el contexto de las metodologías ágiles, la palabra diseño tiene dos acepciones:

Como actividad: El diseño es el proceso donde se introducen, eliminan y reestructuran elementos y relaciones, con el objetivo de mejorar el beneficio proporcionado por estas últimas [Bec09a]. Este proceso, es llevado a cabo de manera iterativa e incremental, por lo que recibe el nombre de diseño evolutivo [Fow02].

Como un producto: El diseño es un estado en el que se encuentran los elementos relacionados beneficiándose de sus interacciones. Lo cual se condice con la definición acuñada por Jack Reeves en su artículo *What is software design?* (¿Qué es el diseño de software?), donde se indica que el diseño de software es principalmente documentado a través del código fuente [Ree92]. Esta aseveración es confirmada por Robert Martin en *Agile Software Development, Principles, Patterns, and Practices* (Desarrollo Ágil de Software, Principios, Patrones y Prácticas) [Mar02a], donde se expresa que el diseño de un software puede ser representado de muchas formas pero la materialización de este es el código fuente.

¿Por qué diseñar?

Yourdon y Constantine, afirman que el objetivo del diseño de software es la minimización de los costos [Con79].

En el contexto de las metodologías ágiles, los costos en el desarrollo de software están dominados por los costos de mantención, los cuales a su vez están relacionados directamente a los costos de cambio. El diseño efectivo minimiza la propagación de los cambios [Bec09b], más específicamente, evita ciertas características como: rigidez¹⁷, fragilidad¹⁸ e inmovilidad¹⁹.

Una de las formas para lograr estos objetivos es separar un problema en distintas preocupaciones [Dij74]. Esta separación de preocupaciones está directamente relacionada con la maximización de la cohesión²⁰ y la minimización del acoplamiento²¹ [Con79].

¹⁷ Un código es rígido cuando es difícil de ser modificado, debido a que cualquier cambio repercute en muchos otros módulos del sistema

¹⁸ Un código es frágil cuando un cambio genera que el sistema deje de funcionar en partes no previstas

¹⁹ Un código es inmóvil cuando su reutilización requiere demasiado esfuerzo

²⁰ La cohesión es el índice de cuán estrechamente relacionadas se encuentran las distintas funcionalidades de un módulo

Durante el presente capítulo se abarcarán principios y prácticas que tienen por objetivo minimizar el costo que los cambios tienen en el desarrollo de software y de esta manera habilitan el diseño evolutivo.

4.1.Principios SOLID

En la presente sección se presentan los principios SOLID²² para el diseño orientado a objeto, estos principios son: de la responsabilidad única de las clases, abierto/cerrado, de la sustitución de Liskov, de la segregación de interfaces y de inversión de dependencia.

SRP – Principio de la Responsabilidad Única de las Clases

“Una clase debe tener una sola razón por la cual deba cambiar” [Mar02b].

En este contexto una responsabilidad se define como una razón para cambiar. Este principio se origina debido a que los cambios en los requerimientos se manifiestan a través de modificaciones en las responsabilidades. Si una clase tiene múltiples responsabilidades, estas se convierten en acopladas. Esto implica que cambios provocados por una de las responsabilidades podrían repercutir en otra, forzando más cambios.

A modo de ejemplo se presenta la Figura 4.1.

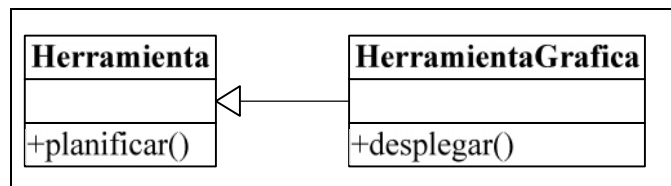


Figura 4.1. Diseño acorde a SRP.

En el ejemplo se pueden apreciar dos distintas responsabilidades separadas en clases distintas, la planificación de una herramienta y el despliegue de esta. Se puede ver como ambas son ejes de cambio; la planificación de una herramienta es utilizada por el módulo de planificación, mientras que el despliegue de una herramienta depende de la interfaz gráfica.

OCP – Principio Abierto/Cerrado

“Las entidades del software deben ser abiertas para ser extendidas, pero cerradas a las modificaciones” [Mey88,23].

²¹ El acoplamiento es el índice de la dependencia entre módulos

²² SOLID es un acrónimo para las primeras letras de cada principio en inglés SRP, OCP, LSP, ISP, DIP.

Este principio, establece que el código no debe cambiar con el tiempo; cuando se agreguen o modifiquen los requisitos, se debe agregar más código y no modificar el antiguo. El objetivo es evitar que las alteraciones al código, generen una “onda expansiva”, obligando a cambios en muchos otros módulos dependientes. Esto es logrado a través del uso de abstracciones, las cuales están fijas para ser modificadas, pero abiertas a ser extendidas.

Como un ejemplo se presenta en la Figura 4.2.

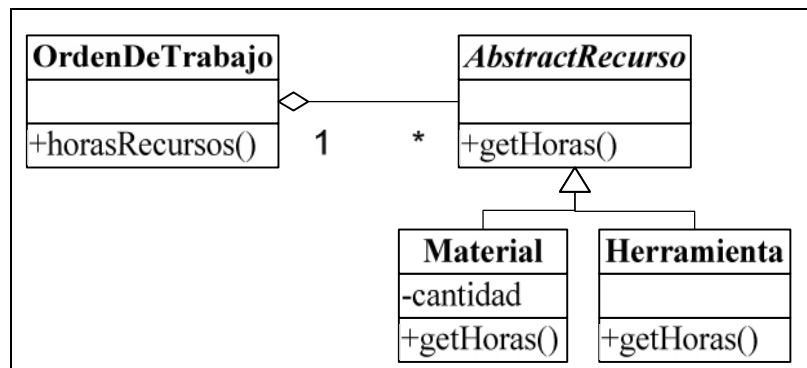


Figura 4.2. Diseño acorde a OCP.

A continuación se presenta la implementación de la clase **OrdenDeTrabajo**, esta clase se encuentra escrita en el lenguaje PHP²³.

```
<?php
class OrdenDeTrabajo {
    var $recursos = array();//arreglo de recursos
    function horasRecursos(){
        $horas = 0;
        foreach($this->recursos as $recurso){
            $horas+=$recurso->getHoras();
        }
        return $horas;
    }
}
?>
```

Este diseño es abierto para ser extendido con nuevos recursos, y esto no obliga a modificar el código antiguo.

No es posible cerrar un módulo completamente, siempre existirá algún tipo de cambio para el cual será necesario modificar el código existente. Dado esto, es preciso proveer una “cerradura” estratégica. Esto significa, que se deben elegir los cambios contra los cuales el diseño estará cerrado, esta “cerradura” está basada en lo experiencia de los diseñadores.

Este principio ha dado origen a algunas convenciones en el diseño orientado a objeto, como que todas las variables deben ser privadas y que no deben existir variables globales.

²³ <http://www.php.net>

LSP – Principio de la Sustitución de Liskov

“Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado” [Lis94].

Este principio, está focalizado en la forma en la cual se definen los subtipos. Específicamente, plantea que la herencia entre clases debe estar definida por el **comportamiento**. Es decir, la clase B es subclase de A siempre y cuando sea capaz de comportarse como A.

A modo de ejemplo podemos pensar en un cuadrado el cual, según su forma, es un tipo de rectángulo. Sin embargo, **un cuadrado no se comporta como un rectángulo**. Esto es evidente en el siguiente ejemplo:

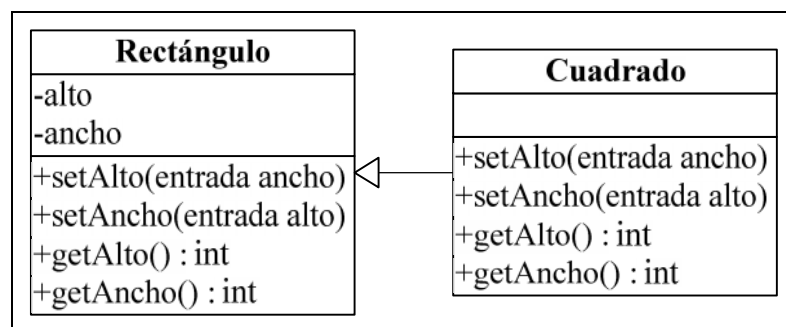


Figura 4.3. Diagrama de clases que no conforman LSP.

En la Figura 4.3 se muestra un diseño que no conforma LSP, debido a que un cuadrado no se comporta como un rectángulo. Esta circunstancia es presentada a través de la siguiente prueba unitaria.

```
<?php
function testCalculaAreaCorrectamente($r){
    $r->setAlto(4);
    $r->setAncho(5);
    $this->assertTrue(20,$r->getAlto()*$r->getAncho());
}
?>
```

Esta prueba unitaria sería pasada por un **Rectángulo**, no así por un **Cuadrado**. Por lo tanto, siguiendo el principio de la sustitución de Liskov, un cuadrado no es un tipo de rectángulo.

ISP – Principio de la Segregación de Interfaces

“Los clientes no deben ser forzados a depender de interfaces que no utilizan”
[Mar96a].

Este principio, define que las subclases no deben depender de interfaces que no son utilizadas.

Esto genera un acoplamiento entre las clases dependientes, debido a que cambios en una de ellas pueden forzar cambios en la interfaz y por lo tanto en el resto de las clases que dependen de la interfaz. A modo de ejemplo se presenta la Figura 4.4.

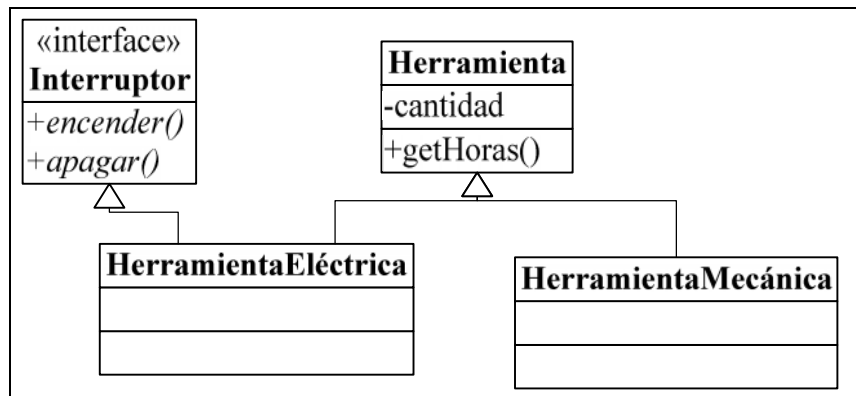


Figura 4.4. Diseño acorde a ISP.

En la Figura 4.4 se muestra la clase **HerramientaEléctrica**, la cual mediante herencia múltiple, evita que la clase **HerramientaMecánica** dependa de la interfaz **Encendible**, que no necesita.

DIP – Principio de Inversión de Dependencia

“Los módulos de alto nivel no deben depender de módulos de bajo nivel: ambos deben depender de abstracciones. Las abstracciones no deben depender de detalles: los detalles deben depender de las abstracciones”

[Mar96b]

Los módulos de alto nivel, son aquellos que contienen aquellas decisiones importantes y los modelos de una aplicación. Cuando estos módulos de alto nivel, dependen de aquellos de bajo nivel, cambios en estos últimos pueden tener directos efectos en los primeros y pueden forzarlos a cambiar. Además, esta dependencia hace muy difícil la reusabilidad de aquellos módulos de alto nivel.

El objetivo de este principio es desacoplar los componentes de alto nivel, de aquellos de bajo nivel, y así permitir la reusabilidad de los componentes de bajo nivel. Esto es logrado mediante la separación de los componentes de alto y bajo nivel en distintos paquetes, donde las interfaces que definen el comportamiento requerido por los componentes de alto nivel pertenecen y existen en su paquete.

A modo de ejemplo se presenta la Figura 4.5:

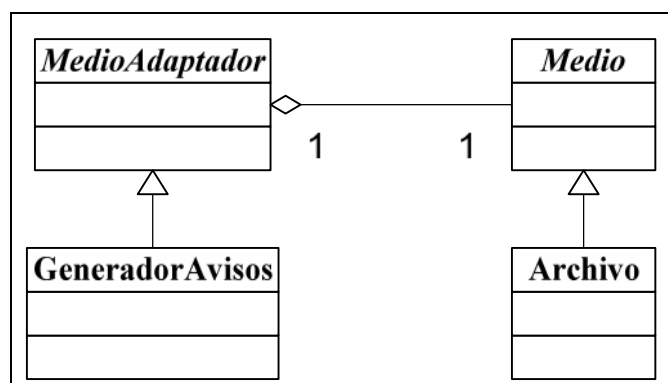


Figura 4.5. Diseño acorde a DIP.

Este diseño conforma el principio de la inversión de dependencia, dado que la clase **GeneradorAvisos**, no depende del medio que utilizará para entregar el aviso, permitiendo su reutilización, al mismo tiempo la clase **Archivo**, puede ser utilizada por otro módulo.

4.2. Prácticas de Diseño Propuestas por *eXtreme Programming*

En la siguiente sección se presentan algunas prácticas propuestas por XP, las cuales están centradas en habilitar el diseño evolutivo, estas son: el desarrollo guiado por pruebas, la refactorización y la integración continua.

Desarrollo Guiado por Pruebas

Es una práctica que propone escribir una prueba unitaria antes de modificar el código, lo que convierte a ésta, en una técnica de diseño, pruebas y documentación.

Las pruebas unitarias son trozos de código que sirven para probar clases, funciones y métodos. Una prueba unitaria contiene una afirmación (*assertion* en inglés), la cual indica si la prueba ha sido pasada. Las pruebas unitarias son escritas e implementadas por los mismos programadores. El ciclo de vida para el desarrollo guiado por pruebas es descrito en la Figura 4.6.

