Ministerio de Producción

En colaboración con el

Ministerio de Educación y Deportes

Analistas del Conocimiento

Dimensión Programador

Apunte del Módulo

Desarrollo de Software

Tabla de Contenido

Definición del Módulo

Denominación de Módulo: **Desarrollo de Software**

El módulo específico de Desarrollo de Software tiene como propósito general, contribuir a la formación de los estudiantes del ámbito de la Formación Profesional en sujetos que se integrarán a equipos de desarrollo, dado que el desarrollo de software es una actividad esencialmente de trabajo en equipo.

Este módulo selecciona un conjunto de conocimientos vinculados con la disciplina de Ingeniería de Software, particularmente en los aspectos de gestión y soporte de los proyectos de desarrollo de software.

Este módulo se orienta al desarrollo de las siguientes capacidades profesionales, estando estas articuladas con las funciones que se describen en el alcance del perfil profesional:

- Integrar un equipo en el contexto de un Proyecto de Desarrollo de Software.

El desarrollo de software es una actividad social, que se desarrolla principalmente en equipo, en consecuencia, el Programador debe poder integrarse en un equipo de trabajo, sea este un contexto de proyecto de gestión tradicional o de gestión ágil.
Debe poder manejar su entorno personal y el entorno laboral donde se insertará.

- Dimensionar su trabajo en el contexto del proyecto de desarrollo de software.

El Programador como parte integrante de un equipo de proyecto debe poder estimar el esfuerzo que necesita para realizar un trabajo que le fue asignado. Para ello deberá procurarse la información que necesite para dimensionar el trabajo, considerando la utilización de recursos de los que disponga para ser productivo, por ejemplo, utilización de bibliotecas de componentes, aplicación de patrones, entre otros.

Para el presente módulo, y desde el punto de vista del contenido de la formación, se define para el agrupamiento, la selección y el tratamiento de los contenidos, los siguientes bloques:

- Bloque Disciplinas implicadas en el Desarrollo de Software
- Bloque de Gestión de Proyectos

En el bloque Disciplinas implicadas en el Desarrollo de Software se focaliza en introducir los conceptos fundamentales que conforman la problemática del desarrollo de software. A partir de ellos se abordarán conceptos sobre la ingeniería de software y sus disciplinas constitutivas. Se profundizará sobre actividades vinculadas con la calidad del software tanto en etapas de construcción, como son las revisiones técnicas, como para control de calidad, como es el testing.

El bloque Gestión de Proyectos abordará conceptos clave para el logro exitoso de un software, vinculados a la planificación y el monitoreo de los proyectos que los desarrollan. Se presentarán los dos enfoques principales que conviven en la industria: la gestión tradicional y la gestión ágil. Se desarrollarán los conceptos clave que sustentan cada enfoque.

En relación a las prácticas formativas de carácter profesionalizante, se definen como unos de los ejes estratégicos de la propuesta pedagógica para el ámbito de la FP, el situar al participante en los ámbitos reales de trabajo con las problemáticas características de desempeño ocupacional/profesional.

Las prácticas formativas que se proponen para este módulo se organizan en torno a la integración de los participantes en un proyecto de desarrollo de software y que puedan desarrollar actividades vinculadas con la construcción de un producto de software, con algún rol específico, asignado. La expectativa sobre este tipo de prácticas es lograr que la vivencia los aproxime a la realidad de la problemática de la industria de software.

Los objetivos de aprendizaje a tener en cuenta para la evaluación al finalizar el cursado del módulo de "Desarrollo de Software" serán:

- Identificar las disciplinas que conforman la Ingeniería de Software y las técnicas y herramientas relacionadas.
- Conocer los tipos de procesos y los modelos de procesos más adecuados para el desarrollo de software en cada situación particular.
- Introducir los enfoques de gestión de proyectos tradicional y ágil.
- Conocer los principales métodos de desarrollo y gestión ágil.
- Valorar la relación existente entre el Proceso, el Proyecto y el Producto de Software a construir
- Reconocer la importancia de la Gestión de Configuración de Software.
- Conocer técnicas y herramientas para realizar pruebas y revisiones técnicas al software.
- Integrar por medio de casos prácticos concretos los conocimientos adquiridos en la parte teórica, empleando así las técnicas y herramientas de aplicación de la ingeniería de software.

Los contenidos del módulo se resumen seguidamente:

Bloque Disciplinas implicadas en el Desarrollo de Software

- Proceso de Desarrollo definidos y empíricos.
- Procesos de Desarrollo y Modelos de Proceso. Criterios para elección de ciclos de vida en función de las necesidades del proyecto y las características del producto.
- Disciplinas que conforman la Ingeniería de Software: disciplinas técnicas, disciplinas de gestión y de soporte.
- Gestión de Configuración de Software
- Testing de Software.
- Revisiones técnicas al software.

Bloque: Gestión de Proyectos

- Gestión de Proyectos de Desarrollo: tradicional y ágil.
- Introducción a los Métodos Ágiles.
- Manifiesto Ágil.
- Requerimientos en ambientes ágiles - User Stories.
- Métodos Ágiles: SCRUM, XP y TDD

Introducción

El software en su origen era la parte insignificante del hardware, lo que venía como añadidura, casi como regalo, no pasó mucho tiempo de su evolución y se generó una idea de software: igual programa; y en breve tiempo adquirió una entidad propia, la cual desde el comienzo fue acompañada de complejidad (dificultad de…), por ser un producto difícil de entender, difícil de explicar, difícil de construir y difícil de mantener, es decir, un producto difícil, un producto intangible, maleable y con diversas representaciones, en constante cambio y evolución. El software se ha convertido en ubicuo, está inserto en la mayoría de las actividades de la persona humana, en su cotidianeidad. Por lo anterior, es importante no subestimarlo.

En el diccionario bilingüe Longman, se define el término software como el conjunto de programas que controlan la operación de una computadora. Pressman, define que el software se forma con:

1) las instrucciones (programas de computadoras) que al ejecutarse proporcionan las características, funciones y el grado de desempeño.

2) las estructuras de datos que permiten que los programas manipulen la información de manera adecuada.

3) los documentos que describen la operación y el uso de los programas.

Por su parte, Freeman caracteriza al software como "el alma y cerebro de la computadora, la corporización de las funciones de un sistema, el conocimiento capturado acerca de un área de aplicación, la colección de los programas, y los datos necesarios para convertir a una computadora en una máquina de propósito especial diseñada para una aplicación particular, y toda la información producida durante el desarrollo de un producto de software".

El software no sólo hace referencia a los programas ejecutables sino también representa diversas cosas, las cuales difieren en el tiempo, con las personas y principalmente en cómo se lo va a emplear. El software viabiliza el producto más importante de nuestro tiempo: *la información*.

Características del software.

El software es un elemento lógico, no físico, para comprender mejor al software es importante conocer las características que lo distinguen de otros productos que construye el hombre.

Sumado a lo antes dicho, recordemos que "hacemos software con software". Esto no sólo dificulta el proceso de diseño en nuestra mente, sino que dificulta severamente la comunicación entre las mentes.

Robert Cochram, utiliza las siguientes características para describir lo que es único y especial en el software:

1. El software es intangible.
2. Tiene alto contenido intelectual.
3. Generalmente, no es reconocido como un activo por los Contadores, por lo que no está en los balances.
4. Su proceso de desarrollo es humano intensivo, basado en equipos y construido en proyectos.
5. El software no exhibe una separación profunda entre I&D [1] y producción.

[1] I&D: Investigación y Desarrollo

6. El software puede ser potencialmente modificado, en forma permanente.

Pressman, expresa las características del software haciendo énfasis en la diferencia que éste tiene con el hardware:

1. El software no se manufactura, se desarrolla.
2. El software no se desgasta.
3. A pesar de la tendencia de la industria a desarrollar por componentes, gran parte del software aún se construye a medida.

Pese a la diversidad de opiniones que expresan los autores respecto a las características del software, todos coinciden en su complejidad. Según Basili, el software es intangible y su intangibilidad es lo que contribuye fuertemente a su complejidad. El software es complejo, complejo de construir, complejo de entender, complejo de explicárselo a otros. Esta característica inherente del software, junto al hecho de que el software es, desarrollado y no construido y la falta de principios y componentes bien definidos hace de él un producto diferente a cualquier otro, con el que se haya tratado antes.

La existencia de pedidos de cambio y evolución constante de su función y/o estructura, su desarrollo propenso a errores, la dificultad para su estimación, la falta de comprensión de las implicancias que trae aparejado el cambio, son algunas de las razones que fundamentan su complejidad.

Sistema Informático y la Ingeniería para su desarrollo

Un sistema informático está compuesto por hardware y software. En cuanto al hardware, su producción se realiza sistemáticamente y la base de conocimiento para el desarrollo de dicha actividad está claramente definida. La fiabilidad del hardware es, en principio, equiparable a la de cualquier otra máquina construida por el hombre. Sin embargo, respecto del software, su construcción y resultados han sido históricamente cuestionados debido a los problemas asociados, entre ellos podemos destacar los siguientes:

- Los sistemas no responden a las expectativas de los usuarios.
- Los sistemas “fallan” con cierta frecuencia.
- Los costos del software son difíciles de prever y normalmente superan las estimaciones.
- La modificación del software es una tarea difícil y costosa.
- El software se suele presentar fuera del plazo establecido y con menos prestaciones de las consideradas inicialmente.
- Normalmente, es difícil cambiar de entorno de hardware usando el mismo software.
- El aprovechamiento óptimo de los recursos no suele cumplirse.

El primer reconocimiento público de la existencia de problemas en la producción de software tuvo lugar en la conferencia organizada en 1968 por la Comisión de Ciencias de la OTAN en Garmisch (Alemania), dicha situación problemática se denominó *crisis del software*. En esta conferencia, así como en la siguiente realizada en Roma en 1969, se estipuló el interés hacia los aspectos técnicos y administrativos del desarrollo y mantenimiento de productos software.

Se pretendía acordar las bases para una ingeniería de construcción de software. Según Fritz Bauer, lo que se necesitaba era “*establecer y usar principios de ingeniería orientados a obtener software de manera económica, que sea fiable y funcione eficientemente sobre máquinas reales*”. Esta definición marcaba posibles cuestiones tales como: ¿Cuáles son los principios robustos de la

ingeniería aplicables al desarrollo de software? ¿Cómo construimos el software económicamente para que sea fiable? ¿Qué se necesita para crear software que funcione eficientemente?

Ingeniería de Software

El "IEEE Standard Glossary of Software Engineering Terminology" ha desarrollado una definición más completa para ingeniería del software: "(1) La aplicación de un enfoque sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software.

El objetivo principal que busca la ingeniería de software es convertir el desarrollo de software en un proceso formal, con resultados predecibles, que permitan obtener un producto final de alta calidad; que satisfaga las necesidades y expectativas del cliente. Generalmente a partir de un complejo esquema de comunicación en el que interactúan usuarios y desarrolladores, el usuario brinda una concepción de la funcionalidad esperada y el desarrollador especifica esta funcionalidad a partir de esta primera concepción mediante aproximaciones sucesivas. Este ambiente de interacción motiva la búsqueda de estrategias robustas para garantizar que los requisitos del usuario serán descubiertos con precisión y que además serán expresados en una forma correcta y sin ambigüedad, que sea verificable, trazable y modificable.