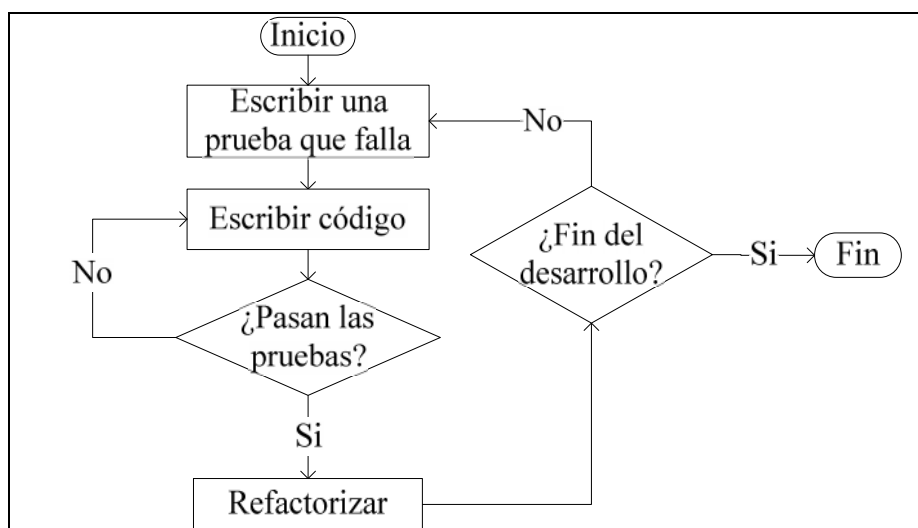


Figura 4.6 Ciclo del desarrollo guiado por pruebas. Fuente [Bec03].

Esta práctica tiene directa influencia en la menor cantidad de errores, en la mejora de la confianza en el software producido y en código más simple y fácil de modificar [Sin06].

Existen *frameworks* para la implementación y ejecución de las pruebas unitarias, como JUnit²⁴ y SimpleTest²⁵.

Para describir la implementación, a modo de ejemplo se utilizará una clase llamada **DateHelper**, la cual se encarga de dar formato a fechas.

Para diseñar la funcionalidad que permite dar un correcto formato a fechas, se escribirá una prueba unitaria utilizando Simpletest integrado con el *framework* CakePHP²⁶. Esta prueba en una primera instancia fallará.

```

<?php
App::import('Helper', 'Date');
class DateTest extends CakeTestCase {
    private $date=new DateHelper();
    function testFormateaCorrectamenteLasFechas(){
        $entrada = '2009-08-28 10:12:52';
        $esperado = '28/08/2009';
        $resultado = $this->date->format($entrada);
        $this->assertEqual($esperado,$resultado);
    }
}
?>
  
```

La razón por la cual esta prueba devuelve un error se debe a que no existe la clase **DateHelper**, por lo que se procede a su implementación.

²⁴ <http://www.junit.org>

²⁵ <http://www.simpletest.org>

²⁶ <http://www.cakephp.org>

```

<?php
class DateHelper extends AppHelper {
    var $formato = 'd/m/Y';
    public function format($date){
        return date($this->formato,strtotime($date));
    }
}
?>

```

Esta implementación es correcta y la prueba unitaria se muestra “pasada”, como se aprecia en la Figura 4.7.

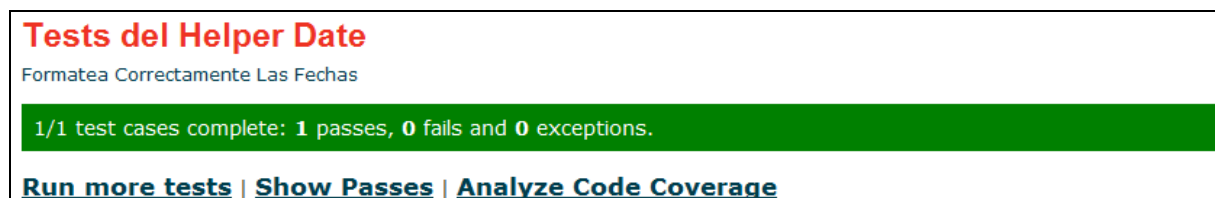


Figura 4.7. Prueba unitaria pasada.

Posteriormente, será posible extraer el formato de la fecha, de manera que se encuentre en un archivo de configuración. Si el procedimiento no ha sido correcto, esta prueba unitaria fallará.

Refactorización

La refactorización se puede entender como la reestructuración del código existente sin alterar su funcionamiento. Esto con el objetivo de mejorar algunos aspectos no funcionales del sistema [Fow99]. Entre los aspectos podemos encontrar:

- Mejorar la legibilidad del código y permitir que sea más fácil de entender.
- Mejorar la estructura del código y permitir que pueda ser extendido y modificado fácilmente.

La refactorización está directamente ligada al desarrollo guiado por pruebas, esta última disciplina permite disminuir el riesgo al obtener retroalimentación inmediata y conocer si se ha cambiado el comportamiento de la pieza modificada.

Entre las técnicas más comunes de refactorización podemos contar:

- Extraer: Esta técnica consiste en extraer parte de un método o función y colocarla en otra diferente, esto con el objetivo de hacerla más legible y utilizar este trozo de código en otras partes del sistema.

A continuación se presenta un ejemplo de esta técnica para lo cual se comienza con una función sin refactorizar, encargada de retornar la cantidad de horas planificadas que tiene un usuario:

```

<?php
class Usuario{
    var $actividades = array();
    public function horasPlanificadas(){
        $actividades = $this->actividades;
        $horas_planificadas = 0;
        foreach($actividades as $actividad){
            $horas_planificadas += $actividad->getHoras();
        }
        return $horas_planificadas;
    }
}
?>

```

Parte de la función `horasPlanificadas` es extraída y ubicada en una nueva función llamada `sumarHoras`. Este cambio es presentado a continuación y permite entender de mejor manera el código al mismo tiempo que permite que otros elementos utilicen la función `sumarHoras`.

```

<?php
class Usuario{
    var $actividades = array();
    public function horasPlanificadas(){
        $actividades = $this->actividades;
        return $this->sumarHoras($actividades);
    }
    private function sumarHoras($actividades){
        $horas = 0;
        foreach($actividades as $actividad){
            $horas += $actividad->getHoras();
        }
        return $horas;
    }
}
?>

```

- **Renombrar:** Este técnica consiste en modificar el nombre de una variable o función con el objetivo de revelar su propósito. Se muestra un ejemplo en la Figura 4.8.

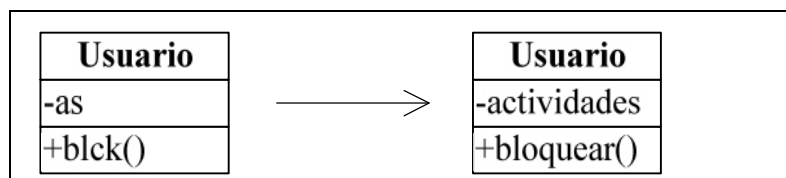


Figura 4.8. Refactorización técnica Renombrar.

- PullUp: Consiste en mover un miembro de una subclase a una superclase. Se muestra un ejemplo en la Figura 4.9.

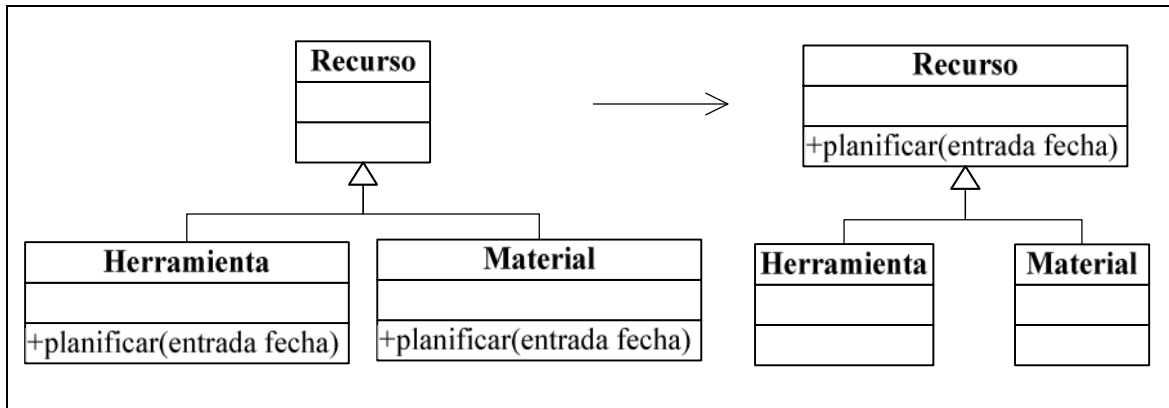


Figura 4.9. Refactorización técnica PullUp.

- PushDown: Consiste en mover un miembro de una superclase hacia una subclase. Se muestra un ejemplo en la Figura 4.10.

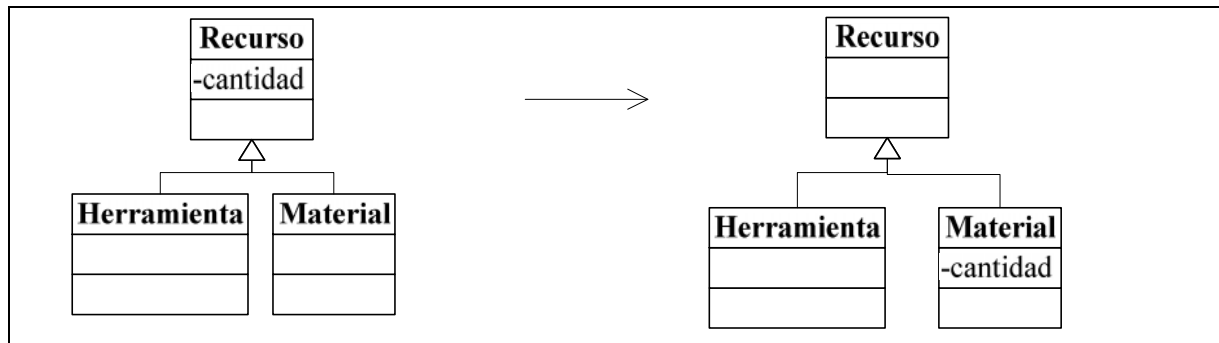


Figura 4.10. Refactorización técnica PushDown.

En la actualidad, existen herramientas que permiten la refactorización de manera automática o asistida. Estas herramientas se encuentran generalmente integradas en algunos entornos de desarrollo integrados o IDE (del inglés *Integrated Development Enviroment*).

Integración Continua

Esta práctica, propone que los cambios al sistema realizados por los distintos miembros de un equipo deben ser integrados con una frecuencia de al menos una vez al día. El objetivo de esta práctica es la disminución de riesgos. Al mantener baja la cantidad de cambios que son integrados -y al hacerlo de manera continua- el tiempo estimado en esta tarea se vuelve conocido [Fow06].

A modo de ejemplo, la integración continua puede ser llevada a cabo de la siguiente manera:

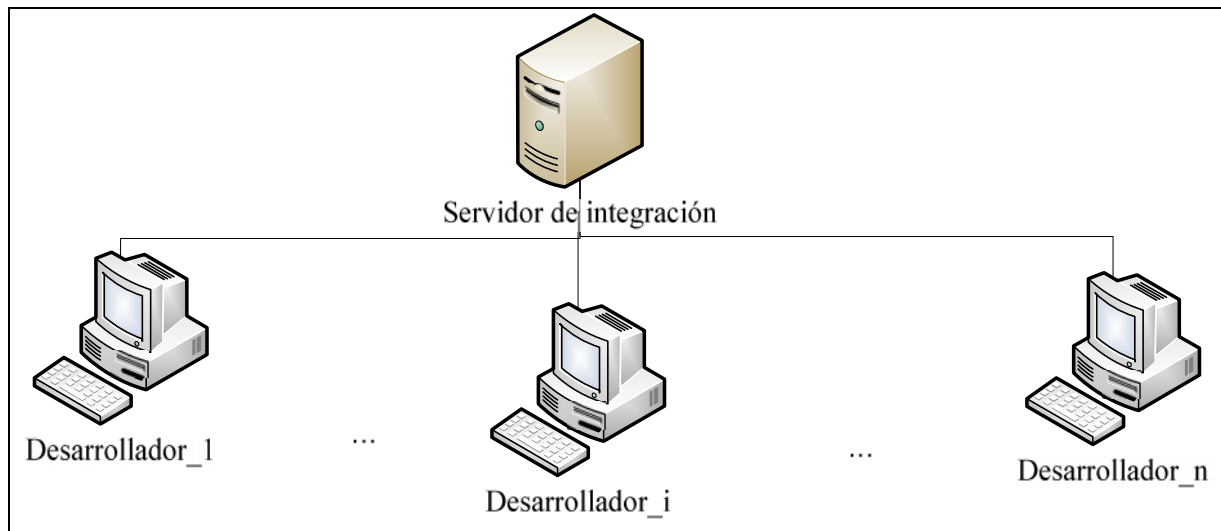


Figura 4.11. Integración continua, configuración de red.

La Figura 4.11 muestra la configuración de red con los desarrolladores teniendo acceso al servidor de integración.

1. Desarrollador_i obtiene el último código del sistema de control de versiones, que se encuentra en el servidor de integración.
2. Desarrollador_i realiza cambios, pasa las pruebas automáticas y construye el sistema.
3. Dado que los otros desarrolladores han realizado cambios y los han entregado al servidor de integración, Desarrollador_i debe obtener estas últimas modificaciones.
4. Desarrollador_i resuelve los posibles conflictos y ejecuta las pruebas automáticas. Si existen problemas estos deben ser corregidos.
5. Los cambios realizados por Desarrollador_i son subidos al servidor de integración. En el servidor de integración el sistema es construido, este puede ser un proceso manual o automático.

Entre las prácticas de integración continua más utilizadas podemos encontrar:

- Mantener un repositorio de código.
- Automatizar la construcción del sistema.
- La construcción del sistema ejecuta sus propias pruebas.
- Todos los desarrolladores entregan sus cambios al menos una vez al día.
- Cada entrega es construida, en la máquina de integración.
- Mantener rápida la construcción del sistema.
- Probar la construcción del sistema en una réplica del ambiente de producción.

- La última construcción es fácilmente obtenida por todas los clientes.
- Todos los miembros del equipo pueden ver el resultado de la última construcción.
- La implantación del sistema es automática.

5. HISTORIAS DE USUARIO Y ARQUITECTURA PRELIMINARES

"Lo último que uno sabe es por donde empezar."

Blaise Pascal 1623-1662.

En el presente capítulo se presentarán las historias de usuario y arquitectura preliminares del prototipo. Estos dos elementos configuran la planificación y diseño inicial.

Las historias de usuario y arquitectura preliminares serán refinados de manera iterativa e incremental, es decir mediante un enfoque JIT (*Just In Time*) [Wat07]. Este esquema es presentado en la Figura 5.1.

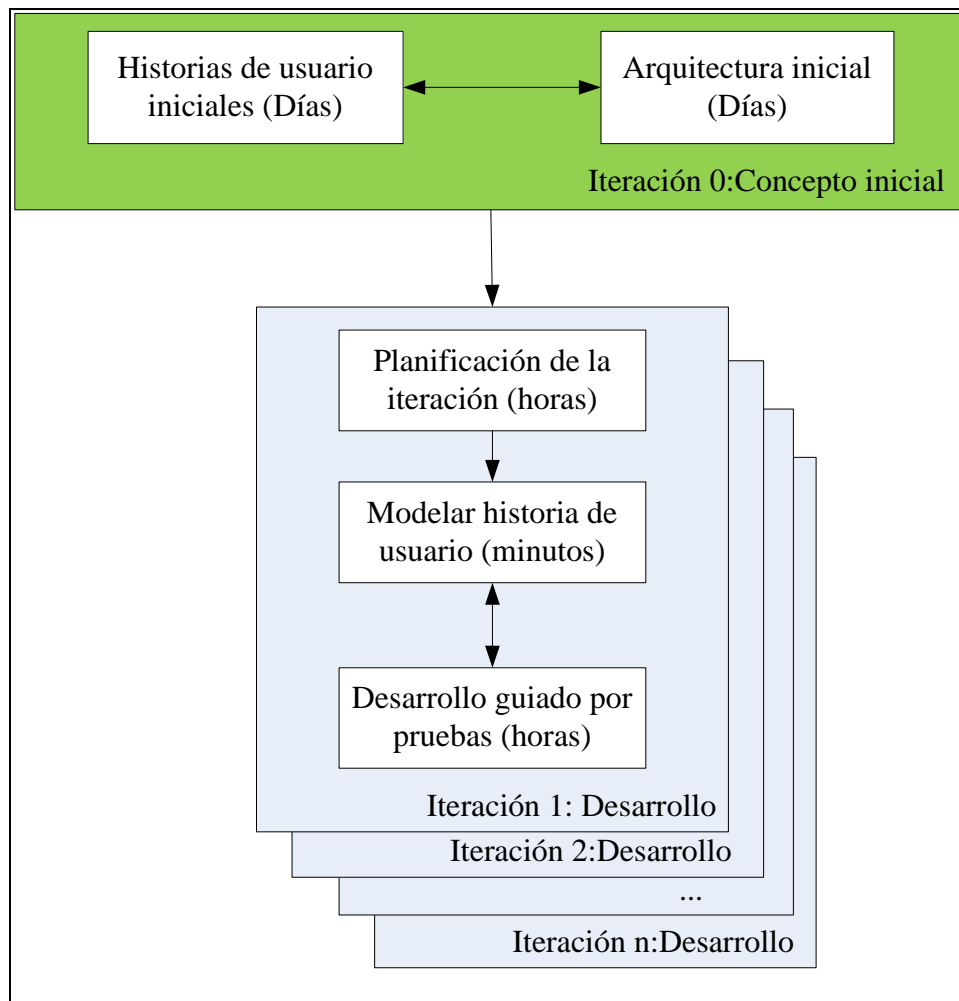


Figura 5.1. Esquema para materializar las historias de usuario.

Con el objetivo de acotar el desarrollo se establecieron los siguientes aspectos:

- **Cantidad de personas involucradas:** En el desarrollo de este proyecto participaron el estudiante tesista como desarrollador, un representante de la empresa patrocinante encargado de especificar y refinar las historias de usuario y un asesor experto con 15 años de trayectoria en dirección de proyectos de infraestructura de empresas de telecomunicaciones.

- **Tiempo de desarrollo:** Para este proyecto se utilizó la técnica conocida como *timeboxing*, esta consiste en utilizar períodos fijos de tiempo con el objetivo de disminuir el riesgo. Cada *timebox* (iteración en XP o sprint en Scrum) debe contar con su presupuesto y entregables. Sin embargo, proyectos completos también pueden ser desarrollados enmarcados en un *timebox* [Lar04,13].

Específicamente, este proyecto se desarrolló durante el plazo fijo de diez semanas.

- **Restricciones:** El lenguaje de programación debe ser PHP5, la persistencia la herramienta debe ser MySQL y la presentación debe utilizar la biblioteca ExtJS²⁷.

Habiendo fijado las restricciones, tiempo de desarrollo y las personas involucradas, se estableció como variable de gestión el alcance del proyecto, el cual es acotado a través de las historias de usuario preliminares.

5.1. Historias de Usuario Preliminares

Las historias de usuario preliminares en el contexto de este Proyecto de Titulación se entenderán como una descripción de funcionalidad de alto nivel expresada desde el punto de vista de un usuario.

En base al ciclo de vida de un trabajo presentado en el capítulo 2 y a las propuestas hechas en el mismo, a continuación se describirán las historias de usuario preliminares.

- A. **Un usuario puede crear una orden de trabajo.** Utilizando una especificación técnica, un usuario puede crear una nueva orden de trabajo, esta debe estar compuesta de actividades, y recursos requeridos. Para la construcción de esta orden de trabajo el usuario puede utilizar trabajos modelo o trabajos frecuentes.
- B. **Un usuario puede valorizar órdenes de trabajo.** Utilizando valores especificados en un contrato entre la empresa mandante y la contratista, es posible valorizar las actividades y materiales.
- C. **Un usuario puede planificar órdenes de trabajo.** Dadas las actividades y recursos requeridos, es posible asignar una fecha de inicio y término estimados, definir puntos de control y planificar recursos.

²⁷ <http://www.extjs.com/>

- D. **Un usuario puede ingresar el estado de una orden de trabajo en ejecución.** Dados los puntos de control en una orden de trabajo es posible ingresar la documentación requerida y el avance de la orden de trabajo.
- E. **Un usuario puede cerrar una orden de trabajo.** Una vez que todas las actividades de han finalizado es posible cerrar la orden de trabajo y recopilar su documentación.
- F. **Mantener un inventario de existencias.** La empresa contratista mantiene recursos propios y entregados por los mandantes, debe ser conocida la cantidad actual en bodega y es posible determinar donde han sido utilizados.

Para solucionar algunas preguntas básicas del dominio y realizar estimaciones se utilizaron un modelo de dominio y *wireframes* o maquetas de interfaces sin diseño gráfico.

Modelo de dominio

Un modelo de dominio, es un modelo conceptual el cual describen las distintas entidades involucradas en el sistema y sus relaciones. Más específicamente, Eric Evans escribe:

“Un modelo de dominio no es un diagrama en particular; es la idea que el diagrama pretende transmitir. No es sólo el conocimiento que tiene un experto sobre el dominio del problema; **Es una abstracción rigurosamente organizada y seleccionada de ese conocimiento.** Un diagrama puede representar un modelo, como también un código cuidadosamente escrito o una frase en inglés.” [Eva04].

Para expresar el modelo de dominio, en esta etapa del desarrollo se utilizará el diagrama en la Figura 5.2.

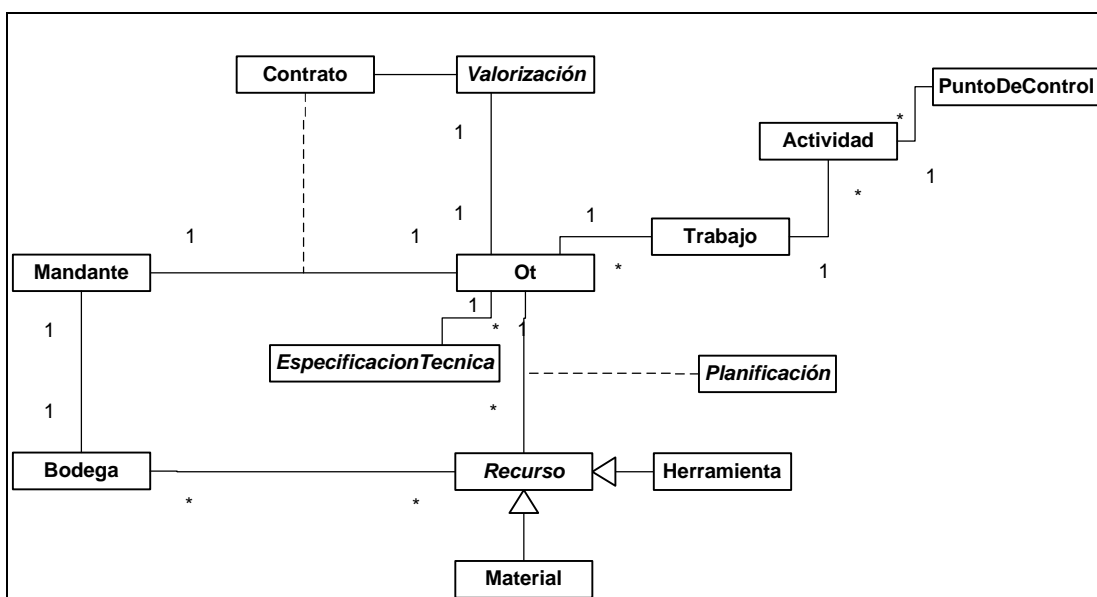


Figura 5.2. Diagrama de dominio.

Tabla 5.1. Estimación de las historias de usuario preliminares.

Índice	Historia	Tiempo estimado
A	Un usuario puede crear una orden de trabajo	4 semanas
B	Un usuario puede valorizar órdenes de trabajo	4 semanas
C	Un usuario puede planificar órdenes de trabajo	7 semanas
D	Un usuario puede ingresar el estado de una orden de trabajo en ejecución	3 semanas
E	Un usuario puede cerrar una orden de trabajo	2 semanas
F	Mantener un inventario de existencias	6 semanas

De las historias anteriormente expuestas fueron consideradas de mayor importancia y para ser desarrolladas en el marco de este Proyecto de Titulación las historias C y D.

5.2. Arquitectura Preliminar

En el desarrollo de este Proyecto de Titulación, el concepto de arquitectura fue entendido como: “aquellas decisiones que deben ser tomadas correctamente temprano durante el desarrollo, puesto que son difíciles de cambiar” [Fow03a]. Las decisiones descritas a continuación son: la selección de un patrón de arquitectura y un *framework* a utilizar.

5.2.1. Selección de un Patrón de Arquitectura

De acuerdo a la definición anterior, algunos patrones pueden ser considerados como arquitectónicos, en el sentido que son difíciles de modificar y tendrán una gran influencia durante el diseño de la aplicación.

En la presente sección se presentan y comparan dos patrones de arquitectura, los que corresponden a Modelo-Vista-Controlador y el propuesto por Eric Evans en su libro *Domain-Driven Design: Tackling Complexity in the Heart of Software* [Eva04].

Ambos patrones separan las distintas responsabilidades en capas o *layers*.

Modelo-Vista-Controlador

Fue descrito por primera vez en 1979 por Trygve Reenskaug [Ree79], el principal objetivo de este patrón es separar los objetos del dominio que son usados para modelar el entendimiento del problema, de los objetos que corresponden a la interfaz gráfica (GUI por sus siglas en inglés).

Las distintas capas de este patrón son descritas a continuación.

- **Modelo:** En esta capa se encuentran los objetos de dominio, los que son llamados “modelos”, y sus relaciones. En esta capa se encuentra expresado el entendimiento que el equipo tiene sobre el dominio del problema.

- Vista: Esta capa es la encargada de presentar al usuario información, y generalmente corresponde a la interfaz de usuario.
- Controlador: Esta capa está encargada de recibir e interpretar las acciones de los usuarios, realizan operaciones con los modelos y permiten la navegación a través de las vistas.

En la Figura 5.4 se muestra el flujo de información entre los distintos elementos de este patrón de arquitectura.

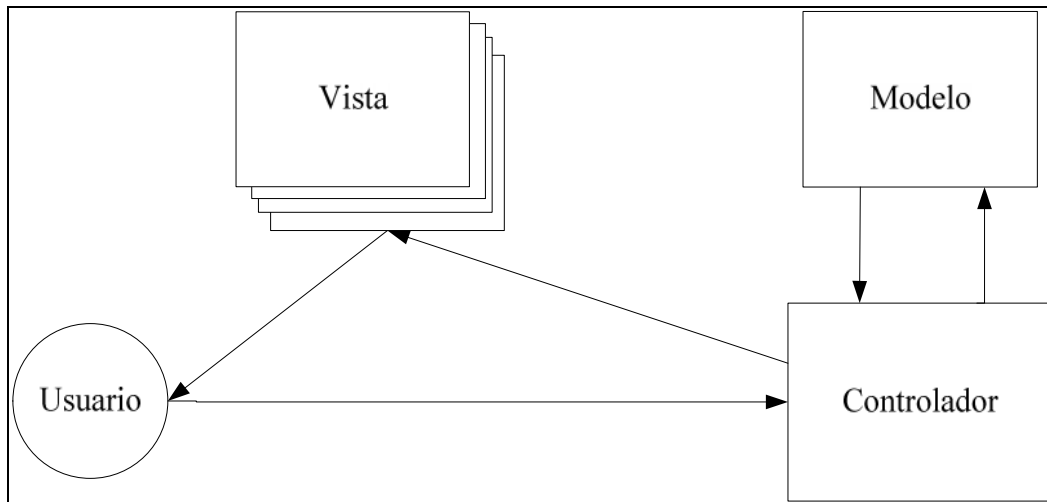


Figura 5.4. Flujo de información en MVC.