El estudio de enfoques en Pressman, caracteriza la Ingeniería de Software como "una tecnología multicapa", ilustrada en la Figura 1.

Figura 1: Capas de la Ingeniería de Software.

Dichas capas se describen a continuación:

- Cualquier disciplina de ingeniería (incluida la ingeniería del software) debe descansar sobre un esfuerzo de organización de **calidad**. La gestión de calidad y las filosofías similares fomentan una cultura continua de mejoras de procesos que conduce al desarrollo de enfoques cada vez más robustos para la ingeniería del software.
- El fundamento de la ingeniería de software es la **capa proceso**. El proceso define un marco de trabajo para un conjunto de áreas clave, las cuales forman la base del control de gestión de proyectos de software y establecen el contexto en el cual: se aplican los métodos técnicos, se producen resultados de trabajo, se establecen hitos, se asegura la calidad y el cambio se gestiona adecuadamente.
- Los **métodos** de la ingeniería de software indican cómo construir técnicamente el software. Los métodos abarcan una gran gama de tareas que incluyen análisis de requisitos, diseño, construcción de programas, pruebas y mantenimiento. Estos métodos

dependen de un conjunto de principios básicos que gobiernan cada área de la tecnología e incluyen actividades de modelado y otras técnicas descriptivas.

- Las **herramientas** de la ingeniería de software proporcionan un soporte automático o semi-automático para el proceso y los métodos, a estas herramientas se les llama herramientas CASE (*Computer-Aided Software Engineering*), lo que significa Ingeniería de Software Asistida por computadoras.

Dado lo anterior, el objetivo de la ingeniería de software es lograr productos de software de calidad (tanto en su forma final como durante su elaboración), mediante un proceso apoyado por métodos y herramientas.

Disciplinas de la Ingeniería de Software

Considerando que el propósito de la Ingeniería de Software es obtener productos de software de calidad, se apoya en un grupo de campos de estudio, llamados comúnmente disciplinas, que aportan un desarrollo y estudio organizado de saberes. Dada la diversidad de campos de estudio que abarca la Ingeniería de Software, se las puede clasificar según su propósito en disciplinas Técnicas, disciplinas de Gestión y disciplinas de Soporte.
Las disciplinas denominadas **Técnicas** son las relacionadas directamente con la **construcción del producto de software**. Las disciplinas de **Gestión** están relacionadas con el proyecto, que tal como se profundizará más adelante en este apunte, es el medio que se utiliza para guiar la construcción de software. Las disciplinas de **Soporte,** también conocidas como *disciplinas protectoras*, son las que brindan herramientas que ayudan a obtener calidad para el software que se va a desarrollar:

Figura 1: Principales Disciplinas de la Ingeniería de Software.

El desarrollo de software en una mirada

El desarrollo de software se trata de desarrollar y entregar gran software. Un software es un gran software cuando complace a quienes lo necesitan, a quienes lo pidieron, a quienes lo usarán.

Un gran desarrollo de software entrega lo que se necesita, a tiempo y dentro del presupuesto. Cada pieza de software comienza con una idea de un cliente. Es tu trabajo como desarrollador de software es ***dar vida a esa idea***. Sin embargo, tomar una idea vaga y transformarla en software funcionando, software que satisfaga a su cliente, no es una tarea trivial. El propósito de este apunte es poder introducirlos en el mundo del desarrollo de software, y llegar a entregar el software que se necesita, a tiempo y dentro del presupuesto definido.

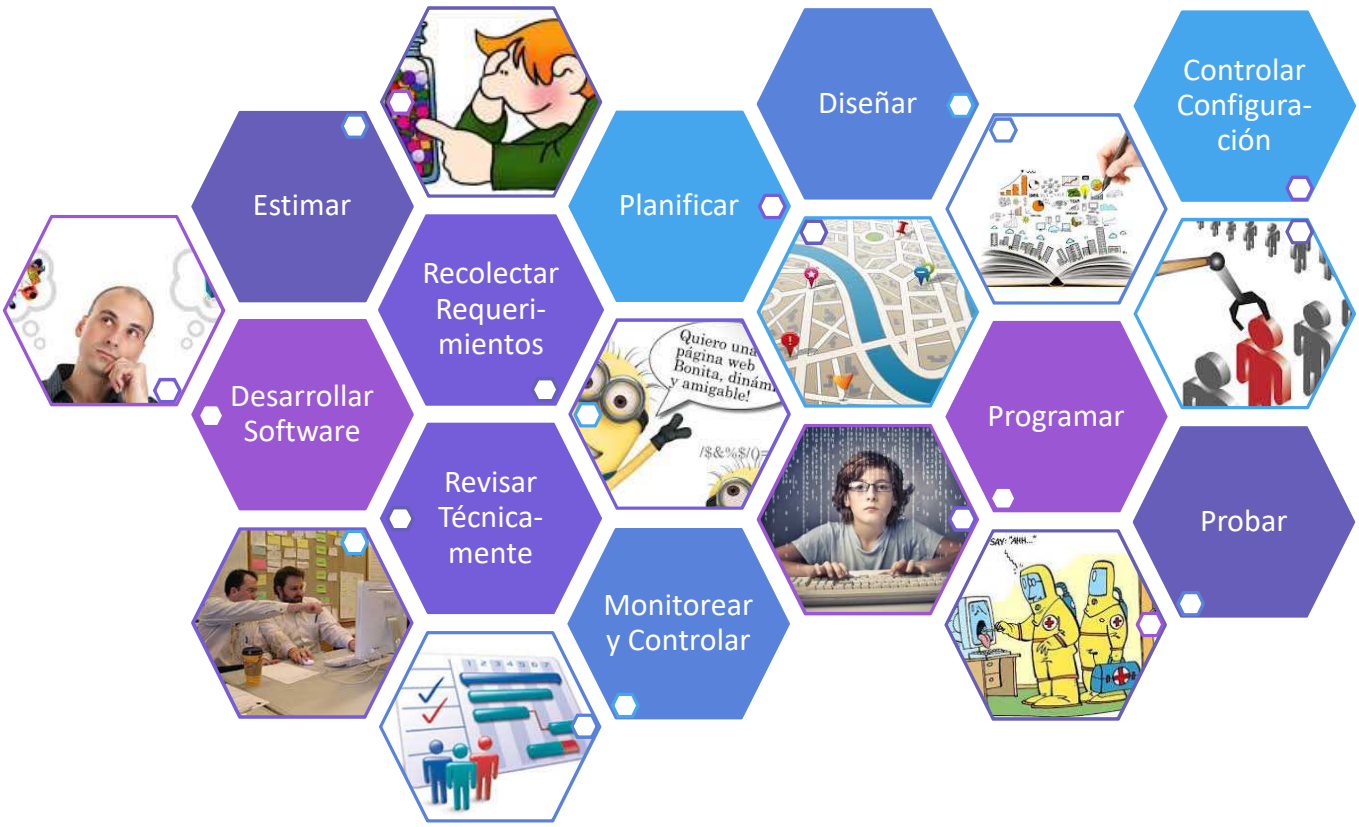


Figura 2: Una mirada al Desarrollo de Software.

Desarrollar Software ⇨ Dejar contento al Cliente

El desarrollo de Software es una actividad cuyo propósito es transformar ideas, necesidades y recursos en un producto de software.

El software se desarrolla por personas, que deben estar motivadas y capacitadas para hacer su trabajo. Debe emplear un proceso que guía la transformación; y ayudarse con herramientas que faciliten el trabajo.

A continuación, vamos a hacer un recorrido por las actividades principales, involucradas en el Desarrollo de Software.

Recolectar Requerimientos ⇨ Saber que quiere el Cliente

En el ámbito del desarrollo de software se define un requerimiento cómo *una característica que el producto que se va a construir debe satisfacer*.

Esa característica puede está relacionada con algún comportamiento o con alguna capacidad, que el Cliente/Usuario espera encontrar en el producto de Software.

De todas las actividades involucradas en el desarrollo de software, **la más difícil de todas es precisamente esta: acordar qué software** es el **que se quiere** construir.

Planificar ⇨ Definir la hoja de ruta para el viaje

Un plan es a un proyecto lo que una hoja de ruta a un viaje.

Planificar *es definir qué es lo que haremos, cuándo lo haremos, cómo vamos a hacerlo y quién lo hará*.

La planificación establece compromisos que esperamos cumplir, y si bien con frecuencia los planes pueden están equivocados, si no se planifica, no se puede responder las preguntas básicas antes mencionadas.

Si no sabemos planificar, lo mejor que podemos hacer es planificar con más frecuencia.

Estimar ⇨ ¿Cuán grande será? ¿Cuánto costará? ¿Cuándo estará terminado?

Estimar *es predecir el tiempo y el costo que llevará desarrollar un producto de software, basándonos es el tamaño de lo que queremos construir.*

Las estimaciones tienen asociada una probabilidad, y esa probabilidad está relacionada con la información con la que contamos al momento de predecir.

El problema con las estimaciones es que se las requiere muy pronto, y es difícil hacer predicciones, especialmente sobre el futuro.

Diseñar ⇨ Crear el Software

Diseñar *es crear, diseñar es tomar decisiones respecto de cuál es la mejor forma de satisfacer los requerimientos definidos para el producto*.

Durante el diseño se aplican varias técnicas y principios, para crear modelos que especificarán el producto a construir. Al diseñar software se crean representaciones que visualizan diferentes partes del mismo, a saber: su arquitectura, sus procesos, sus bases de datos y la forma en la que el software interactuará con sus usuarios.

La profundización de los conceptos, principios, técnicas y herramientas relacionadas con el Diseño de Software están fuera del alcance de este material.

Programar ⇨ Construir el Software

Programar *es escribir código, que permitirá controlar lo que una computadora hará*.

Se transforman diseños utilizando uno o varios lenguajes de programación, obteniendo un producto que *implementa* los requerimientos acordados.

El resultado de este proceso es lo que se instala en las computadoras de los usuarios.

Los conceptos, principios, técnicas y herramientas relacionadas con la Programación en general, y con el paradigma Orientado a Objetos en particular, son abordados en el **Módulo de Programación Orientada a Objetos.**

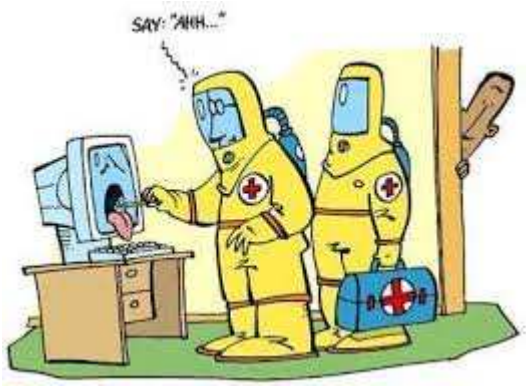
Revisar Técnicamente ⇨ Verificar con un colega lo que hicimos

Las *revisiones técnicas del software tienen por objetivo detectar tempranamente errores que se comenten al desarrollar*.