La simpleza de este patrón ha influido en su amplia aceptación y uso en aplicaciones empresariales, lo que a su vez ha impulsado que múltiples *frameworks* utilicen MVC como su estructura subyacente.

Patrón propuesto por *Domain-Driven Design*

Domain-Driven Design es un enfoque para el diseño de software, el cual está centrado en aplicaciones de alta complejidad. DDD provee una estructura de prácticas y terminología para que las decisiones de diseño se enfoquen en acelerar el desarrollo. Además plantea un patrón de arquitectura. Este patrón propone que el modelo de dominio debe residir en una capa, de manera tal que se encuentre separado de otros aspectos como la presentación o persistencia. Esta capa es conocida como la capa de dominio. En la Figura 5.5 se presenta la estructura de este patrón de arquitectura.

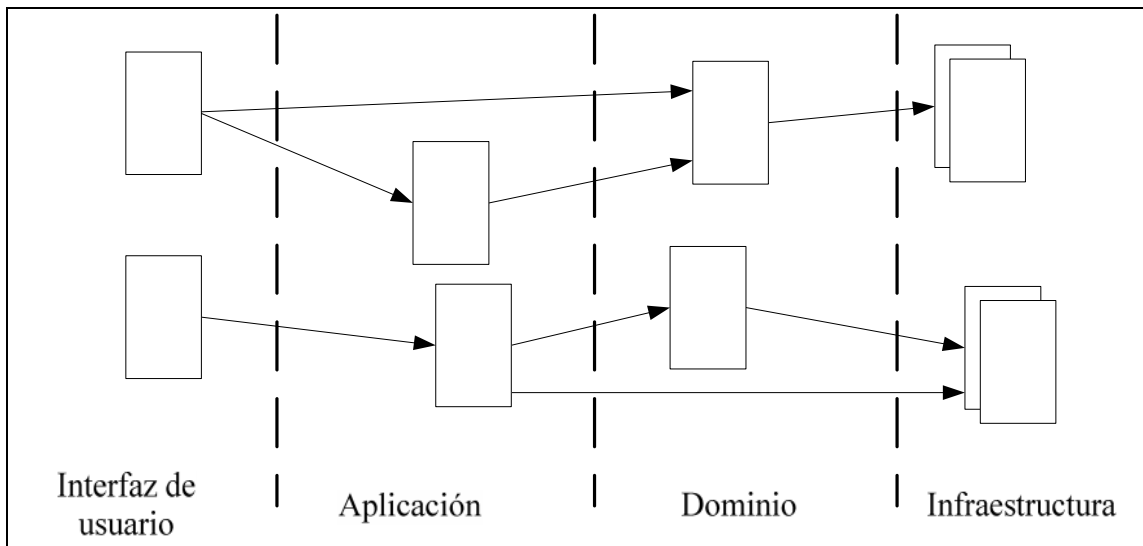


Figura 5.5. Patrón de arquitectura propuesto por DDD.

A continuación se describen todas las capas:

- Interfaz de usuario: Es responsable de presentar la información al usuario e interpretar sus acciones.
- Capa de aplicación: Es responsable de coordinar las actividades en la aplicación. Es la encargada de mantener el estado de la aplicación, sin embargo, no debe contener lógica de dominio.
- Capa de dominio: Contiene toda la información referente al dominio. El estado de los objetos de dominio son mantenidos por esta capa, es la más importante y contiene el corazón de la aplicación.
- Capa de infraestructura: Es la encargada de proveer de persistencia a la capa de dominio.

Sin embargo, su implementación no es trivial, debido principalmente a que es necesario el desarrollo de un lenguaje específico del dominio y la falta de *frameworks* que lo utilicen como estructura subyacente.

Comparación y selección

En la Tabla 5.2 se presenta una comparación entre estos dos patrones de arquitectura. Las variables fueron seleccionadas tomando en cuenta aquellos aspectos considerados de mayor relevancia en la implementación de este proyecto y los valores están basados en la información previamente expuesta.

Tabla 5.2. Comparativa de patrones de arquitectura.

	MVC	DDD
Curva de aprendizaje	Baja	Alta
Frameworks existentes	Múltiples	No
Ambiente ideal	Dominios de complejidad moderada y baja	Dominios de complejidad alta

Teniendo en cuenta que el dominio explicado en el punto 5.1 es considerado de complejidad moderada y los valores presentados en la Tabla 5.1, se seleccionó el patrón de arquitectura MVC.

5.2.2. Selección de un *Framework* de Desarrollo

En este contexto, podemos entender *framework* de desarrollo como un conjunto de bibliotecas que proveen de múltiples funcionalidades y tienen una estructura definida.

Se evaluaron dos alternativas, las cuales implementan el patrón MVC como su estructura subyacente, están escritos en PHP5 y pueden utilizar MySQL como herramienta para la persistencia. A continuación se describen y comparan CodeIgniter y CakePHP.

Descripción de CodeIgniter

Es un *framework* liviano y de código abierto, centrado desarrollar aplicaciones Web de una manera rápida y sencilla. Sus características se presentan en la Tabla 5.3.

Tabla 5.3. Características y funcionalidades de CodeIgniter.

Información básica	
Url	http://codeigniter.com/
Desarrolladores	EllisLab, Inc.
Licencia	Similar a Apache/BSD
Versión evaluada	1.7.1
Funcionalidades	
Abstracción de datos	Cuenta con una clase de acceso a datos que implementa el patrón ActiveRecord
Ajax	No cuenta con librerías incorporadas pero es fácil integrarlas con ayudantes
Componente de autenticación	Debe ser integrado de manera externa
Plantillas	No cuenta con un sistema de plantillas
Generadores de código	No cuenta con generadores de código
Andamios (<i>Scaffolding</i>)	El sistema de andamios se encuentra obsoleto desde la versión 1.6
Caché	Cuenta con un sistema que permite mantener paginas en caché
Internacionalización	Cuenta con un sistema que permite internacionalizar la aplicación
Curva de aprendizaje	Es muy fácil aprender
Pruebas unitarias	Cuenta con una clase para pruebas unitarias, no cuenta con análisis cobertura de código.
Documentación y soporte	
Documentación	Cuenta con una excelente documentación y tutoriales sin embargo no están en español
Soporte	Foros y wikis

Descripción de CakePHP

Es un *framework* centrado en el desarrollo rápido de aplicaciones, inspirado por Ruby On Rails²⁸. Sus características son presentadas en la Tabla 5.4.

Tabla 5.4. Características y funcionalidades de CakePHP.

Información básica	
Url	http://cakephp.org/
Desarrolladores	Cake Software Foundation, Inc.
Licencia	MIT
Versión evaluada	1.2.3
Funcionalidades	
Abstracción de datos	Cuenta con un ORM que es implementado a través del patrón ActiveRecord
Ajax	Integra JQuery y prototype
Componente de autenticación	Integra un componente de autenticación
Plantillas	Integra un sistema de plantillas
Generadores de código	Cuenta un sistema a través de consola
Andamios (Scaffolding)	Cuenta con un sistema de andamios
Caché	Permite guardar en caché paginas, objetos y consultas
Internacionalización	Cuenta con un sistema que permite internacionalizar la aplicación
Curva de aprendizaje	Es fácil de aprender
Pruebas unitarias	Integra el entorno simpletest para pruebas unitarias
Documentación y soporte	
Documentación	Cuenta con excelente documentación y tutoriales
Soporte	Cuenta con un google-group

Comparación

Con el objetivo de comparar estos dos *frameworks*, se asignaron puntajes a cada una de las características, los puntos serán asignados según la Tabla 5.5.

Tabla 5.5. Sistema de puntaje para evaluar *frameworks*.

No existente	0
Limitado	1
Suficiente	2
Bueno	3
Sobresaliente	4

²⁸ <http://rubyonrails.org/>

A continuación en la Tabla 5.6 se muestra la comparación de estos dos *frameworks*. Las características fueron seleccionadas basadas en los aspectos considerados de mayor importancia para el desarrollo del proyecto.

Tabla 5.6. Comparación entre *frameworks* de desarrollo.

	CodeIgniter	CakePHP
Funcionalidades		
Abstracción de datos	2	3
Ajax	3	3
Componente de autenticación	3	3
Generadores de código	0	3
Andamios (Scaffolding)	1	3
Caché	1	4
Internacionalización	3	3
Facilidad de aprendizaje	4	2
Pruebas unitarias	1	4
Documentación y soporte		
Manuales	3	4
Comunidad	4	3
Total	25	35

Las diferencias observadas en la Tabla 5.6 se deben principalmente a que CodeIgniter es un *framework* más liviano y simple que CakePHP.

Basado en los resultados descritos en la Tabla 5.6, se decidió utilizar CakePHP como *framework* de desarrollo.

6. ADAPTACIÓN DE LAS PRÁCTICAS DE *EXTREME PROGRAMMING*

“Un hombre con papel, lápiz y goma, y sujeto a una disciplina estricta, es en efecto una maquina universal.”

Alan Turing 1912-1954.

En el presente capítulo se describen las prácticas y las métricas usadas durante el desarrollo del prototipo.

6.1. Prácticas

A continuación se describirán las siguientes prácticas utilizadas por el equipo de desarrollo: trabajar con energía, historias de usuario, espacio informativo, ciclo semanal, integración continua, desarrollo guiado por pruebas, refactorización y reuniones *stand-up*.

Trabajar con energía

El desarrollo tomó lugar en las instalaciones de la empresa patrocinante, en un espacio abierto, compartido con otros desarrolladores. El horario de trabajo fue de lunes a viernes, en las mañanas desde las 9 a 13 horas y en las tardes desde 15 a 19 horas, a excepción de los días viernes donde se trabajó hasta las 17 horas. Esto suma un total de 38 horas semanales. Este horario fue, según el equipo desarrollador, uno que resultaba sostenible, pues resulta en una buena productividad y a su vez garantiza un buen descanso. Es importante hacer notar que durante el desarrollo no se trabajaron horas extras.

Historias de usuario

Las historias de usuario fueron escritas al comienzo de cada iteración por el asesor experto de la empresa patrocinante, estas historias contienen un nombre, una descripción y una estimación hecha por el desarrollador. Estas historias fueron divididas en tareas o trabajos específicos. Las historias y tareas son físicamente expresadas a través de tarjetas. Más específicamente, se utilizaron tarjetas blancas para expresar historias de usuario, las cuales en la parte frontal mostraban su nombre, descripción y estimación, mientras que en la parte posterior contenían el tiempo que tomó su compleción. De la misma manera, las tareas fueron escritas en tarjetas de color amarillo, las cuales en la parte frontal tenían un nombre, descripción o notas y una estimación, mientras que en su parte posterior contenían posibles pruebas unitarias y el tiempo real que tomado para su compleción. Un ejemplo de esto es presentado en la Figura 6.1 en donde se muestra como fueron expresadas las historias de usuario y tareas.

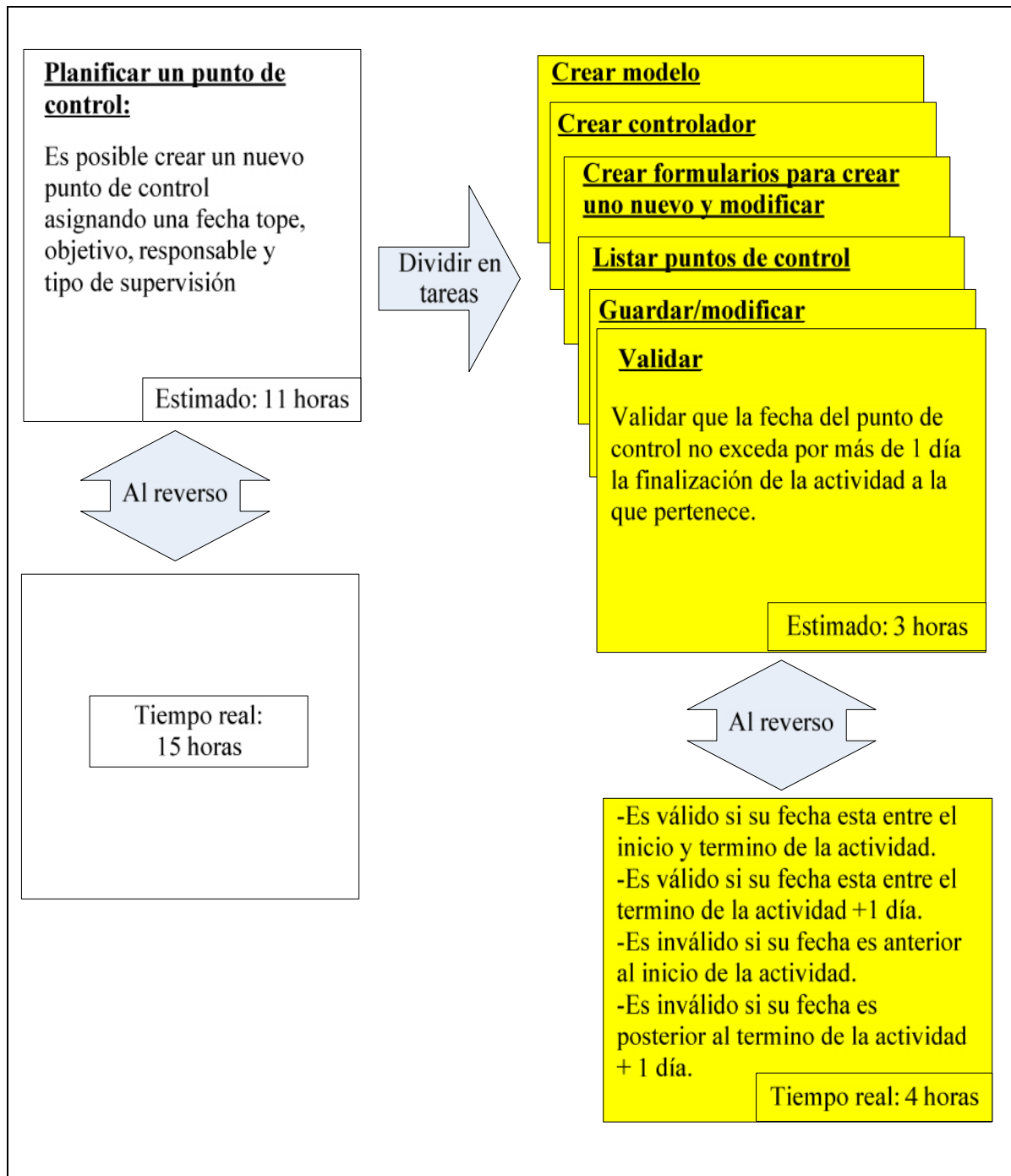


Figura 6.1. Historias de usuario y tareas.

Espacio informativo

Se implementó un panel informativo que contenía las historias de usuario y tareas para ser implementadas durante una iteración, este fue dividido en dos partes, la parte de la izquierda estaba destinada a mostrar aquellas tareas que están por ser comenzadas y aquellas en el lado derecho han sido terminadas. En la Figura 6.2 se muestra la disposición del panel informativo.

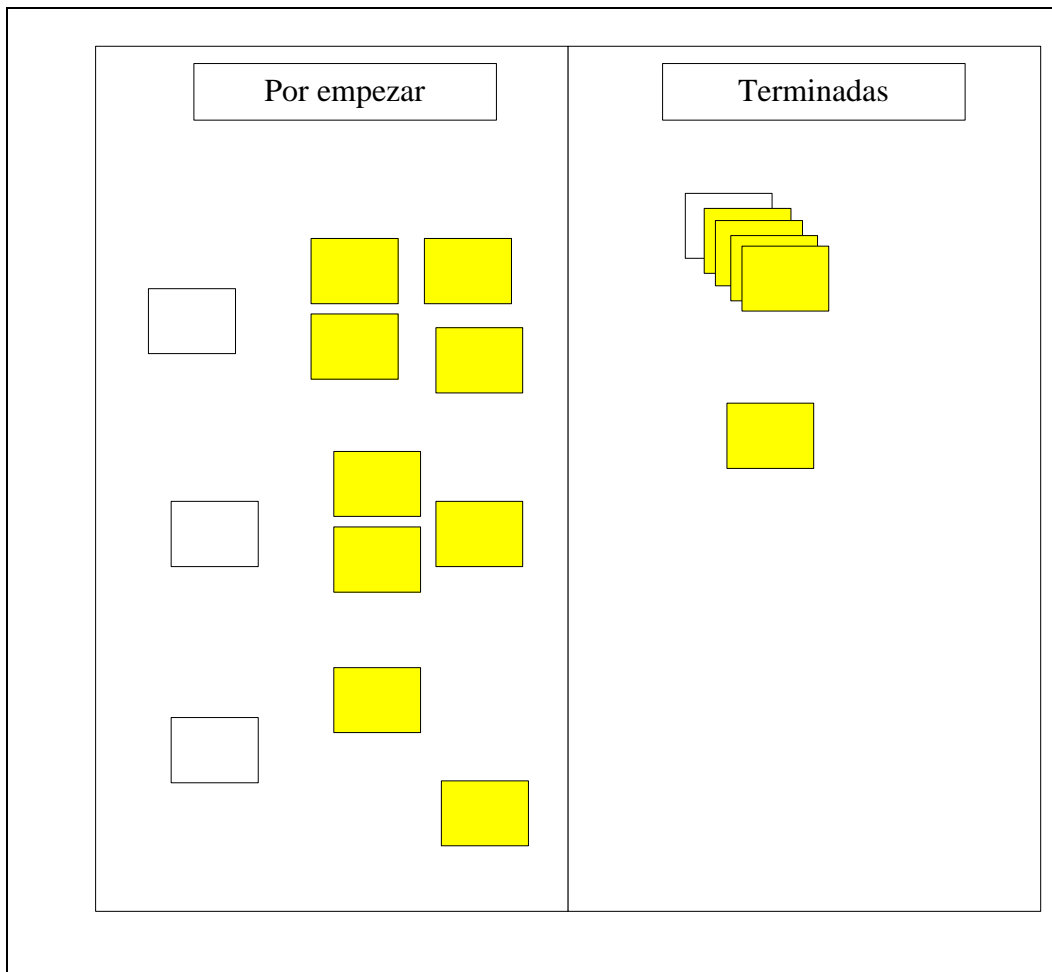


Figura 6.2. Panel informativo.

Aquellas tareas que se encontraban en proceso de implementación, se encontraban pegadas en el monitor del equipo de desarrollo. Es importante hacer notar, que con el paso del tiempo se hizo necesaria la implementación de otro espacio dentro del panel que contuviese aquellas historias que no podían ser implementadas durante la iteración, pero debían ser contempladas a futuro.

Ciclo semanal

Las iteraciones tenían una duración de una semana. Al inicio de cada iteración, se realizó una reunión llamada *Iteration Planning Meeting*, esta reunión tenía una extensión de aproximadamente noventa minutos y se realizaba lo siguiente:

- Analizar los resultados de la iteración anterior.
- Re-estimar aquellas historias de usuario y tareas de la iteración anterior que no se completaron.
- Aquellos aspectos del prototipo que no resultaron satisfactorios, son incluidos como nuevas historias.

- Se proponen nuevas historias de usuario y se estiman.
- Se priorizan las historias de usuario.
- Las historias de usuario son traducidas a tareas.
- Proponer mejoras para la siguiente iteración.

Estas historias de usuario fueron dispuestas en el panel del proyecto desde arriba hacia abajo de acuerdo a su prioridad.

Al finalizar la semana, se subía el incremento al servidor de producción y se preparaban el resultado de las métricas para la próxima *iteration planning meeting*.

Integración continua

El código fue mantenido en un repositorio, el cual fue implementado a través de un sistema de control de versiones usando el software Subversion²⁹. La frecuencia con que los cambios eran añadidos al repositorio era de una vez al día.

Para disminuir el tiempo y la cantidad de errores que tomaba subir el sistema al servidor de producción, este proceso se automatizó utilizando el software Ant³⁰.

Desarrollo guiado por pruebas

Para las pruebas unitarias sobre modelos, controladores y ayudantes, se utilizó el entorno de pruebas Simpletest integrado en CakePHP. Todas las pruebas unitarias están escritas en archivos ubicados en el directorio “tests”. Además, debido a que CakePHP implementa el patrón ActiveRecord³¹, es necesario contar con *fixtures* (datos de prueba).

Las pruebas unitarias buscan traducir las tareas en código, por lo que es importante que los títulos indiquen su objetivo. A modo de ejemplo se muestran las pruebas unitarias correspondientes a la validación de un punto de control, para lo cual se utilizan los *fixtures* representados en la Figura 6.3.

²⁹ <http://subversion.apache.org/>

³⁰ <http://ant.apache.org/>

³¹ <http://martinfowler.com/eaCatalog/activeRecord.html>

ActividadFixture	PuntoDeControlFixture
Id: 7 Nombre: 'actividad siete' Inicio: '2009-08-28' Termino: '2009-09-04'	Id: 1 Titulo : 'Punto de control válido 1' Fecha : '2009-09-01'
	Id: 2 Titulo : 'Punto de control válido 2' Fecha : '2009-09-05'
	Id: 3 Titulo : 'Punto de control inválido 1' Fecha : '2009-09-06'
	Id: 4 Titulo : 'Punto de control inválido 2' Fecha : '2009-08-27'

Figura 6.3. *Fixtures* de la actividad y sus puntos de control.

A continuación –y a modo de ejemplo– se muestra el código de estas pruebas unitarias.

```

class PuntoDeControlTestCase extends CakeTestCase {
    var $PuntoDeControl = null;
    function testEsValidoSiLaFechaEstaEntreElInicioYTerminoDeSuActividad(){
        $this->PuntoDeControl->id = '1';
        $resultado = $this->PuntoDeControl->_validaFecha();
        $this->assertTrue($resultado);
    }
    function testEsValidoSiLaFechaEstaEntreElTerminoDeSuActividadYUnDia(){
        $this->PuntoDeControl->id = '2';
        $resultado = $this->PuntoDeControl->_validaFecha();
        $this->assertTrue($resultado);
    }
    function testEsInvalidoSiLaFechaEsMayorAlTerminoDeLaActividadMasUnDia(){
        $this->PuntoDeControl->id = '3';
        $resultado = $this->PuntoDeControl->_validaFecha();
        $this->assertFalse($resultado);
    }
    function testEsInvalidoSiLaFechaEsAntesDelInicioDeSuActividad(){
        $this->PuntoDeControl->id = '4';
        $resultado = $this->PuntoDeControl->_validaFecha();
        $this->assertFalse($resultado);
    }
}

```

Las pruebas al ser ejecutadas a través de un navegador Web, se ven como la Figura 6.4.

Individual test case: models\punto_de_control.test.php

Tests del Modelo Punto de Control

Es Valido Si La Fecha Esta Entre El Inicio Y Termino De Su Actividad
 Es Valido Si La Fecha Esta Entre El Termino De Su Actividad Y Un Dia
 Es Invalido Si La Fecha Es Mayor Al Termino De La Actividad Mas Un Dia
 Es Invalido Si La Fecha Es Antes Del Inicio De Su Actividad

1/1 test cases complete: 4 passes, 0 fails and 0 exceptions.

Figura 6.4. Ejecución de pruebas unitarias.

Refactorización

La refactorización fue llevada a cabo principalmente de manera manual, debido a que al momento del desarrollo, no se encontraron IDEs que proveyeran de refactorización automática para el lenguaje PHP. Sin embargo, Netbeans³² cuenta con algunas funcionalidades básicas, entre ellas renombrar una variable o función.

Reuniones *stand-up*

Con una mayor frecuencia que las *iteration planning meetings*, se realizaron reuniones cuya duración no excedía los quince minutos, y que buscaban resolver dudas sobre el negocio y validar funcionalidades.

6.2. Métricas

Las métricas en el contexto de la ingeniería de software se entienden como medidas obtenidas a partir de un desarrollo, para conocer el estado de algún aspecto del mismo.

A continuación se describen las siguientes métricas utilizadas por el equipo durante el desarrollo del proyecto: velocidad, precisión de la estimación y cobertura de código.

Velocidad

A medida que las iteraciones avanzan, es posible medir la velocidad del proyecto, esto es, la cantidad de trabajo que es posible realizar durante una iteración. Para este proyecto la velocidad fue entendida como las horas estimadas de las historias de usuario completadas durante una iteración. Este índice es considerado al planificar la siguiente iteración, donde se debe proyectar la misma cantidad de horas que fueron trabajadas en la iteración anterior, este concepto es conocido como *Yesterday's weather*³³ (el clima de ayer).

Precisión de la estimación

De la misma manera, la proporción entre el tiempo real tomado por una historia de usuario y su estimación, representa la precisión con la que se logran las estimaciones. El objetivo es mejorar la calidad de las estimaciones para proyectos e iteraciones posteriores.

A modo de ejemplo se presenta en la Tabla 6.1 el resumen de las historias de la quinta iteración que comenzó el lunes 21 de septiembre de 2009 y concluyó el viernes de la misma semana.

³² <http://www.netbeans.org/>

³³ <http://ns.c2.com/cgi/wiki/YesterdaysWeather>

Tabla 6.1. Velocidad y precisión de la estimación en la quinta iteración.

Historias de usuario	Estimado	Real
Agregar Herramientas	3	5
Correcciones	3	5
Limpiar encabezados código	1	1
Planificar un punto de control	11	15
Guardar la planificación	9	10
Administración	1	1
Total	28	35

Velocidad	28
Precisión de la estimación	80%

Cobertura de código

Corresponde al porcentaje de de código abarcado por las pruebas unitarias. El análisis de cobertura de código está basado en las líneas de código ejecutadas por las pruebas unitarias. A modo de ejemplo, la Figura 6.5 muestra las pruebas unitarias correspondientes a la validación de un punto de control y el análisis de cobertura de código indica que se ha cubierto un 93,75% del total de líneas de código y además indica cuales son aquellas no abarcadas, esto es muestra de que una nueva prueba unitaria es necesaria.



Figura 6.5. Cobertura de código.

En la Figura 6.6 se presenta la existencia de una nueva prueba unitaria y un 100% de cobertura de código.

Tests del Modelo Punto de Control

Es Valido Si La Fecha Esta Entre El Inicio Y Termino De Su Actividad
Es Valido Si La Fecha Esta Entre El Termino De Su Actividad Y Un Dia
Es Invalido Si La Fecha Es Mayor Al Termino De La Actividad Mas Un Dia
Es Invalido Si La Fecha Es Antes Del Inicio De Su Actividad
Es Invalido Si Su Fecha Es Vacía

1/1 test cases complete: 5 passes, 0 fails and 0 exceptions.

Code Coverage: 100%

Figura 6.6. Cobertura de código completa para el modelo Punto de Control. El análisis de cobertura de código es provisto por los entornos de pruebas, para el caso específico de este desarrollo, es suministrado por Simpletest.

7. PROTOTIPO

“Lo supremo en el arte de la guerra consiste en someter al enemigo sin darle batalla.”

Sun Tzu.

En el presente capítulo se presentan los aspectos más significativos del proceso de desarrollo, el producto y el resultado de las métricas.

7.1. Proceso de Desarrollo

El desarrollo del prototipo estuvo comprendido por un total de diez iteraciones, cada una tuvo un largo de una semana. A continuación se describirán de manera general las iteraciones y los principales logros de cada una.

Primera iteración

La primera iteración estuvo centrada en la instalación del *framework* y en el desarrollo del entorno gráfico, se definió que la navegabilidad sería utilizando un sistema de “pestañas” y menús laterales como es mostrado en la Figura 7.1. Se lograron listar dos órdenes de trabajo que estaban aceptadas como puede ser observado en la Figura 7.2.