Dado que el software es invisible, y lo desarrollamos en nuestra mente, es esperable que comentamos errores.

Si mostramos y explicamos a otros el trabajo que hicimos es muy probable que esos errores sean identificados y puedan corregirse antes que se hagan mayores, impacten en otros artefactos; y que sea más costoso corregirlos.

Probar ⇨ ¿Construimos el producto adecuado? ¿Funciona correctamente?



La prueba del software, en inglés *testing*, *es un proceso que tiene por objetivo encontrar defectos, que se asume de ante mano que está allí.*

Durante la prueba debemos poder validar que el producto que se está probando es el que el usuario quería, y verificar que el producto funciona correctamente.

Controlar la Configuración ⇨ Cuidar y defender lo que desarrollamos

Una de las características del software es que es fácil de modificar, maleable y dado que vamos a necesitar cambiarlo muchas veces, es necesario crear mecanismos que nos ayuden a controlar el software a lo largo de su ciclo de vida.

El control de la configuración del software *protege la integridad del producto que construimos de pérdidas de componentes o de cambios realizados a los componentes, evita superposición de cambios y esfuerzos duplicados de mantenimiento*.

Monitorear ⇨ ¿Cómo vamos en el viaje? ¿Podemos decirlo con números?

El monitoreo y control compara los planes realizados con el avance real de un proyecto. Nos da visibilidad respecto de la situación del proyecto en un momento de tiempo, cuánto hicimos, cuánto nos falta para terminar, cuánto gastamos, cuanto tiempo consumimos.

Si a esas preguntar podemos responderlas con números será más fácil interpretar el estado real del proyecto y poder tomar decisiones acertadas respecto de cómo continuar.

La profundización de los conceptos, principios, técnicas y herramientas relacionadas con el Monitoreo de Proyectos están fuera del alcance de este material.

Finalizado el recorrido general que muestra lo que implica desarrollar software, durante el resto del apunte ahondaremos respecto de algunas de estas actividades, presentando un conjunto de herramientas que serán de utilidad.

Desarrollar Software

El Software como producto se desarrolla de manera gradual, cada versión del software es única y se obtiene como resultado de la ejecución de un Proyecto. Cada proyecto involucra personas que son responsables de la transformación de los requerimientos en el producto final. El proyecto utiliza un proceso que define las actividades que se llevarán a cabo para alcanzar los objetivos.

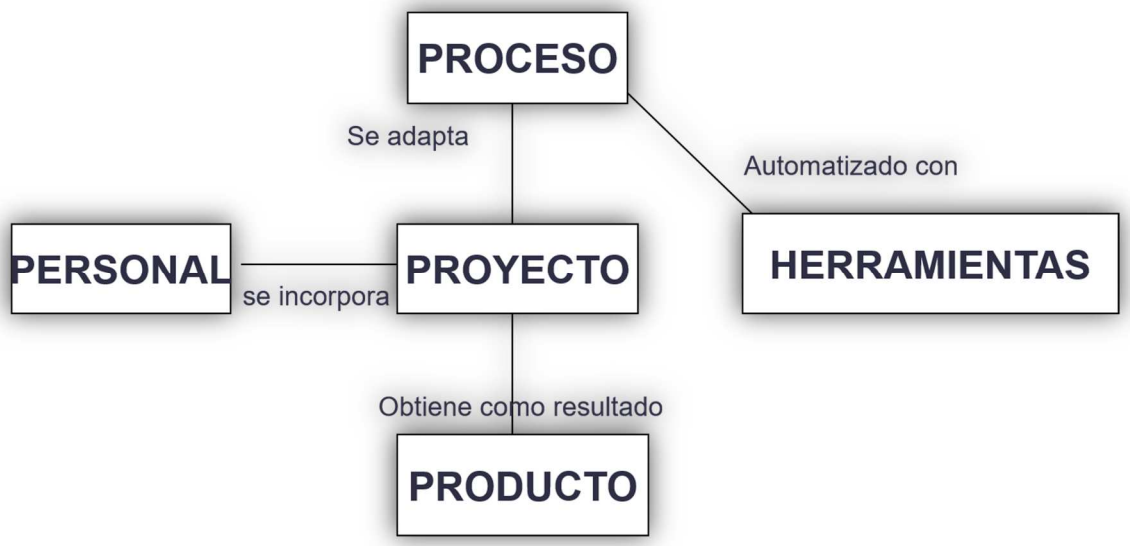


Figura 4: Las 4 "P" en el desarrollo de Software

Personas: ¿Quién hace que?

El factor humano es fundamental para el éxito del proyecto.
Las personas son los principales autores de un proyecto de Software, que asumen diferentes roles tales como analistas de sistemas, diseñadores, programadores, arquitectos, analistas de prueba, líderes de proyecto, revisores técnicos, entre otros. Estas personas conformarán lo que se denomina Equipo de Desarrollo.

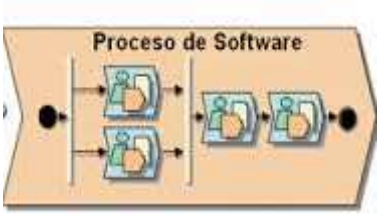
Algunos de los factores que deben considerarse al momento de la conformación de un Equipo de Desarrollo son:

- Experiencia en el dominio de la aplicación.
- Experiencia en la plataforma y el lenguaje de programación.
- Habilidad para resolver problemas.
- Habilidad de comunicación.
- Adaptabilidad.
- Actitud.
- Personalidad

Un equipo de desarrollo para ser eficaz deberá maximizar capacidades y habilidades de cada persona, teniendo en cuenta las características de la organización, la cantidad de personas que

integran el equipo, el grado de habilidad de los integrantes y la dificultad del problema a resolver.
El Equipo de Desarrollo debe interactuar con otros participantes que también están involucrados en la obtención del software, que es importante mencionar: Gerentes de Proyecto, Gerentes de Producto, Clientes, Usuarios Finales.

Proceso: cómo se hacen las cosas



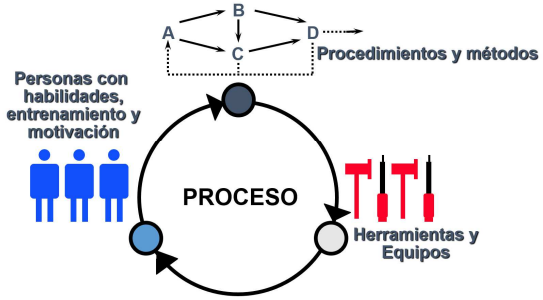


Figura 5: Elementas que conforman un proceso de desarrollo de Software

Un proceso de *software define un conjunto completo de actividades necesarias para transformar los requerimientos de un usuario en un producto*.

El proceso proporciona un marco de trabajo, una estructura que puede tomarse como referencia para definir el plan que guiará el proyecto para el desarrollo del software.

Un proceso de desarrollo de software contiene además de la definición de las actividades, procedimientos y métodos, la identificación de las herramientas que permitan la automatización de las actividades y facilite el trabajo.

Finalmente, las personas, que en el desarrollo de software, son lo más importante, ya que la materia prima para su creación es la mente y el trabajo de las personas que integrarán el equipo.

La definición completa del proceso contiene actividades relacionadas con las **disciplinas técnicas**, **de gestión** y **de soporte o protectoras**, mencionadas con anterioridad en este apunte.
El proceso define además de las actividades que deben realizarse, las entradas que necesita, las salidas que produce, los roles responsables de la realización de las actividades, la definición de las herramientas que se necesitan y la forma de medir y controlar si las actividades efectivamente se realizaron.
Sin embargo, un pequeño número de actividades es aplicable a todos los proyectos de software, sin importar su tamaño o complejidad. Otros conjuntos de tareas permiten que las actividades del proceso se adapten a las características del proyecto de software, así como a los requisitos del equipo del proyecto y a las características del Producto.
Vinculado al concepto de proceso está el concepto de **Modelo de Proceso o Ciclo de Vida del Proceso de Desarrollo**, el cual se define como una representación de un proceso. El modelo de proceso grafica una descripción del proceso desde una perspectiva particular, indicando fases para el proceso y el orden en el que las mismas se llevarán a cabo.
Hay diferentes tipos de modelos de procesos. Algunos, denominados **secuenciales,** plantean que las fases del proceso deben ejecutarse en forma completa y en orden, avanzando con el cien por ciento de los artefactos que deben construirse en cada fase antes de poder pasar a la siguiente.
Otros modelos de proceso denominados **iterativos**, por el contrario, plantean que puede avanzarse con una porción del producto que debe construirse, evolucionando a través de cada una de las etapas hasta obtener una versión completa, para una parte del producto y luego

repetir el proceso hasta terminar el desarrollo completo. Se abordará este tema con más detalle en la última sección del apunte.
El mismo proceso puede ser utilizado con diferentes modelos de proceso en cada proyecto que se realizará, según las características particulares de cada proyecto.

Proyecto: una ejecución única

Es un esfuerzo temporal que se lleva a cabo para crear un producto, servicio o resultado único, en este caso un producto de software.

El proyecto integra personas, utiliza un proceso y herramientas para obtener como resultado un producto de software.

Al inicio del proyecto debe decidirse que conjunto de actividades definidas en el proceso, aplican para el proyecto, es decir **adaptar el proceso al proyecto**. Así como también el **modelo de proceso** o ciclo de vida más conveniente para utilizar en el proyecto.

Un proyecto tiene las siguientes características:

- **Temporal**: esto significa que tiene una fecha de comienzo y fin definidos. Un proyecto termina cuando se alcanza el/los objetivo/s o cuando queda claro que no podrán alcanzarse por la razón que sea. Una línea de producción que obtiene productos en masa, no es un proyecto.

- **Productos, Servicios o Resultados únicos**: Los resultados que se obtienen de los proyectos por similares que sean tienen características que los hacen únicos. La presencia de elementos repetitivos no cambia la condición fundamental de obtener resultados "únicos".

- **Orientado a Objetivos**: Los proyectos están dirigidos a obtener resultados y ello se refleja a través de objetivos. Los objetivos guían al proyecto. Los objetivos que se definen deben ser **claros** y **alcanzables**.
  Un **objetivo es claro** cuando todas las personas que están involucradas en el proyecto comprenden lo que hay que lograr.
  Un objetivo **es alcanzable** cuando lo definido como expectativa a alcanzar es factible de hacerse.

- **Elaboración Gradual**: Desarrollo en pasos que aumenta mediante incrementos. Debe coordinarse cuidadosamente con el alcance del proyecto.
  El **alcance del proyecto** se define como todo el trabajo y solo el trabajo que debe realizarse para cumplir con el objetivo definido. La elaboración gradual de las actividades del proyecto debe mantenerse alineada con el alcance definido para el mismo.

Producto: el resultado

Una **versión de un Producto de Software** puede incluir entre otras cosas: código, bases de datos, manuales de usuario, manuales de configuración, modelos del diseño, documentos de arquitectura, documentos de requerimientos, casos de prueba.

El producto de software *es el conjunto de artefactos o componentes que se obtienen como salida de cada una de las actividades definidas en un proceso, que se ejecutan en un proyecto*.