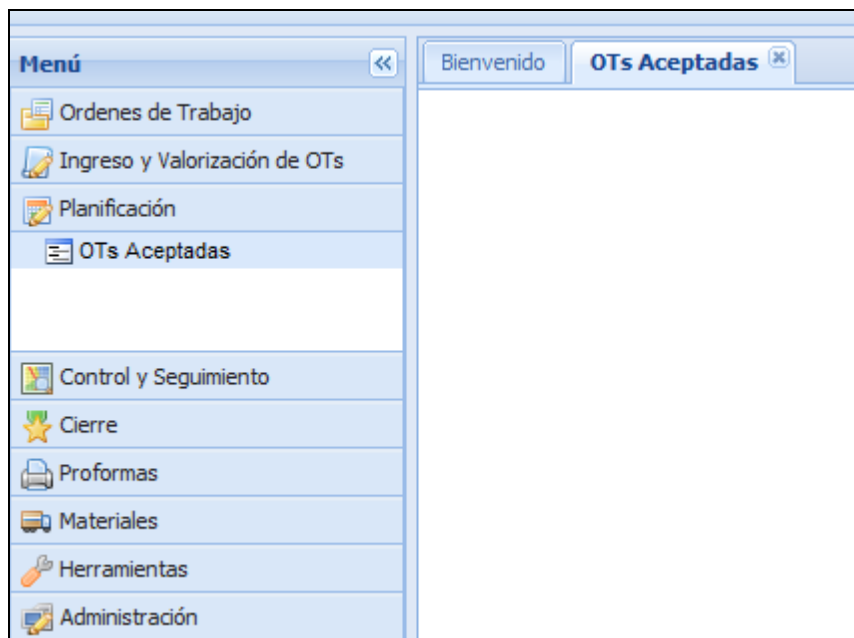


Figura 7.1. Menú lateral

Número	Mandante	Descripción	Estado	Acciones
123456	Perceptum	Ensamblar una casa	aceptada	
654321	Perceptum	Construcción de casetas en P...	aceptada	

Figura 7.2. Dos órdenes de trabajo desplegadas.

Las pruebas unitarias escritas y pasadas en esta etapa del desarrollo se pueden encontrar listadas en la Tabla 7.1.

Tabla 7.1. Pruebas unitarias de la primera iteración.

Controlador	Prueba unitaria
Main	Index
	Lista Menus Correctamente
	Menu Lista Ots Aceptadas
OtsAceptadas	Lista Ots Aceptadas
	Lista Ots Aceptadas Con Un Mandante
	Las Acciones De Ots Aceptadas Tienen Id Texto Y Controller
	Controller De La Accion Planificar Esta Correcto
	Controller De La Accion Cancelar Esta Correcto

El modelo de dominio (capa de modelos en MVC) en esta etapa del desarrollo se encuentran en una fase inicial y solamente existen los modelos más necesarios para expresar las funciones antes descritas. El modelo de dominio en esta etapa puede ser observado en la Figura 7.3.

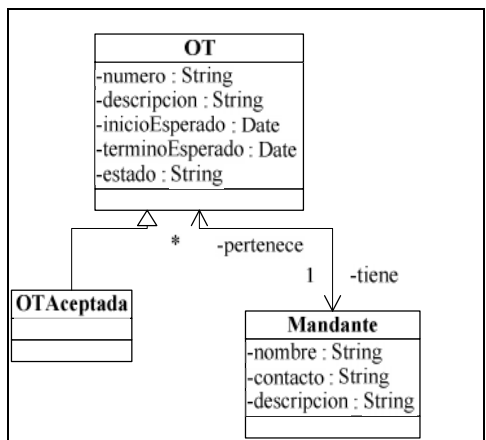


Figura 7.3. Modelo de dominio en la primera iteración.

Segunda y tercera iteraciones

La segunda y tercera iteraciones estuvieron centradas en el crecimiento del modelo de dominio, es decir, los distintos modelos y sus relaciones, la cual se mantuvo sin mayores alteraciones durante el resto del desarrollo.

El modelo de dominio presentado en la Figura 7.3 evolucionó para integrar modelos relativos a la planificación de una orden de trabajo. Esto puede ser observado en la Figura 7.4.

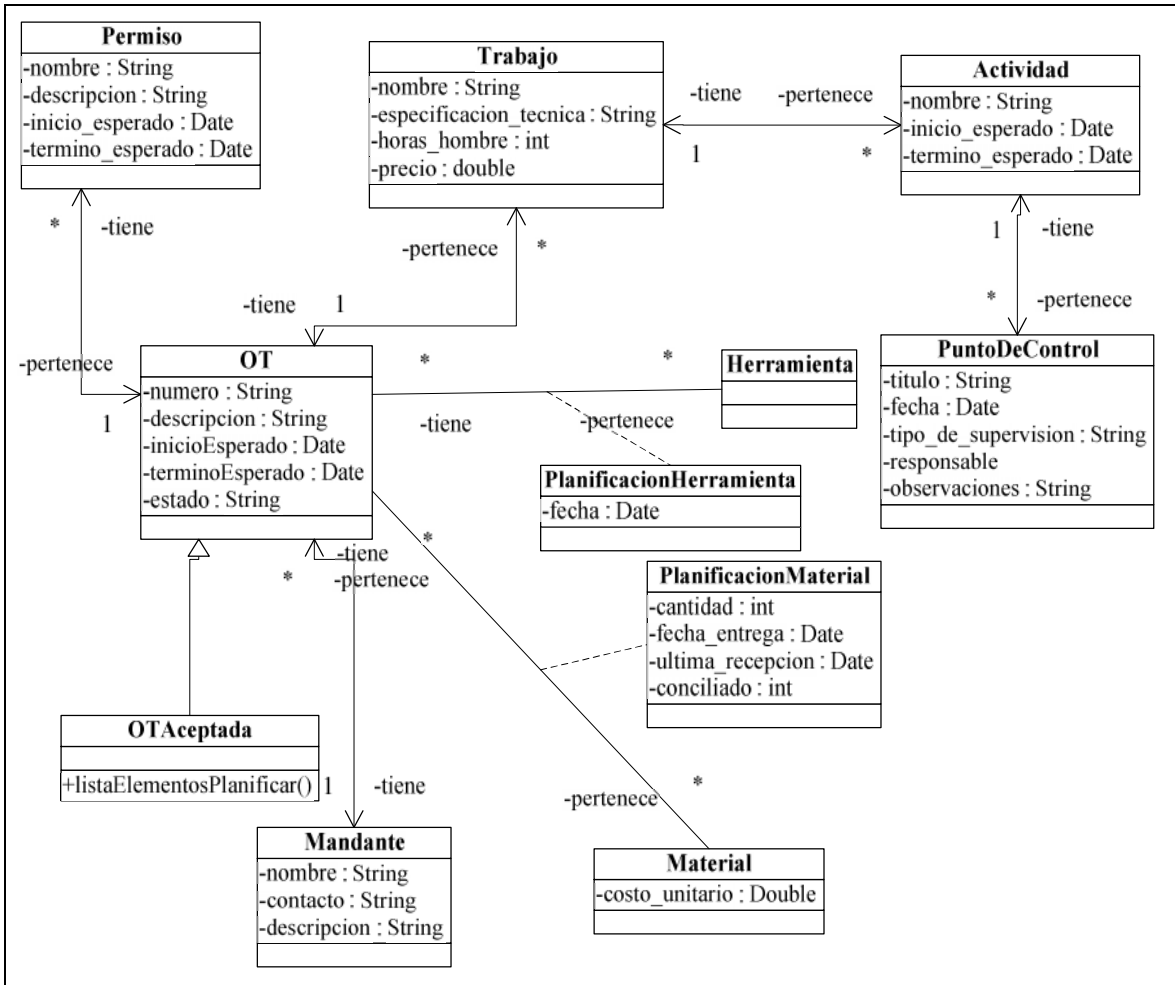


Figura 7.4. Diagrama de modelos en la cuarta iteración.

En cuanto a la interfaz, a los elementos presentados en la Figura 7.1 y 7.2 se agregó una nueva “pestaña” encargada de desplegar los nuevos componentes de una orden de trabajo, como es observado en la Figura 7.5. Es necesario hacer notar que en esta etapa no se puede ejecutar ninguna acción con estos componentes.

Permisos	Inicio	Termino
Permiso Municipal A	12/11/2009	21/12/2009

Actividad	Inicio	Termino
Trabajo 1 (1 Actividad)		
Actividad 1	11/12/2009	21/12/2009
Trabajo 2 (1 Actividad)		
Actividad 3	11/12/2009	21/12/2009

Titulo	Fecha
Actividad 1 (1 Punto de Control)	
Punto de Cotrol Uno	12/12/2009
Actividad 3 (1 Punto de Control)	
Punto de Control 3	12/12/2009

Cantidad	Material	Entrega Planificada
1	torre de 25 mts	12/11/2010
20 mts[mt]	cable 540	12/12/2010

Nombre
Herramienta 1

Figura 7.5. Componentes de una orden de trabajo en la tercera iteración.

A las pruebas unitarias presentadas en la Tabla 7.1 se añadieron y pasaron las listadas en la Tabla 7.2.

Tabla 7.2. Pruebas unitarias nuevas en la cuarta iteración.

Controlador	Prueba unitaria
Actividades	Lista Correctamente Actividades Para Una Ot En Planificacion
Permisos	Lista Permisos Para Una Ot En Planificacion
OtsAceptadas	La Vista De Planificar Una Ot Trae Todos Los Divs
	Devuelve Los Elementos De Una Ot
PlanificacionHerramientas	Listar Planificaciones Para Una Ot
PlanificaciónMateriales	Lista Planificaciones Para Una OT
PuntosDeControl	Lista Puntos De Control Para Una Ot En Planificacion
Modelo	Pruebas unitarias
OtAceptada	Lista Todos Los Elementos De Una OT

Cuarta y quinta iteraciones

La cuarta y quinta iteraciones estuvieron centradas en la implementación de las acciones a realizar por parte del usuario, es decir agregar y modificar puntos de control, herramientas,

materiales y planificar actividades y trabajos. Para lo cual, a las pruebas unitarias listadas anteriormente, se añadieron y pasaron las presentadas en la Tabla 7.3.

Tabla 7.3. Pruebas unitarias nuevas en la quinta iteración.

Controlador	Prueba unitaria
Actividades	Devuelve Correctamente Su Planificacion
	Guarda Correctamente La Planificacion
	Mensaje De Error Si No Existe Data
Permisos	Guarda Correctamente La Planificacion
	Mensaje De Error Si No Existe Data
PlanificacionHerramientas	Despliega Formulario De Seleccionar Herramientas
	Modifica Planificacion Correctamente
	Modifica Y No Crea Elementos De La Planificacion Nuevos
	Elimina Una Planificacion
PlanificaciónMateriales	Despliega Formulario De Planificar Material
	Guarda Planificacion Correctamente
	Guarda Nueva Planificacion Correctamente
	Guarda Nueva Planificacion Correctamente Del Mismo Material En Una Ot
	Elimina Planificacion De Material
PuntosDeControl	Despliega Formulario Planificar
	Guarda Correctamente La Planificacion
	Mensaje De Error Si No Existe Data
	Guarda Correctamente La Nueva Planificacion
	Elimina Correctamente Un Punto De Control
Modelo	Pruebas unitarias
PuntoDeControl	Es Valido Si La Fecha Esta Entre El Inicio Y Termino De Su Actividad
	Es Valido Si La Fecha Esta Entre El Termino De Su Actividad Y Un Dia
	Es Invalido Si La Fecha Es Mayor Al Termino De La Actividad Mas Un Dia
	Es Invalido Si La Fecha Es Antes Del Inicio De Su Actividad

Además se agregaron en nuevos elementos de la capa de las vistas. A modo de ejemplo se pueden contar ventanas emergentes que permiten modificar y definir los distintos elementos de planificación, algunas de estas ventanas se muestran en la Figura 7.6.



Figura 7.6. Nuevas ventanas en la quinta iteración.

Sexta y séptima iteraciones

La sexta y séptima iteraciones estuvieron centradas en mejorar los aspectos visuales y la velocidad de las acciones terminadas en las iteraciones anteriores, esto implicó la creación y modificación de funcionalidades. Por ejemplo, se creó un módulo para entregar avisos, como se muestra en la Figura 7.7.

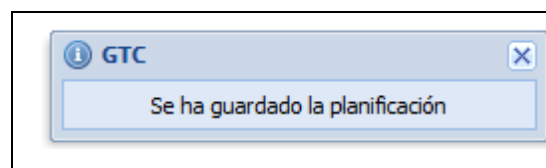


Figura 7.7. Ventana de avisos.

Además, se implementó un modelo llamado `ClasificacionHerramienta` para el despliegue de las distintas herramientas de acuerdo a una clasificación, esta modificación puede ser observada en la Figura 7.8.

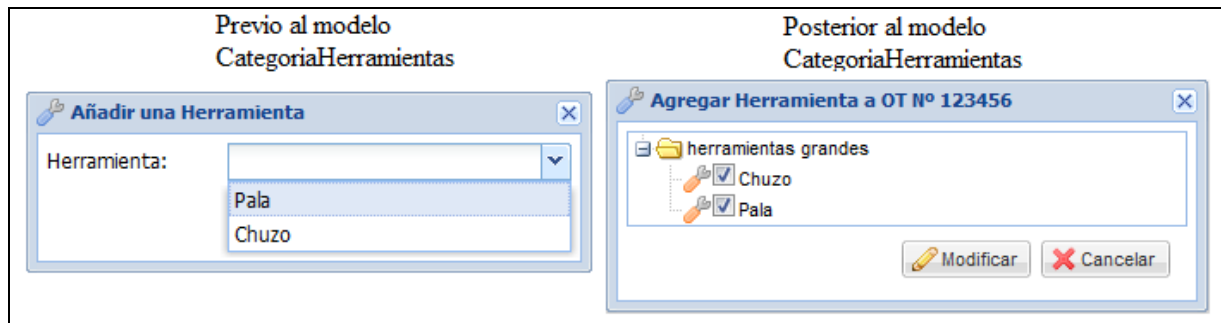


Figura 7.8. Selección de una herramienta.

Por otro lado se creó un modelo abstracto *Recurso*, que permita la extensión del modelo de dominio presentado en la Figura 7.4. El modelo de dominio al finalizar la séptima iteración puede ser observado en la Figura 7.9.

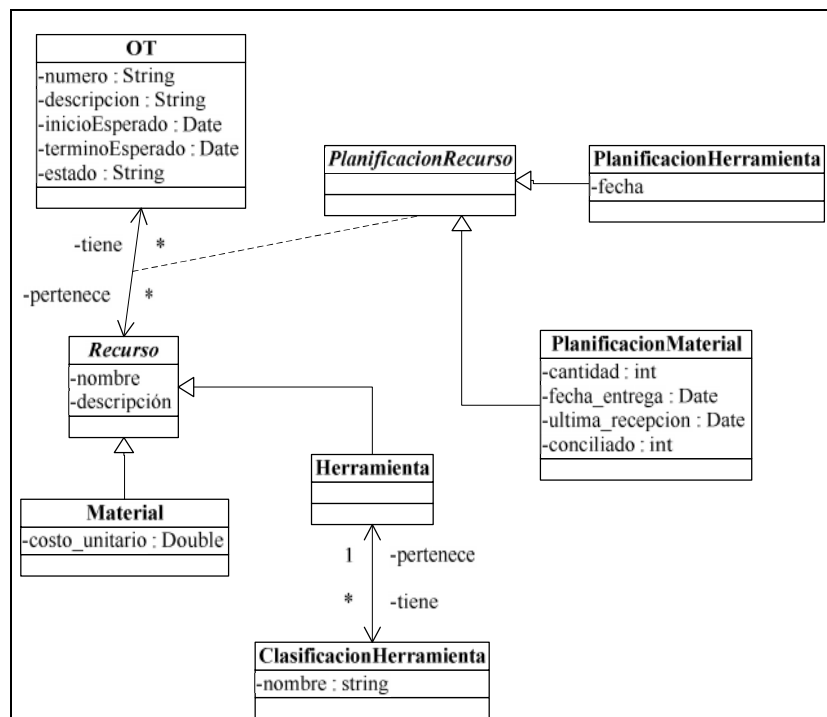


Figura 7.9. Modelo de dominio al finalizar la séptima iteración.

Octava y novena iteraciones

La octava y novena iteraciones estuvieron centradas en marcar una orden de trabajo como planificada, esto involucró la validación de los distintos elementos de una orden de trabajo. Además, se implementó un calendario que resume todas las actividades y puntos de control, como se muestra en la Figura 7.10.

◀		▶		Hoy		Sep 28 — Oct 4 2009					Mes	Semana	Día
Lun 9/28	Mar 9/29	Mie 9/30	Jue 10/1	Vie 10/2	Sab 10/3	Dom 10/4							
Permiso Municipal A													
Instalar Piso													
Paneles	Vigas techo	costaneras											

Figura 7.10. Calendario.

Asimismo, se creó el modelo `Usuario`, en base al cual se estableció el módulo de acceso a usuarios y se modificó la definición de un punto de control, estos cambios son visibles en la Figura 7.11 y 7.12.

Figura 7.11. Ingreso de usuarios.

Figura 7.12. Asignar un responsable a un punto de control.

Durante este período se añadieron y pasaron las pruebas unitarias listadas en la Tabla 7.4.

Tabla 7.4. Pruebas unitarias nuevas al finalizar la novena iteración.

Controlador	Prueba unitaria
Main	Trae Todos Los Elementos Planificables Para El Calendario
OtsAceptadas	Devuelve Error Si Existe Una Ot Mal Planificada
	Contiene Todos Los Elementos Del Excel
	Marcar Ot Como Planificada Correctamente
PlanificacionHerramientas	Obtiene Herramientas Arbol
PlanificaciónMateriales	Despliega Correctamente Aviso De Eliminar
Modelo	Pruebas unitarias
Actividad	Guarda La Fecha De Inicio Y Termino Como Null Si Es Vacía
Ot	Obtiene Fechas Permisos
	Obtiene Fechas Actividades
	Obtiene Fechas Puntos De Control
	Obtiene Fechas Entrega De Materiales
OtAceptada	Valida Permisos Correctamente
	Valida Permiso Errado
	Valida Planificacion Correctamente
	Valida Planificacion Si Existe Un Permiso No Planificado
	No Marca Como Planificada Si Existe Un Permiso No Planificado
	Valida Actividad Correctamente
	Valida Actividad Errada
	No Marca Como Planificada Si Existe Una Actividad No Planificada
	No Valida Permiso Si La Configuración Es Falsa
	No Valida Actividades Si La Configuración Es Falsa
Permiso	Guarda La Fecha De Inicio Y Termino Como Null Si Es Vacía

Décima iteración

La décima y última iteración se centró en implementar la descarga de la planificación de una orden de trabajo como un archivo de tipo Excel, para implementar esta funcionalidad, se añadió y pasó la prueba unitaria presentada en la Tabla 7.5.

Tabla 7.5. Pruebas unitarias nuevas en la décima iteración.

Controlador	Prueba unitaria
OtsAceptadas	Contiene Todos Los Elementos Del Excel

Además se muestra una captura de pantalla de la hoja de cálculo resultante en la Figura 7.13.

N° Orden de Trabajo	123456		
Mandante	Perceptum		
Inicio			
Término			
Permisos			
Permisos	Descripcion	Inicio	Término
Permiso Municipal A	una descripción de permiso	25-09-2009	28-11-2009
Trabajos y Actividades			
Trabajo 1			
Actividad		Inicio	Término
Instalar Piso		25-09-2009	28-11-2009
Trabajo 2			
Actividad		Inicio	Término
Instalar Techo		13-10-2009	02-11-2009
Puntos de Control			
Objetivo	Tipo	Responsable	Fecha
Revisión de pilotes	Administrativo	admin	26-09-2009
Piso	Administrativo	admin	27-09-2009
Paneles	Logístico	admin	28-09-2009
Vigas techo	De Terreno	admin	29-09-2009
costaneras	Logístico	admin	30-09-2009
Materiales			
Material	Cantidad		Entrega Planificada
amarras plásticas	12		10-03-2010
cable 540	3 [mt]		13-05-2010
Herramientas			
Nombre		Categoría	
Chuzo		herramientas grandes	
Pala		herramientas grandes	

Figura 7.13. Planificación de una orden de trabajo en Excel.

7.2.Resultado de las Métricas

A continuación se presenta el resultado de las métricas obtenidas durante el desarrollo de este proyecto.

Velocidad y precisión de la estimación

En la Tabla 7.6 se encuentran: la cantidad de horas trabajadas, velocidad y precisión de la estimación por cada una de las iteraciones.

Tabla 7.6. Horas trabajadas, velocidad y precisión por cada una de las iteraciones.

Iteraciones	Inicio	Término	Horas trabajadas	Velocidad	Precisión
Iteración 1	24-08-2009	28-08-2009	35	22	62,86%
Iteración 2	31-08-2009	04-09-2009	35	23	65,71%
Iteración 3	07-09-2009	11-09-2009	35	28	80,00%
Iteración 4	14-09-2009	17-09-2009	29	23	79,31%
Iteración 5	21-09-2009	25-09-2009	35	25	71,43%
Iteración 6	28-09-2009	02-10-2009	35	29	82,86%
Iteración 7	05-10-2009	09-10-2009	35	30	85,71%
Iteración 8	13-10-2009	16-10-2009	27	22	81,48%
Iteración 9	19-10-2009	23-10-2009	35	30	85,71%
Iteración 10	26-10-2009	30-10-2009	35	31	88,57%

La velocidad y precisión son graficados en la Figura 7.14 y Figura 7.15 respectivamente.

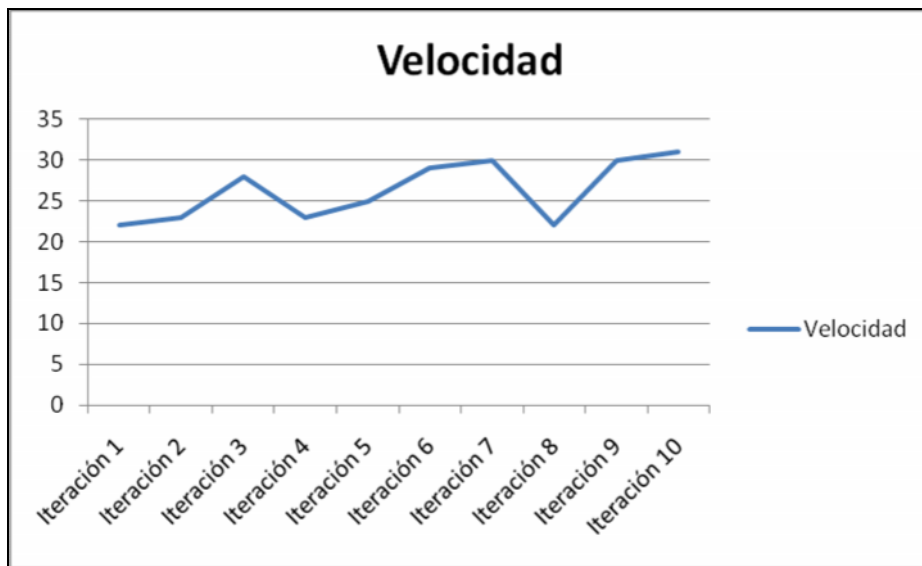


Figura 7.14. Velocidad.

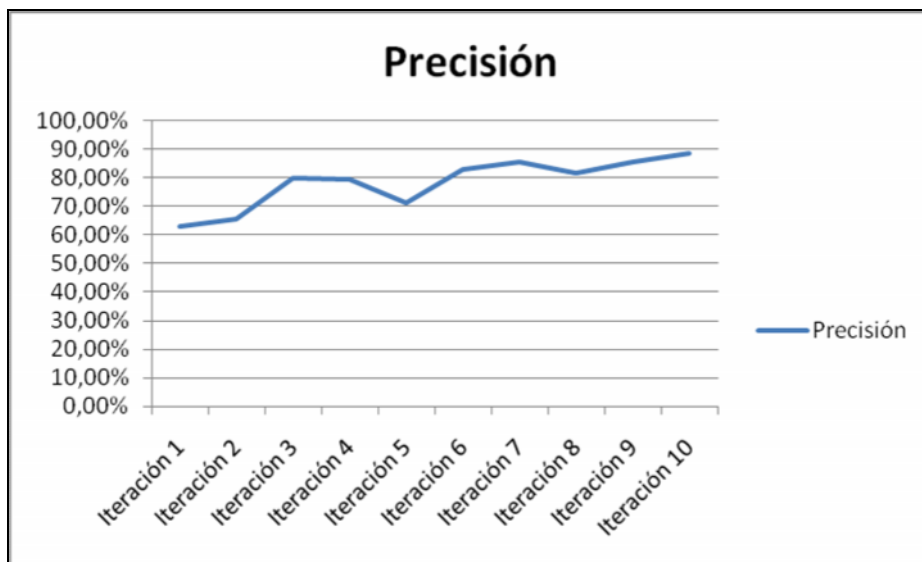


Figura 7.15. Precisión de la estimación.

La velocidad es influida directamente por la precisión en la estimación, de allí que las curvas de las Figuras 7.14 y 7.15 se asemejen y tengan una tendencia a crecer, indicando así una mejoría en ambos aspectos. Sin embargo, la velocidad se ve influida por más factores, entre ellos el largo de la iteración, a modo de ejemplo, las iteraciones 3 y 8 tuvieron una menor cantidad de horas (debido a la presencia de días festivos), lo cual se ve reflejado en una caída en la velocidad.

Cobertura de código

Las pruebas unitarias expresadas en el punto 7.1 producen una cobertura de código que se encuentra expresada en la Tabla 7.8.

Tabla 7.7. Cobertura de código del desarrollo.

Componente	Cobertura de código
Modelo Actividad	100,00%
Modelo Ot	100,00%
Modelo OtAceptada	100,00%
Modelo Permiso	100,00%
Modelo PuntoDeControl	100,00%
Controlador Actividades	100,00%
Controlador Main	100,00%
Controlador OtsAceptadas	94,39%
Controlador Permisos	100,00%
Controlador PlanificacionHerramientas	100,00%
Controlador PlanificacionMateriales	100,00%
Controlador PuntosDeControl	100,00%

La Tabla 7.8 muestra que las coberturas de código son del 100% en todas las componentes, excepto en la componente **Controlador OtsAceptadas** que alcanza sólo el 94,39%. La razón por la cual esta componente no tiene la misma cobertura que las restantes es debido a que el ambiente de pruebas **Simpletest** no puede emular la descarga de un archivo del tipo Excel, más específicamente la planificación de una orden de trabajo en este formato.

8. EVALUACIÓN

"No se vive celebrando victorias, sino superando derrotas."

Ernesto Guevara 1928-1967.

En el presente capítulo se presenta una evaluación realizada por el estudiante tesista y por la empresa patrocinante sobre el prototipo resultante y la aplicación de la metodología.

8.1. Evaluación del Prototipo

En cuanto al alcance, la estimación inicial del proyecto presentada en la Tabla 5.1 tuvo un margen de error del 30%, esto es evidente al evaluar el alcance logrado por el desarrollo (solo alcanzó a cubrir la historia de usuario B), sin embargo, la precisión en la estimación se mejoró llegando ser cercana al 90%, como es apreciado en la Figura 7.15.

En cuanto a la calidad del prototipo alcanzado, esta puede ser entendida de dos maneras: la calidad interna³⁴ y la calidad externa³⁵.

Calidad interna

La calidad interna significa que los conceptos centrales del sistema funcionan en conjunto como un todo fluido y cohesionado [Pop03]. Como fue explicado en el Capítulo 3 no es recomendable utilizar esta calidad como una variable de gestión.

La calidad interna es un pre-requisito para la calidad externa.