El producto de software se obtiene como consecuencia de una evolución y refinamiento continuo de los modelos, que parten desde los requerimientos del usuario y/o cliente.

El Producto es más que código, en el contexto del desarrollo de software, el producto que se obtiene es un **sistema software**. El termino producto aquí hace referencia al sistema entero, y no sólo al código ejecutable que se entrega al cliente.

Un sistema software es la sumatoria de todos los artefactos que se necesitan para representarlo en una forma comprensible para las máquinas, los trabajadores y los interesados.

El término **Artefacto,** *se utiliza para referenciar cualquier información creada, producida, cambiada o utilizada por las personas en el desarrollo del sistema*.

Básicamente, hay dos tipos de artefactos: artefactos de ingeniería y artefactos de gestión. Los de ingeniería son creados durante la ejecución de las disciplinas técnicas (Requerimientos, Análisis, Diseño, Implementación, Prueba y Despliegue).

Los artefactos de gestión tienen un tiempo de vida corto, lo que dura la vida del proyecto; A este conjunto pertenecen artefactos como el plan de proyecto.

Tener un Proceso en la Vida…Tener un Proceso en el Software

El software se desarrolla para algún cliente en particular o para un mercado en general. Para el diseño y desarrollo de productos de software se aplican metodologías, modelos y técnicas que permiten resolver los problemas. En los años 50 no existían metodologías de desarrollo, el desarrollo estaba a cargo de los propios programadores.

Los resultados eran impredecibles, no se sabía la fecha exacta en que concluiría un proyecto de software, no había forma de controlar las actividades que se estaban desarrollando. Tampoco se contaba con documentación estandarizada. El nacimiento de técnicas estructuradas es lo que da origen al desarrollo de aplicaciones aplicando técnicas, métodos y herramientas de ingeniería.

La informática aporta herramientas y procedimientos que se apoyan en la ingeniería de software con el fin de mejorar la calidad de los productos de software, aumentar la productividad y trabajo de los desarrolladores de software, facilitar el control del proceso de desarrollo de software y suministrar a los desarrolladores las bases para construir software de alta calidad en una forma eficiente. En definitiva, aporta lineamientos para conformar un "proceso" para estructurar el desarrollo del software.

Desde 1985 hasta el presente, aparecen constantemente, herramientas, metodologías y tecnologías que se presentaban como la solución definitiva al problema de la planificación, previsión de costos y aseguramiento de la calidad en el desarrollo de software. La dificultad propia de los nuevos sistemas, y su impacto en las organizaciones, ponen de manifiesto las ventajas, y en muchos casos la necesidad, de aplicar una metodología formal para llevar a cabo los proyectos de este tipo.

Una parte importante de la ingeniería de software es el desarrollo de metodologías y modelos. Muchos esfuerzos que se encaminan al estudio de los métodos y técnicas para lograr una aplicación más eficiente de las metodologías y lograr sistemas más eficientes y de mayor calidad con la documentación necesaria en perfecto orden y en el tiempo requerido.

Una metodología define *una forma disciplinada para desarrollar software con el objetivo de hacerlo más predecible y eficiente*. Una metodología contiene representaciones que facilitan la manipulación de modelos, y la comunicación e intercambio de información entre todas las personas involucradas en la construcción de un sistema.

La experiencia ha demostrado que los proyectos exitosos son aquellos que son administrados siguiendo una serie de procesos que permiten organizar y luego controlar el proyecto. Es necesario destacar la importancia de los métodos, pero el éxito del proyecto depende más de la comunicación efectiva con los interesados, el manejo de las expectativas y el nivel de involucramiento y motivación de las personas que participan en el proyecto.

Existen diferentes metodologías que han sido en los últimos años herramientas de apoyo para el desarrollo del software. Sommerville (2005), menciona que:

• Metodología de desarrollo de software: es un enfoque estructurado para el desarrollo de software que incluye modelos de sistemas, notaciones, reglas, sugerencias de diseño y guías de procesos.

En este sentido, Jacobson (2000), aporta el concepto de Proceso de Desarrollo, con la intención que sirva para hacer operativo y concreto, lo expuesto en una metodología, definiéndolo como:

- Proceso de Desarrollo de Software *define quién está haciendo qué, cuándo y cómo alcanzar un determinado objetivo.* En la ingeniería de software, el objetivo es construir un producto de software o mejorar uno existente. Es necesario que el proceso sirva de guía a los participantes (clientes, usuarios, desarrolladores, diseñadores, líderes de proyectos, por mencionar algunos ejemplos).

Procesos de Control

Cuando trabajamos en el desarrollo de software, utilizando proyectos, podemos elegir dos enfoques muy diferentes de procesos de control. Esta elección determinará la forma en la que se realizará el trabajo en el contexto del proyecto y las decisiones que se tomen, por consecuencia afectará la forma en la que se obtendrán y presentarán los productos resultantes.

Proceso Predictivo de Control

El proceso predictivo de control asume que es posible detallar las principales variables de un proyecto (requisitos, tareas, recursos, costos y tiempos) y predecir su comportamiento a largo plazo.
Adicionalmente a esta predicción, se crea una cadena de actividades donde cada una recibe una entrada, ejecuta una serie de tareas preestablecidas y produce una salida esperada. Cada una de estas tareas está asociada a un determinado rol que posee ciertas habilidades para transformar la entrada en una salida. Otro supuesto fundamental de este enfoque de control, es que la responsabilidad sobre un ítem en progreso se traspasa entre los roles que se ocupan de las diferentes actividades, quedando en el rol destinatario del traspaso la responsabilidad de la aceptación del ítem.
Dado que cada actividad requiere diferente cantidad de esfuerzo, el control se realiza midiendo el esfuerzo incurrido contra el estimado.
El supuesto de que el flujo de trabajo está prediseñado y predefinido nos lleva a la conclusión de que la predictibilidad se puede asegurar en cualquier momento durante el proyecto.

Proceso Empírico de Control

A diferencia del proceso Predictivo de Control, el proceso Empírico acepta la complejidad y en vez de pretender controlar la realidad, su propósito es medir constantemente los resultados y adaptar el comportamiento en consecuencia. Bajo este paradigma, las variables del proyecto se pueden predecir, pero con un horizonte temporal acotado.

Manteniendo entonces un intervalo de tiempo pequeño y constante, el control puede ser bastante granular al revisar sistemáticamente el resultado del trabajo realizado durante ese intervalo de tiempo y ajustando las variables del contexto y permitiendo así una constante estabilización y optimización del sistema.

El rol del equipo de trabajo es fundamental en un proceso empírico ya que los individuos dejan de ser percibidos como agentes independientes y pasan a ser parte de un conjunto interdependiente de personas que forman un equipo, donde el éxito o el fracaso es colectivo y no individual.

Creer que los procesos predictivos, por más detallados y definidos que estén (predictivos que sean), son más confiables y seguros que las personas que los ejecutan, es desde la perspectiva de gran parte de la industria de software, una falacia. Los procesos predictivos no aprenden. Los procesos predictivos no piensan. Las personas son las que piensan, aprenden y crean cosas nuevas.

Los procesos predictivos son extremadamente valiosos y eficientes para ejecutar tareas repetitivas en contextos predecibles. ¿Alguna vez te sentaste frente a una hoja por un rato para "ser creativo"? ¿Alguna vez trabajaste en un proceso predictivo con un paso donde había que crear algo nuevo? ¿Cómo te ha ido? Los procesos predictivos son racionales, lógicos. Los procesos empíricos dan lugar a la creatividad. Los procesos predictivos se basan en experiencias pasadas e intentan repetirlas, pero debemos vivir mirando hacia el futuro, hacia la innovación.

Del paradigma de los procesos empíricos de control, surgen las metodologías ágiles para el desarrollo de software.

Al final de este apunte se describen los fundamentos y principios de las metodologías ágiles; el manifiesto ágil y se introducen algunas de las metodologías ágiles más conocidas.

Modelos de proceso para el desarrollo de software

Tal como se mencionó antes, un modelo de proceso o ciclo de vida, *es una representación abstracta de un proceso.* Cada modelo representa un proceso desde una perspectiva particular y así proporciona información parcial sobre el proceso. Éstos modelos generales no son descripciones definitivas de los procesos del software, más bien son abstracciones de los procesos que se pueden utilizar para el desarrollo del software. Puede pensarse en ellos como marcos de trabajo del proceso, que definen detalles de cómo trabajar con los procesos. Un proceso en particular puede utilizarse con distintos modelos de proceso, en proyectos diferentes, en función de las situaciones particulares de esos proyectos.

A continuación, se mencionan algunos modelos de proceso utilizados en los proyectos de desarrollo de software, existen muchos más:

Modelo de proceso en Cascada

Este modelo de proceso de tipo secuencial, que considera las actividades del proceso de desarrollo: Requerimientos, Análisis, Diseño, Implementación y Prueba, como fases separadas del proceso, asumiendo que se deben realizar en forma completa cada una de ellas antes de comenzar con la siguiente. En este modelo de proceso, la retroalimentación del cliente, respecto del producto construido, no llega sino hasta el final. Esto es muy costoso en el caso que el producto presentado al cliente no sea el producto correcto, lo que implica costos muy altos para cambiarlo, dado que se desarrollaron todas las etapas del proyecto basadas en una asunción de necesidades que no eran las correctas.

La siguiente figura muestra la representación gráfica del modelo de proceso en Cascada.

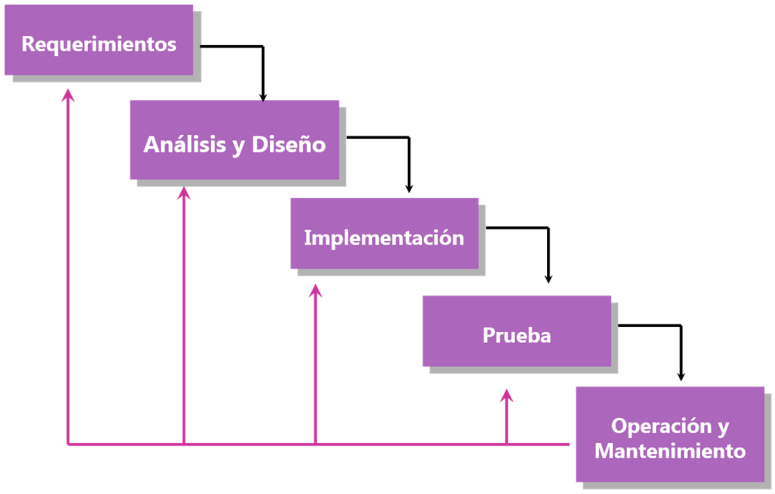


Figura 6: Modelo de Proceso o Ciclo de Vida en Cascada

Modelo de proceso en Cascada con Subproyectos

Este modelo de proceso surge como una evolución del anterior, donde se deja la totalidad de la definición del producto de software a construir al principio del proyecto y luego se divide la totalidad de los requerimientos en partes que se resolverán en "subproyectos". Dentro de cada subproyecto se realiza análisis y diseño detallado de la porción del sistema asignada como alcance a ese subproyecto y Prueba, haciendo al final la integración de cada subsistema y la prueba del sistema completo. Se puede llevar adelante subproyectos en paralelo si hay gente disponible para hacerlo. La siguiente figura muestra la representación gráfica del modelo de proceso en Cascada con Subproyectos.