La forma de mantener la calidad interna siempre lo más alto posible, fue a través de pasar todas las pruebas unitarias y tener un alto porcentaje del código cubierto por estas. Si bien es cierto un alto porcentaje de cobertura de código, no implica la inexistencia de errores en el sistema, es un buen indicador de la calidad del producto.

No obstante, la experiencia ha proporcionado evidencia de una falencia en el diseño, más específicamente el llamado Modelo de Dominio Anémico (del inglés *Anemic Domain Model*) el que es considerado un anti-patrón y consiste en que la lógica de negocios no se encuentra en la capa de dominio (para el patrón MVC esto es la capa de modelos), si no en capas transaccionales (para el patrón MVC esto es la capa de controladores). Es un anti-patrón, puesto que fomenta la programación procedural, en desmedro de la programación orientada a

³⁴ La calidad interna es también conocida como la "integridad conceptual".

³⁵ La calidad externa es también conocida como la "integridad percibida".

objetos [Fow03b]. Esta falencia es evidente al comparar la cantidad de pruebas unitarias escritas en los modelos y en los controladores.

Desde el punto de vista de la empresa patrocinante, el prototipo resultante consiguió un gran nivel de calidad interna, reflejada, por ejemplo, en que la cantidad de defectos (o *bugs* por su expresión en inglés) presentados durante las entregas parciales fue nula.

Calidad externa

La calidad externa es el tipo de calidad que es percibida por el cliente. Generalmente está relacionada con la usabilidad de la GUI (interfaz de usuario por sus siglas en inglés) y la velocidad de respuesta del sistema [Bec01].

El uso extensivo de la librería ExtJS, influyó de gran manera en la interfaz de usuario, en la forma como se presentó la información y en la navegabilidad del sistema. Algunos ejemplos de la forma como fue abordado este concepto se describen a continuación:

- El menú lateral, que proporciona un acceso ordenado a cada una de las funcionalidades del sistema.
- El sistema de “pestañas”, encargadas de proporcionar la navegabilidad en el sistema.
- El calendario encargado de mostrar los distintos elementos planificados.
- El módulo de entrega de avisos.
- Las ventanas emergentes, utilizadas para definir algún componente de la planificación.
- La entrega de la planificación como un archivo Excel.

Estos aspectos buscan mejorar la experiencia vivida por el usuario al emplear el sistema, al mismo tiempo que solucionan el problema.

A juicio de la empresa patrocinante y su asesor, el prototipo cumple con dar una solución adecuada al problema en gran parte gracias a lo simple y prolijo del despliegue y uso de su información. Puede concluirse, en tanto, que la calidad externa alcanzó un muy buen nivel.

8.2. Evaluación de la Aplicación de la Metodología

Desde el punto de vista del estudiante tesista el uso disciplinado de las prácticas de desarrollo propuestas por *eXtreme Programming* y su estilo rítmico de desarrollo, implicaron una mayor confianza en la calidad del producto, mejor control del riesgo y eficiente cooperación entre

los miembros del equipo. Además, el constante aprendizaje, se tradujo en crecimiento profesional y en una mayor motivación.

Por su parte, el encargado del proyecto en la empresa patrocinante reconoció que el desafío de un prototipo funcional en diez semanas fue cumplido a cabalidad gracias al esquema de trabajo centrado en la generación de valor del estudiante tesista.

La selección precisa de las prácticas de desarrollo – así como la persistencia en su aplicación – fueron el factor decisivo en el cumplimiento del objetivo del proyecto.

9. CONCLUSIONES Y TRABAJO A FUTURO

"El hombre es bueno por naturaleza."

Jean-Jacques Rousseau 1712-1778.

Este Proyecto de Titulación muestra el desarrollo de un prototipo para la gestión de proyectos de contratistas utilizando la metodología *eXtreme Programming*.

Durante el desarrollo de este proyecto se llevaron a cabo una serie de actividades entre las cuales se cuentan:

- Se realizó un estudio de las necesidades de las empresas contratistas y de los sistemas de gestión de proyectos encontrados en el mercado. Como resultado de este estudio se seleccionaron algunas características y funcionalidades que serían implementadas en el prototipo.
- Se estudiaron los valores y principios de *eXtreme Programming* que sustentan su filosofía, la que finalmente se ve traducida a prácticas de desarrollo.
- Se estudiaron los principios SOLID y prácticas de diseño propuestas por *eXtreme Programming* que forman las bases técnicas para la aplicación la metodología y que a su vez habilitaron al diseño para evolucionar y adaptarse a los cambios en los requerimientos.
- Se describieron las historias de usuario y arquitectura preliminares al desarrollo, lo que conforma una planificación y diseño iniciales del prototipo.
- Se definió la forma en la cual fueron utilizadas durante el desarrollo del proyecto las prácticas propuestas y se describieron las métricas usadas durante el desarrollo.
- Se implementó un prototipo para la gestión de proyectos de contratistas utilizando la teoría anteriormente expuesta. A su vez, se midió el avance del proyecto utilizando las métricas descritas previamente.
- Se evaluó la calidad del producto resultante y la aplicación de la metodología.

9.1. Conclusiones

De este proceso se puede concluir que:

- La filosofía detrás de *eXtreme Programming* guía la puesta en marcha de las prácticas, priorizando de mejor manera los aspectos que resultan de mayor importancia para el

equipo de desarrollo. La comprensión de reglas simples por parte de los distintos miembros del equipo permite la distribución de responsabilidad y autoridad entre ellos.

- El uso disciplinado de las prácticas propuestas por *eXtreme Programming* disminuyen el riesgo, mantienen una alta calidad y fomentan la cooperación entre los distintos miembros del equipo.
- Estas prácticas son reforzadas por los principios SOLID que maximizan la adaptabilidad y extensibilidad del producto, al mismo tiempo que resultan en un diseño elegante.
- La medición constante de algunos aspectos del desarrollo permiten su mejora, a modo de ejemplo, la estimación al final del desarrollo tuvo una precisión superior a la del inicio del desarrollo.
- Finalmente, las personas y sus interacciones son los factores más influyentes en el éxito de un proyecto de software, sin embargo, es aún insuficiente el trabajo en esta materia y se debe mejorar la forma en la cual las metodologías de desarrollo de software abordan el tema.

9.2. Trabajo Futuro

En términos del prototipo logrado, sólo se alcanzó a completar el módulo de planificación, por lo que restan terminar otros cuatro módulos, para lo cual se deben reestimar las historias de usuario preliminares expresadas en la Tabla 5.1.

En cuanto a la aplicación de metodologías ágiles, se propone la implementación y estudio de la adaptación hecha por Tom y Mary Poppendieck [Pop03] de la metodología *Lean* para la manufactura y el desarrollo de un nuevo producto, la que es conocida como *Lean Software Development*, y constituye una ampliación a las actuales metodologías ágiles. *Lean Software Development* se compone de siete principios fundamentales y veintidós herramientas para el desarrollo de software. Entre algunas de las herramientas más destacables se pueden contar:

- Mapas de cadena de valor.
- Desarrollo basado en conjuntos.
- Uso de modelos económicos para tomar decisiones.

10.REFERENCIAS

"Educad a los niños y no será necesario castigar a los hombres"

Pitágoras 587 A.C.-507 A.C.

10.1. Libros y Publicaciones

- [And01] Ann Anderson, Ron Jeffries & Chet Hendrickson(2001). Extreme Programming Installed. Addison-Wesley Professional.
- [And04] David J. Anderson (2004). Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results. Prentice Hall.
- [Bec99] Kent Beck (1999). Extreme Programming Explained: Embrace Change (First Edition). Addison-Wesley Professional.
- [Bec01] Kent Beck (2001). Planning eXtreme Programming. Addison-Wesley Professional.
- [Bec03] Kent Beck (2003). Test Driven Development: By Example. Addison-Wesley Professional.
- [Bec05a] Kent Beck with Cynthia Andres (2005). Chapter 4:Values. Extreme Programming Explained: Embrace Change (Second Edition). Addison-Wesley Professional.
- [Bec05b] Kent Beck with Cynthia Andres (2005). Chapter 5:Principles. Extreme Programming Explained. Embrace Change (Second Edition). Addison-Wesley Professional.
- [Boe81] Barry Boehm (1981). Software Engineering Economics. Prentice Hall.
- [Boe88] Barry Boehm (1988). The Spiral Model of Software Development and Enhancement. ACM SIGSOFT Software Engineering Notes. 11(4):14-24, August 1986.
- [C3T98] C3 Team (1998). Chrysler goes to "Extremes". Distributed Computing. 24-28. October.
- [Coc99] Alistar Cockburn. Characterizing people as non linear first order components in software development. en 4th International Multi-Conference on Systems, Cybernetics and Informatics, Orlando, Florida, June, 2000.
- [Con79] Larry L. Constantine & Edward Yourdon (1979).Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall.

- [Dem99] Tom Demarco & Timothy Lister (1999). Peopleware: Productive Projects and Teams (Second Edition). Dorset House Publishing Company, Incorporated.
- [Eva04] Eric Evans (2004). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.
- [Fow99] Martin Fowler (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.
- [Fow03a] Martin Fowler (2003). Who Needs an Architect?.IEEE Software, 20 (5) :11-13. July/August.
- [Gil76] Tom Gilb (1976). Software Metrics. Studentlitteratur.
- [Ker09] Harold Kerzner Phd (2009). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.
- [Lar03] Craig Larman & Victor R. Vasili (2003). Iterative and Incremental Development: A Brief History. IEEE Computer Society, 6 (36): 47-56. June.
- [Lar04,13] Craig Larman (2004). Agile and iterative development: a manager's guide. Addison-Wesley Professional
- [Lis94] Barbara Liskov & Jeannette Wing (1994). A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, 6 (16): 1811-1841. November.
- [Mar02a] Robert C. Martin (2002). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall.
- [Mar02b] Robert C. Martin (2002). Chapter 9: SRP. Agile Software Development, Principles, Patterns, and Practices. . Prentice Hall.
- [Mar91] James Martin (1991). Rapid Application Development. Macmillan Coll Div.
- [Mar96a] Robert C. Martin (1996). The Interface Segregation Principle. Engineering Notebook, C++ Report, Nov-Dec, 1996.
- [Mar96b] Robert C. Martin (1996). The Dependency Inversion Principle. Engineering Notebook, C++ Report, May, 1996.
- [McC04] Steve McConnell (2004). "Code Complete: A Practical Handbook of Software Construction". Microsoft Press.
- [Mey88,23] Bertrand Meyer (1988). Object Oriented Software Construction. Prentice Hall.
- [Mil76] Harlan D. Mills (1976). Software Development. IEEE Transactions on Software Engineering, 4 (2): 265-273. December.

- [Pop03] Mary Poppendieck & Tom Poppendieck (2003). Lean Software Development An agile Toolkit. Addison-Wesley Professional.
- [Pro08] Project Management Institute (2008). A Guide to the Project Management Body of Knowledge. Project Management Institute.
- [PUC05] Pontificia Universidad Católica de Chile & Cámara de Comercio de Santiago (2005). El impacto de las tecnologías de la información en las empresas chilenas respecto a España y Estados Unidos: resultados de la primera encuesta BIT-Chile 2005. Business and Information Technologies (BIT) 2005 Project – Chile (Proyecto Fondecyt 1050769). Disponible en http://fcom.altavoz.net/prontus_fcom/site/artic/20080418/asocfile/20080418230431/bit_chile_2005_informe_final.pdf. Consultado el 17 de Junio del 2010.
- [Roy70] Winston Royce (1970). Managing the Development of Large Software Systems: Concepts and Techniques. Proceedings IEEE WESCON, 1-9.IEEE.
- [Sin06] Maria Siniaalto (2006). Test driven development: empirical body of evidence. Information Technology for European Advancement. Disponible en http://www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf. Consultado el 17 de Junio del 2010.
- [Sta06] The Standish Group (2002), CHAOS Report. The Standish Group.
- [Sub06] Subsecretaría de economía / División de Tecnologías de Información y Comunicación (2006). Acceso y uso de Tecnologías de Información y Comunicación en las Empresas Chilenas. Disponible en http://www.economia.cl/1540/articles-187096_recurso_1.pdf. Consultado el 17 de Junio del 2010.

10.2. Material en Internet

- [Bec09a] Kent Beck (2009). Design is beneficially relating elements. Disponible en <http://www.threeriversinstitute.org/blog/?p=111>. Consultado el 31 de mayo del 2010.
- [Bec09b] Kent Beck (2009). Coupling and Cohesion. Disponible en <http://www.threeriversinstitute.org/blog/?p=104>. Consultado el 31 de mayo del 2010.
- [Dij74] Edsger W. Dijkstra (1974). On the role of scientific thought. Disponible en <http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>. Consultado el 17 de Junio del 2010.

- [Fow02] Martin Fowler (2002). Is design dead?. Disponible en <http://martinfowler.com/articles/designDead.html>. Consultado el 8 de Noviembre de 2009.
- [Fow03b] Martin Fowler (2003). AnemicDomainModel. Disponible en <http://www.martinfowler.com/bliki/AnemicDomainModel.html>. Consultado el 27 de mayo del 2010.
- [Fow06] Martin Fowler (2006). Continuous Integration. Disponible en <http://martinfowler.com/articles/continuousIntegration.html>. Consultado el 24 de Noviembre del 2009.
- [Jef98] Ronald E Jeffries (1998). You're NOT gonna need it!. Disponible en <http://www.xprogramming.com/Practices/PracNotNeed.html>. Consultado el 3 de diciembre del 2009.
- [Ree79] Trygve Reenskaug (1979). MVC. Disponible en <http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html>. Consultado el 16 de abril del 2010.
- [Ree92] Jack Reeves (1992). What is software design?. Disponible en http://www.developerdotstar.com/mag/articles/reeves_design.html. Consultado el 10 de abril del 2010.
- [Wat07] Kelly Waters (2009). Agile Principle #4: Agile Requirements Are Barely Sufficient. Disponible en <http://www.agile-software-development.com/2007/03/agile-requirements-just-in-time-and.html>. Consultado el 19 de abril de 2009.