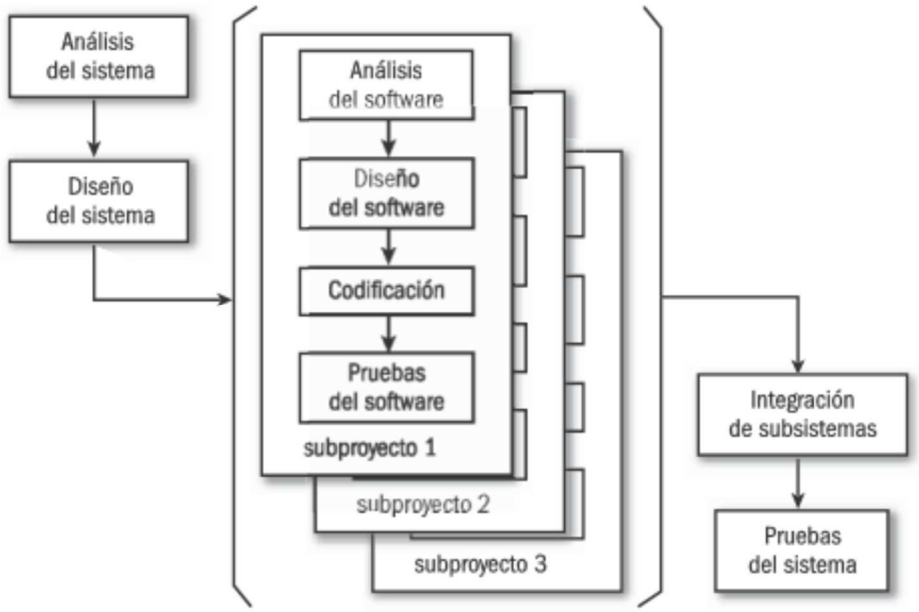


Figura 7: Modelo de Proceso o Ciclo de Vida en Cascada con Subproyectos

Modelo de Proceso Incremental

Este es un modelo de proceso o ciclo de vida que plantea que, el software se desarrolla gradualmente, por funcionalidades que al terminarse incrementan las capacidades del producto. Es una repetición del ciclo de vida en cascada, aplicándose a una porción del producto cada vez. Al finalizar cada ciclo, le entregamos una versión al cliente que contiene nuevas funcionalidades. Este modelo permite entregas al cliente antes de finalizar completamente el proyecto.

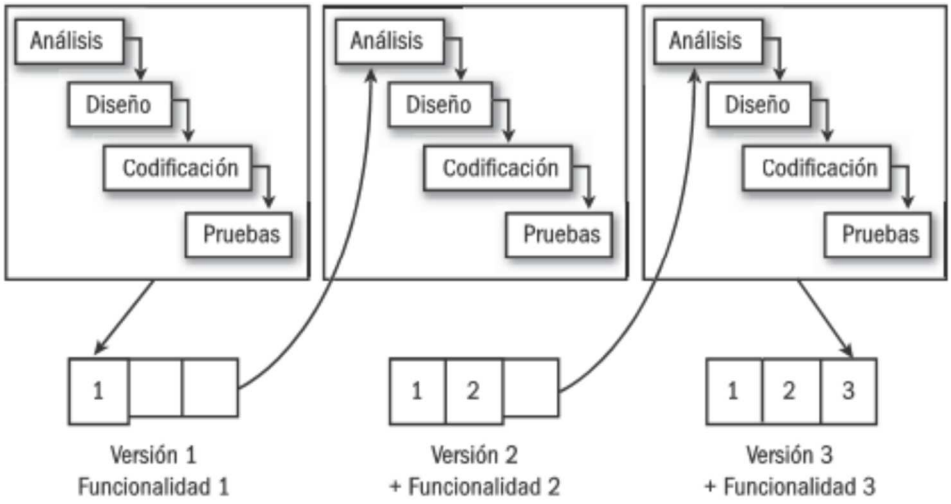


Figura 8: Modelo de Proceso o Ciclo de Vida Incremental

Modelo de Proceso Iterativo

Este modelo de proceso o ciclo de vida, también derivado del modelo de cascada, busca reducir el riesgo que surge de brechas entre las necesidades del usuario y el producto final, debido a malos entendidos durante la etapa de requerimientos.

Es la iteración de varios ciclos de vida en cascada, donde al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto. El cliente es quién luego de cada iteración, evalúa el producto, lo corrige o propone mejoras. Estas iteraciones se repetirán hasta que se obtenga un producto que satisfaga al cliente. Este modelo de proceso suele utilizarse en proyectos en los que los requerimientos no están claros de parte del usuario.

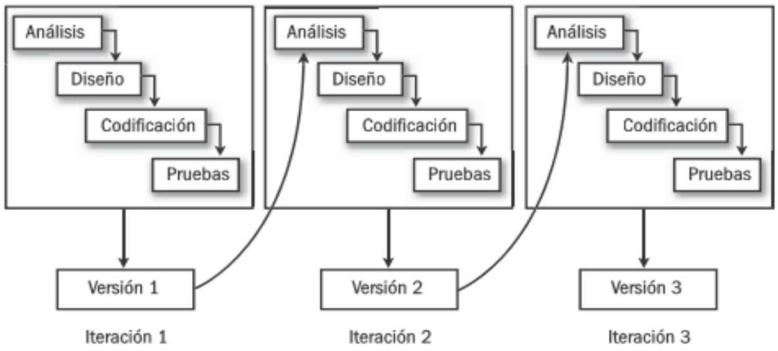


Figura 9: Modelo de Proceso o Ciclo de Vida Iterativo

Recolectar Requerimientos

> *"La parte más difícil de construir un sistema de software es decidir precisamente qué construir. Ninguna otra parte del trabajo conceptual, es tan difícil como establecer los requerimientos técnicos detallados... Ninguna otra parte del trabajo afecta tanto el sistema resultante si se hace incorrectamente. Ninguna otra parte es tan difícil de rectificar más adelante"*
>
> *Fred Brooks - "No Silver Bullet - Essence and Accidents of Software Engineering". IEEE Computer, Abril de 1987.*

Los requerimientos cumplen un papel primordial en el desarrollo de software, ya que representan algo fundamental: *la definición de lo que se desea construir*. Una de las actividades principales dentro de un proyecto es la definición y especificación de requerimientos que describan con claridad, sin ambigüedades, en forma consistente y compacta, las características y capacidades que se espera que el software satisfaga. De esta forma se pretende minimizar los problemas relacionados con el desarrollo del software basado en requerimientos equivocados.

Existen muchas definiciones para el término "requerimiento", una de las más utilizadas y abarcativa, lo define como: "condición o capacidad que debe cumplir el sistema que se desarrollará y que proviene directamente de los usuarios finales". Esta definición se ha extraído del libro Proceso Unificado de Desarrollo, referenciado al final del apunte.

El IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) lo define de manera muy similar: *"Una condición o necesidad de un usuario para resolver un problema o alcanzar un objetivo".*" Una condición o capacidad que debe estar presente en un sistema o componentes de sistema para satisfacer un contrato, estándar, especificación u otro documento formal".

Los requerimientos de software no consisten sólo de funciones, por lo tanto, podemos clasificarlos en dos tipos:

- **Requerimientos Funcionales**: son los que definen las funciones que el sistema será capaz de llevar a cabo, sin tener en cuenta restricciones físicas. Describen las transformaciones que el sistema realiza sobre las entradas para producir salidas. Por ejemplo: Imprimir una entrada para el cine, o registrar una reserva de una entrada para una función del cine.

- **Requerimientos No Funcionales**: están relacionados con las características que, de una u otra forma, puedan limitar al sistema. Especifican propiedades del sistema, como restricciones de entorno o de implementación. Ejemplos de tipos de requerimientos no funcionales son: uso (consistencia en la interfaz del usuario, documentación de usuario, material de capacitación, etc.); confiabilidad (frecuencia de fallas, certeza, tiempo mínimo entre fallas, capacidad de recuperación, etc.); desempeño (velocidad de respuesta, eficiencia, tiempo de respuesta, tiempo de recuperación, etc.); Tolerancia (capacidad de mantenimiento, facilidad de instalación, internacionalización, etc.). Para el sistema del cine, por ejemplo: "El tiempo de respuesta asociado a la impresión de cada entrada no debe superar los 5 segundos".

¿Por qué son importantes los requerimientos?

Simplemente porque de la definición, selección e implementación del conjunto correcto de requerimientos depende el éxito de un proyecto de desarrollo de software. Ningún otro tipo de error tiene consecuencias mayores que el error en la definición y especificación de los requerimientos que el producto debe satisfacer.

Los requerimientos son importantes porque permiten:

- Llegar a un acuerdo con el cliente y los usuarios sobre lo que el sistema debería hacer.
- Ayudar a los programadores y demás integrantes del equipo de desarrollo a comprender lo que el sistema debería hacer.
- Delimitar el producto de software a construir.
- Servir de base para la planificación de las iteraciones de un proyecto.
- Definir la interfaz de usuario del sistema.

¿De dónde provienen los requerimientos?

En el ámbito del desarrollo de software, se denomina **Dominio del Problema** a la parte del mundo que tiene las necesidades que darán origen al software, que luego se insertará y producirá efectos. Por otro lado, se denomina **Dominio de la Solución**, al espacio donde se construyen los artefactos de software que satisfarán las necesidades identificadas en el dominio del problema.
El espacio de intersección entre ambos dominios es la **Especificación de los requerimientos**, de allí la importancia de que exista un conjunto de requerimientos acordado y que el mismo sea correcto.

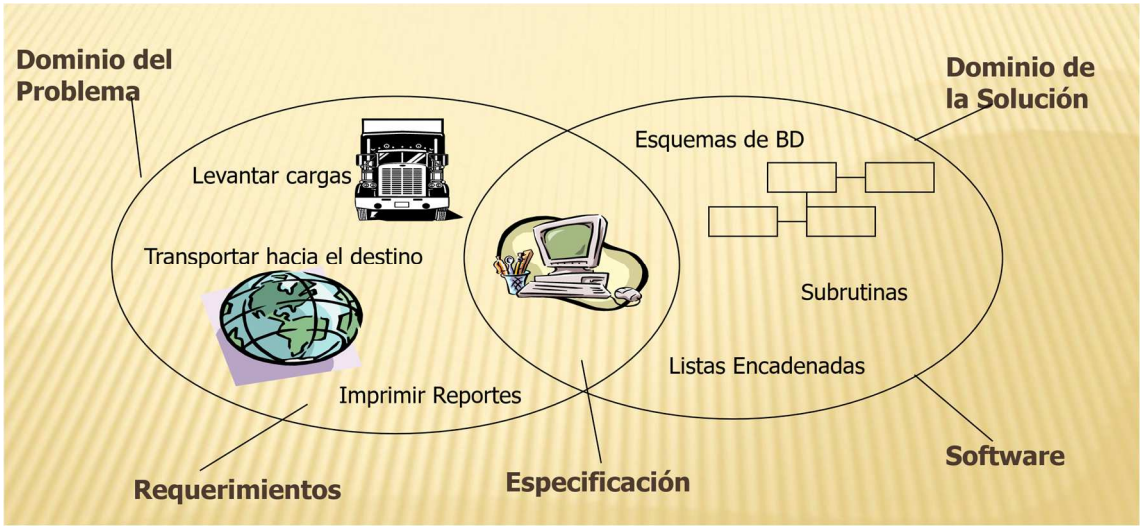


Figura 10: Dominios involucrados en el desarrollo de Software

Problemas asociados a Requerimientos

Alcanzar un acuerdo respecto de los requerimientos para un producto de software es el mayor desafío al que se enfrenta el desarrollo de software, es un problema fuertemente vinculado con la comunicación y los vínculos sociales que se establecen entre todos los involucrados en el proyecto de desarrollo. La problemática se representa en la imagen que se muestra a continuación:

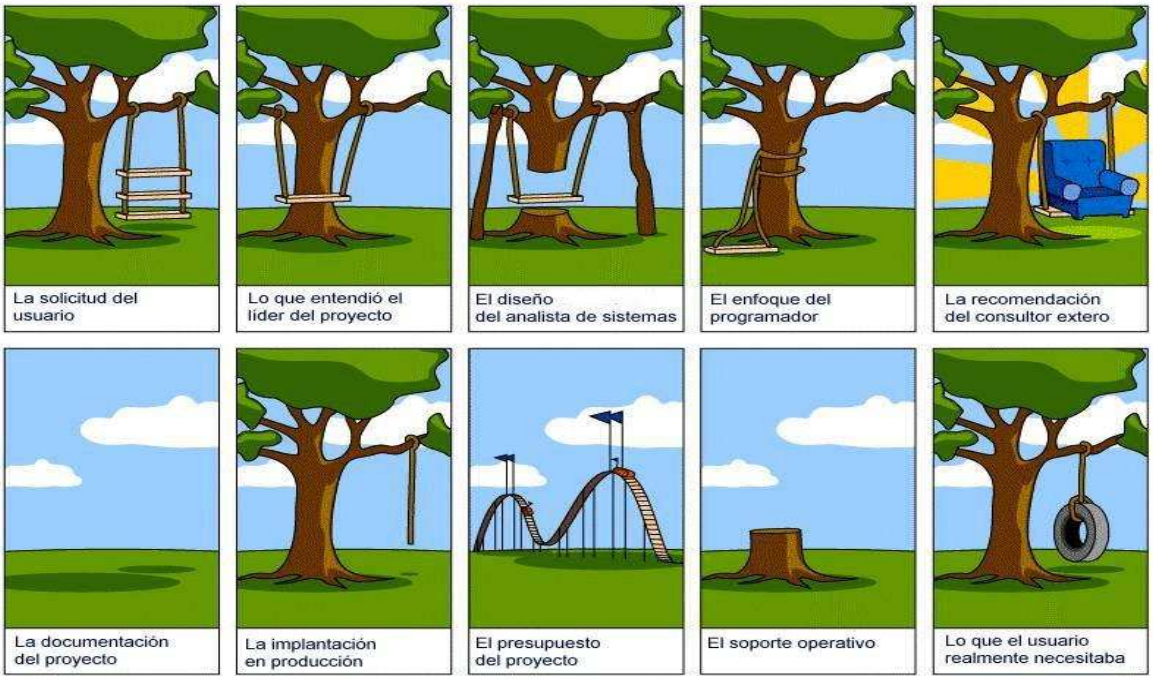


Figura 11: El problema de los requerimientos: la comunicación.

Entre los problemas más comunes, relacionados con los requerimientos, podemos incluir:

- Dificultad en el seguimiento de los cambios en los requerimientos.
- Dificultad en la especificación de los requerimientos.
- Errores en la detección de características esperadas para el software
  Mala organización.
- Los requerimientos no son siempre obvios y provienen de fuentes diferentes.
- Los requerimientos no son siempre fáciles de expresar de manera clara mediante palabras.
- Existen diferentes tipos de requerimientos a diferentes niveles de detalle.
- El número de requerimientos puede volverse inmanejable si no se controla.
- Existen varias partes interesadas y responsables, lo que significa que los requerimientos necesitan ser gestionados por grupos de personas de funciones cruzadas.
- **Los requerimientos cambian**

En la actualidad hay un abanico de técnicas y herramientas empleadas para especificar requerimientos de software. Las mismas oscilan entre la formalidad, tales como especificaciones matemáticas o lógicas; a la informalidad del lenguaje natural en forma de prosa.
Si bien, cada una de estas técnicas tiene sus fortalezas y debilidades, ninguna de ellas es perfecta y ha logrado evitar ambigüedades y malentendidos derivados de la interacción entre todas las personas que participan en el desarrollo de un producto de software.

En síntesis, la problemática de los requerimientos puede resumirse en:

- Dificultad de establecer un esquema de comprensión común entre el equipo de desarrollo y el grupo de clientes/usuarios.
- El cambio de los requerimientos como consecuencia de cambios en el dominio del problema.
- La imposibilidad de identificar la totalidad de los requerimientos de un producto al inicio del desarrollo.

La expectativa es lograr que los requerimientos cumplan con dos características:

- **Que sean objetivos**: que exista una interpretación única por parte de todos los involucrados.
- **Que sean verificables**: que exista un proceso finito en tiempo y conveniente en costo para determinar que el requerimiento se ha implementado adecuadamente.

Un enfoque ágil para el tratamiento de los Requerimientos

Dada la situación habitual en los proyectos de desarrollo do software donde clientes/usuarios se comunican con los equipos de desarrollo a través de extensos documentos de requerimientos que están sujetos a interpretaciones, lo que provoca malos entendidos y que finalmente el software construido no se corresponda con la realidad esperada.
Una de las principales razones por las cuales la utilización de especificaciones detalladas, como medio de comunicación, no conduce a resultados satisfactorios es porque sólo cubre una porción mínima (7%) del espectro de la comunicación humana: el contenido. Según Albert Mehrabian, la comunicación humana se compone de tres partes[2]:

1. En un 7%: El contenido (las palabras, lo dicho)
2. En un 38%: El tono de la voz
3. En un 55%: Las expresiones faciales

Por lo que se deduce que, para tener una comunicación sólida, completa, es necesario el contacto cara-a-cara entre los interlocutores.
En un esfuerzo orientado a que esas conversaciones existan y por resolver los problemas asociados a la identificación de los requerimientos, es que Mike Cohn creó la técnica de Historias de Usuario (en inglés User Stories), que se introduce a continuación.

[2] "Silent messages: Implicit communication of emotions and attitudes.", Albert Mehrabian, 1981

Historias de Usuario

Una **historia de usuario** (user story) es una descripción corta de una funcionalidad, valuada por un usuario o cliente de un sistema.

Componentes de una Historia de Usuario

Una Historia de Usuario se compone de 3 elementos, también conocidos como "las tres Cs"[3] de las Historias de Usuario:

- **Card (Ficha)** – Toda historia de usuario debe poder describirse en una ficha de papel pequeña. Si una Historia de Usuario no puede describirse en ese tamaño, es una señal de que estamos traspasando las fronteras y comunicando demasiada información que debería compartirse cara a cara.
- **Conversación** – Toda historia de usuario debe tener una conversación con el Dueño de Producto (Product Owner). Una comunicación cara a cara que intercambia no sólo información sino también pensamientos, opiniones y sentimientos.
- **Confirmación** – Toda historia de usuario debe estar lo suficientemente explicada para que el equipo de desarrollo sepa qué es lo que debe construir y qué es lo que el Product Owner espera. Esto se conoce también como *Pruebas de Aceptación*.

Redacción de una Historia de Usuario

Mike Cohn[4] sugiere una determinada forma de redactar Historias de Usuario bajo el siguiente formato:

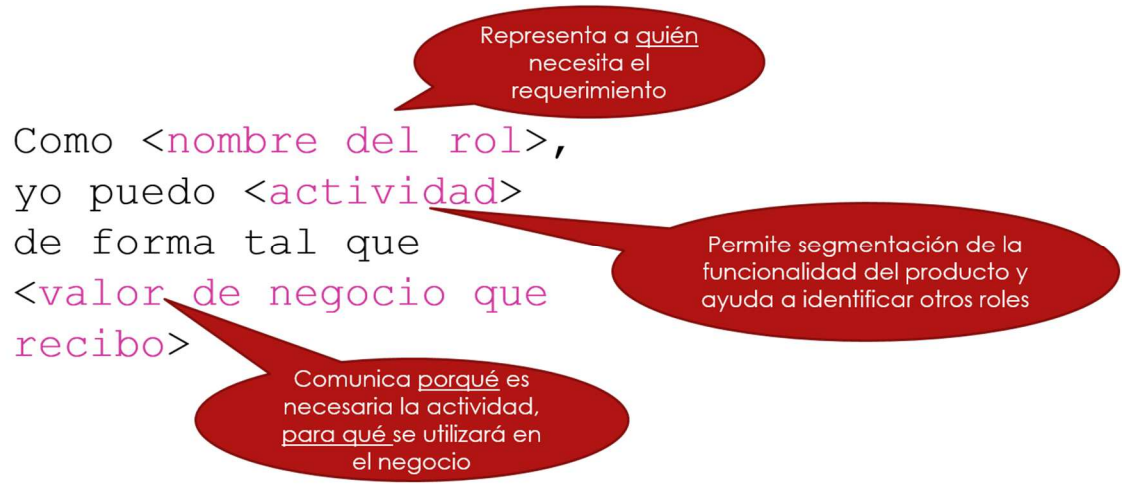


Figura 12: Sintaxis de una historia de usuario.

[3] "Essential XP: Card, Conversation, Confirmation", Ron Jeffries, 2001

[4] "Advantages of the "As a user, I want" user story template", Mike Cohn, 2008

Como Conductor quiero buscar un destino a partir de una calle y altura para poder llegar al lugar deseado.

Nota: la altura es el número de calle.

Figura 13: Ejemplo de una historia de usuario.

Las ventajas de escribir las historias de esta forma son, principalmente:

- **Primera** Persona: La redacción en primera persona de la Historia de Usuario invita a quien la lee a ponerse en el lugar del usuario.
- **Priorización**: Tener esta estructura para redactar la Historia de Usuario ayuda al responsable del producto a priorizar. Le permite visualizar mejor cuál es la funcionalidad, quien se beneficia y cuál es el valor de la misma.
- **Propósito:** Conocer el propósito de una funcionalidad permite al equipo de desarrollo plantear alternativas que cumplan con el mismo objetivo en el caso de que el costo de la funcionalidad solicitada sea alto o su construcción no sea viable.

- ***Como Conductor quiero buscar un destino a partir de una calle y altura para poder llegar al lugar deseado.***
- ***Nota: la altura es el número de calle.***

Pruebas de Aceptación

- ❑ ***Probar buscar un destino en un país y ciudad existentes, de una calle existente y la altura existente (pasa).***
- ❑ ***Probar buscar un destino en un país y ciudad existentes, de una calle inexistente (falla).***
- ❑ ***Probar buscar un destino en un país y ciudad existentes, de una calle existente y la altura inexistente (falla).***
- ❑ ***Probar buscar un destino en un país inexistente (falla).***
- ❑ ***Probar buscar un destino en País existente, ciudad inexistente (falla).***
- ❑ ***Probar buscar un destino en un país y ciudad existentes, de una calle existente y demora más de 30 segundos (falla).***

Figura 14: Ejemplo de una historia de usuario con sus pruebas de aceptación.

INVEST - Características de una Historia de Usuario

El modelo INVEST es conjunto de características recomendadas para evaluar la calidad de una historia de usuario, el uso de la regla mnemotécnica "INVEST"[5], ayuda a su recordación:

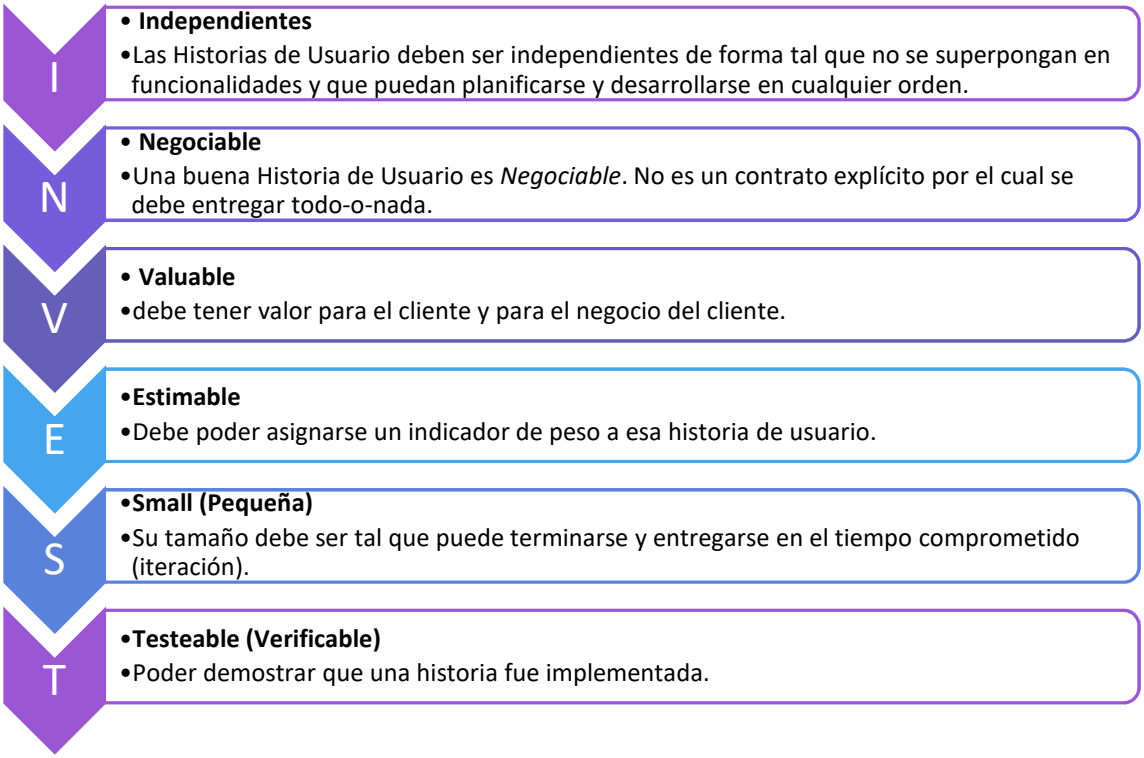


Figura 15: Modelo INVEST para historias de usuario.

Independiente (I)

Las historias de usuario deben ser independientes de forma tal que no se superpongan en funcionalidades y que puedan planificarse y desarrollarse en cualquier orden. Muchas veces esta característica no puede cumplirse para el 100% de las historias. El objetivo que debemos perseguir es preguntarnos y cuestionarnos en cada Historia de Usuario si hemos hecho todo lo posible para que ésta sea independiente del resto.

Negociable (N)

Una buena Historia de Usuario es negociable. No es un contrato explícito por el cual se debe entregar todo-o-nada. Por el contrario, el alcance de las Historias (sus criterios de aceptación) podrían ser variables: pueden incrementarse o eliminarse con el correr del desarrollo y en función del feedback del usuario y/o la performance del Equipo. En caso que uno o varios criterios de aceptación se eliminen de una Historia de Usuario, estos se transformarán en una o

[5] "INVEST in Good Stories, and SMART Tasks", Bill Wake, 2003

varias Historias de Usuario nuevas. Esta es la herramienta útil para negociar el alcance de cada ejecución del proyecto.

Valorable (V)

Una historia de usuario debe poder ser valorada por el responsable del producto. En general, esta característica representa un desafío a la hora de dividir historias de usuario; puesto que, si el responsable del producto no puede visualizar el valor de negocio de esa historia, no podrá priorizarla.

Estimable (E)

Una historia de usuario debería ser estimable. Mike Cohn[6], identifica tres razones principales por las cuales una Historia de Usuario no podría estimarse:

- **La historia de usuario es demasiado grande**. En este caso la solución sería dividir la Historia de Usuario en historias más pequeñas que sean estimables.
- **Falta de conocimiento funcional**. En este caso es necesario tener una nueva conversación para obtener mayor detalle de la historia.
- **Falta de conocimiento técnico**. Muchas veces el Equipo de Desarrollo no tiene el conocimiento técnico suficiente para realizar la estimación. En estos casos debe separarse esa historia hasta tanto se obtenga la información necesaria.

Pequeña (Small)

Toda historia de usuario debe ser lo suficientemente pequeña de forma tal que permita ser estimada y completada por el equipo de desarrollo. Las descripciones de las historias de usuario también deberían ser pequeñas, y tener tarjetas de tamaño pequeño ayuda a que eso suceda.

Verificable (Testable)

Una buena Historia de Usuario es Verificable. Se espera que el responsable del producto no sólo pueda describir la funcionalidad que necesita, sino que también logre verificarla (probarla). Si el responsable de producto no sabe cómo verificar una historia de usuario o no puede enumerar los criterios y pruebas de aceptación, esto podría ser una señal de que la historia en cuestión no es lo suficientemente clara.

[6] "User Stories Applied", Mike Cohn, 2003

Planificar Proyectos de Software

Cuando el mapa y el territorio no coinciden… confía en el territorio

La planificación y la estimación son críticas para el éxito de cualquier proyecto de desarrollo de software de cualquier tamaño. Los planes guían nuestras decisiones de inversión. Podríamos iniciar un proyecto específico si estimamos que llevará 6 meses y costará 1 millón de pesos, pero rechazaríamos el mismo proyecto si costara 4 millones y llevará 2 años. Los planes ayudan a saber quiénes deben estar disponibles para trabajar en un proyecto durante un período de tiempo. Los planes ayudan a saber si un proyecto podrá entregar la funcionalidad que el usuario necesita y espera. Sin planes exponemos a los proyectos a un sin número de problemas.

La planificación es difícil y los planes a menudo están equivocados. Frente a esta situación los equipos de proyecto suelen responder yéndose de un extremo a otro. O no planifican nada en absoluto o bien, ponen mucho esfuerzo en los planes, convencidos de que éstos deben estar correctos.

Los equipos que no planifican no pueden contestar las preguntas más básicas, tales como: ¿Cuándo estará hecho? ¿Podremos planificar el release para Junio? ¿Cuánto costará?

Los equipos que se exceden en la planificación, se convencen a sí mismos de que cualquier plan puede ser correcto. Sus planes puede que sean más minuciosos, pero no significa que sean más precisos o útiles.

Un buen proceso de planificación ayuda a:

- Reducir Riesgos
- Reducir incertidumbre
- Soportar la toma de decisiones
- Generar confianza
- Transmitir información

¿Qué hace que un plan sea bueno?

Un buen plan es aquel que los interesados en el proyecto encuentran lo suficientemente confiable como para utilizarlo como base para tomar decisiones. Al principio del proyecto, el plan podrá decir que el producto se liberará en el tercer trimestre del año, en lugar del segundo, contendrá una descripción aproximada de las características. Más tarde en el proyecto, para mantenerlo útil para la toma de decisiones, el plan deberá ser más preciso.

La planificación en el contexto de los procesos de control empírico y definido

La elección del proceso de control que utilizaremos para el proyecto define el enfoque de planificación que se utilizará para el proyecto.

Si la elección es un **proceso de control *definido***, el proceso de planificación es más detallado, formal y parte de la concepción de que el alcance del proyecto[7] puede fijarse de forma completa y en etapas tempranas.

La responsabilidad sobre la planificación del proyecto recae en el rol del Líder o Jefe de Proyecto, así como también la asignación de las actividades a los demás miembros del equipo de proyecto. También es responsable de negociar con clientes, usuarios y demás interesados del proyecto.

La planificación de proyectos con procesos de control definido, conocida como planificación tradicional implica:

- Definir objetivos para el proyecto, que deben ser claros y alcanzables.
- Definir alcance para el proyecto, que está directamente relacionado con el alcance del producto o servicio que se entregará.
- Decidir el proceso y el ciclo de vida que se utilizará.
- Definir roles y responsabilidades para cada uno de los miembros del equipo.
- Estimar los recursos que serán requeridos.
- Estimar tamaño, esfuerzo, tiempo y costo.
- Identificar y realizar un análisis de riesgos.
- Realizar el cronograma para el proyecto.
- Definir el presupuesto.
- Determinar el mecanismo de monitoreo y control y las métricas que se utilizarán.
- Crear planes para las actividades de soporte, tales como administración de configuración y aseguramiento de calidad.

Existen diversas metodologías y modelos, que plantean como llevar a cabo la planificación de un proyecto. El tratamiento de estas metodologías está fuera del alcance de este material.

[7] **Alcance del Proyecto**: se define como todo el trabajo y sólo el trabajo que debe realizarse para cumplir con el objetivo del proyecto.

Si la elección es un **proceso de control *empírico***, la planificación del proyecto se realiza con un enfoque ágil. Los planteos que sustentan la planificación de proyectos con este enfoque están centrados en contrarrestar las fallas que con frecuencia afectan a la planificación de proyectos de software, las que se resumen a continuación:

- La planificación hace foco en tareas, no en características de software, lo que se traducen en un desfasaje, puesto que completar tareas en el desarrollo de software no necesariamente implica tener un producto en condiciones de ser entregado al cliente.
- Suelen consumirse todos los recursos asignados al proyecto, sin que esto signifique alcanzar los objetivos comprometidos.
- Actividades no independientes, provocan demoras y cuellos de botella.
- La asignación de múltiples tareas a una misma persona, en forma simultánea, causa demoras adicionales.
- Las características de software no se entregan teniendo en cuenta su prioridad para el negocio.
- Ignoramos el hecho de que el desarrollo de software funciona en un universo de incertidumbre, inherente a su naturaleza, el ignorarlo hace que se tomen decisiones equivocadas.
- Se confunden estimaciones con compromisos.

A continuación, se describen las características generales de la planificación en contextos ágiles, basada en procesos de control empírico. Las mismas son:

- Foco en el proceso, en la acción de planificar más que en el plan, como artefacto resultante.
- Promueve la acción de planificar, en forma reiterativa.
- El plan debe ser fácilmente modificable.
- El esfuerzo de planificar se distribuye a lo largo del proyecto.

En las secciones finales de este material, se describen algunas metodologías ágiles, entre las cuales está SCRUM, que es en la actualidad la más referenciada como framework de gestión ágil de proyectos. Las prácticas que propone SCRUM están sustentadas en los siguientes principios:

- Trabajar como equipo.
- Iteraciones cortas.
- Entregar algo que el cliente pueda utilizar, en cada iteración.
- Foco en las prioridades de negocio, entregamos valor de negocio, no piezas de software.
- Inspeccionar y adaptar frecuentemente, tanto el producto que se construye como el proceso que se utiliza para hacerlo.

Estimaciones en el Software

¿Qué es estimar?

La estimación puede definirse como el proceso de encontrar una aproximación sobre una medida, lo que se ha de valorar con algún propósito. El resultado del proceso es utilizable incluso si los datos de entrada estuvieran incompletos, inciertos, o inestables.

En el ámbito de la estadística, la estimación implica “usar el valor de una estadística derivada de una muestra para determinar el valor de un parámetro correspondiente a una población”; la muestra establece que la información puede ser proyectada a través de diversos factores, formal o informalmente y descubrir la información que falta.

Las estimaciones tienen asociado un valor de probabilidad, dado que no se realizan estimaciones en universos de certeza, lo que implica que los valores que utilizamos para estimar están basados en presunciones a cerca de la realidad que desea estimarse.

Cuando una estimación resulta ser incorrecta, se denomina “sobreestimar” si la estimación superó el resultado real y “subestimar” si la estimación fue un valor inferior respecto del resultado real.

Estimaciones en el Software

Si bien la definición que define la estimación *como la predicción de cuánto costará y cuánto durará un proyecto* es correcta; la estimación de proyectos de software interactúa con objetivos de negocio, compromisos y control.
Un *objetivo* es una declaración de una meta deseable para el negocio; por ejemplo: “Necesitamos tener la versión 2.1 del producto X, lista para mostrarla en la exposición de mayo próximo”. O "Estas funcionalidades deben estar listas para el 1ero. de marzo, para poder cumplir con las regulaciones del gobierno nacional."
Los negocios tienen razones importantes para establecer objetivos independientes de las estimaciones de software. No obstante, que los objetivos sean deseables e incluso necesarios u obligatorios, no significa que sean alcanzables.
Mientras el objetivo es la descripción de una meta deseable para el negocio, un *compromiso* es una promesa de entregar una funcionalidad definida con un nivel de calidad específico para una fecha determinada. Un compromiso puede ser lo mismo que una estimación, o puede ser más agresivo o más conservador que una estimación. En otras palabras, no debería asumirse que las estimaciones y los compromisos son lo mismo porque no lo son.

Distinguir entre estimaciones, objetivos y compromisos.

Hacer una buena estimación de software antes de ofertar un proyecto nos puede ayudar a detectar proyectos que no conviene abordar y que no son rentables. Aunque la realidad diga que normalmente el negocio, o la parte comercial, fija inamoviblemente y sin estimación previa, el tiempo del proyecto, esto no debería evitar las estimaciones, ya que estas nos ayudarán a saber de qué tamaño es el problema en que nos hemos metido. Mejor saber al principio que es imposible hacer el proyecto en el tiempo ofertado que al final, cuando ya hay muy poco o ningún margen de maniobra.

Las estimaciones y los planes están relacionados, sin embargo, estimar no es planificar y planificar no es estimar. La estimación debería tratarse como un proceso *analítico e imparcia*l, la planificación debería tratarse como un proceso parcial, de búsqueda de objetivos. Con la estimación, es riesgoso querer estimar cómo llegar a una respuesta particular. El objetivo es la exactitud, no la búsqueda de un resultado particular. Por otro lado, el objetivo de la planificación es buscar un resultado particular. Deliberada y adecuadamente, direccionamos los planes para poder alcanzar resultados específicos. Planificamos recursos y medios específicos para lograr fines específicos.

Las estimaciones son la base de los planes, pero los planes no tienen que ser iguales a las estimaciones. Si una estimación es drásticamente diferente de los objetivos, el proyecto podrá necesitar reconocer una brecha y contabilizar un nivel alto de riesgo asociado. Si las estimaciones están cerca de los objetivos, entonces lo planes asumen menos riesgos.

Las estimaciones y los planes son importantes, pero no son lo mismo. Combinarlas puede resultar en pobres estimaciones y pobres planes.

¿De dónde vienen los errores de estimación?

Los errores en la estimación pueden provenir de 4 fuentes genéricas:

- Información imprecisa acerca del software a estimar, es decir no están claros los requerimientos o características del software que se va a desarrollar.
- Información imprecisa acerca de la capacidad del equipo y/o la empresa que realizará el proyecto.
- Demasiado caos en el proyecto o falta de gestión en el mismo.
- Imprecisión y/o errores generados por el mismo proceso de estimación.

Otro factor muy relevante que provoca errores en las estimaciones es *olvidar actividades*. Esto ocurre cuando al estimar no se consideran todas las tareas que deben realizarse para desarrollar el producto. Por lo general se considera únicamente el esfuerzo necesario para la programación y se omiten actividades como la gestión del proyecto, el análisis, el testing, el seguimiento del proyecto, la creación de documentación de soporte, programa de instalación, ayuda, conversión de datos. También suelen olvidarse aspectos como problemas con el hardware y/o el software, vacaciones, enfermedades de los miembros del equipo, etc.

Estimaciones de Software en Procesos de Control Definidos

Si estamos gestionando el proyecto basado en procesos de control definidos, el proceso de estimación implica la estimación de:

Figura 16: Proceso de estimación para proyectos de gestión tradicional

Tamaño

El tamaño del producto de software a construir es lo primero que debe estimarse. Con esta estimación respondemos a la pregunta ¿cuán grande es? ¿Qué vamos a construir un velero o un portaaviones? ¿Un departamento de un ambiente o un rascacielos? La base del proceso de estimación es acordar cuán grande es el producto de software a construir.

La estimación de tamaño es el paso intelectual más difícil, luego de la determinación de los requerimientos, pero no es imposible; muchas veces se saltea y se trabaja directamente en la estimación de tiempo. De todos modos, si no se ha pensado detenidamente en lo que se pide que se construya, no se tiene una buena base para predecir un cronograma o para evaluar cómo los cambios pueden afectar al cronograma.

Para estimar el tamaño de un producto de software debemos elegir qué contar. En las etapas tempranas del proyecto podemos contar requerimientos, características, casos de usos, etc. En la mitad del proyecto, es decir luego del análisis y diseño, podemos contar puntos de función, pedidos de cambios, páginas web, reportes, pantallas, tablas, etc. Más avanzado el proyecto: defectos, clases, etc. La recomendación es elegir algo que esté disponible lo más pronto posible en el proyecto. Lo que se elige para contar, debe requerir poco esfuerzo.

Esfuerzo

La estimación de esfuerzo, es una medida que determina el trabajo que una persona necesita realizar para obtener el producto de software que se estimado con la estimación de tamaño. Acá debemos responder a la pregunta ¿cuánto trabajo requerirá? Por consiguiente, la estimación de esfuerzo se deriva de la estimación de tamaño. El esfuerzo se mide en horas persona lineales, esto significa que se asume para realizar la estimación, que el trabajo lo realizará una sola persona trabajando con una cosa por vez, sin hacer trabajo en paralelo.

Recordando que una de las características del software, que se presentaron al principio de este material, es que es una actividad "*humana intensiva*", es decir el trabajo de las personas, su intelecto, es la materia prima principal para la creación del software. Esta característica hace que esta industria sea muy dependiente de las personas involucradas en el desarrollo del producto. En este sentido cuando estimamos esfuerzo debemos tener en cuenta el esfuerzo *de quien es el que realizará el trabajo*, puesto que factores como el nivel de conocimiento en la tecnología que se utilizará, en el dominio del problema, los años de experiencia laboral, entre otros, son factores modificadores del esfuerzo. Es decir, las horas que necesita un profesional senior con 15 años de experiencia y amplio manejo de las tecnologías que se utilizarán en el

proyecto, no son las mismas que las que necesita un profesional recién iniciado, que no trabajó nunca antes, para hacer la misma pieza de software.

Tiempo

El siguiente paso en el proceso de estimación es la determinación de tiempo calendario, es decir damos respuesta a la pregunta ¿cuándo estará terminado? La estimación de tiempo puede realizarse en horas, en días, meses u años, dependiendo del cuán grande sea el producto.

Para obtener la estimación de tiempo, que se deriva de la estimación de esfuerzo, debemos incorporar al análisis varias consideraciones de contexto del proyecto, a saber:

- ¿Cuántas personas trabajarán en el proyecto?
- ¿Qué asignación diaria de horas tendrá cada persona para el proyecto?
- ¿Cuántos días a la semana se trabajará?
- ¿Se pueden solapar actividades?
- ¿Qué dependencia hay entre el trabajo que cada persona debe realizar?

Respondiendo a esta serie de preguntas se puede determinar una fecha en la que el producto de software podría ser liberado al cliente.

Recursos

Si bien hemos mencionado antes que el esfuerzo de las personas es la variable más significativa a la hora de desarrollar software, también se necesitan otros recursos, algunos de ellos son necesarios siempre como computadoras, software de base, herramientas de desarrollo, ambientes de desarrollo y de prueba, licencias para el software que se utilizará, si fuera el caso.

Hay otros recursos que son particulares para ciertas situaciones, y de ser necesarios, el no contar con ellos en tiempo y en forma, los transforma en críticos y puede representar un riesgo para la consecución de los objetivos del proyecto. Por ejemplo, el producto que se desarrollará es para ser utilizado con los anteojos de google (google glass), por consecuencia se necesitan anteojos suficientes para que los desarrolladores y los analistas de prueba los tengan disponibles cuando los necesiten.

En este caso lo que se estima es la cantidad de recursos de cada tipo y el momento en el que éstos deberán estar disponibles.

Costos

La estimación de costos es la responsable de asignar un valor monetario al producto de software que se desarrollará. Debe responder a la pregunta ¿Cuánto costará? El costo más significativo en el desarrollo de software es el costo del esfuerzo, es decir el costo del trabajo de las personas involucradas en su creación. El resto de los costos, si bien se los considera, son significativamente menores.

En la determinación del costo del esfuerzo, debemos considerar el valor del costo de trabajo de cada una de las personas afectadas al proyecto y multiplicarlo por la cantidad de horas que se estimó que esa persona dedicará al proyecto. Algunos proyectos realizan la estimación de costos con valores discriminados por rol, es decir valores hora para líderes de proyecto, analistas de

sistemas, desarrolladores, analistas de prueba, arquitectos, diseñadores de base de datos, diseñadores de experiencia de usuario, etc. Otros proyectos realizan la estimación asumiendo lo que se conoce como "tarifa plana", es decir tomar un único valor de referencia y lo multiplican por la cantidad de horas obtenida en la estimación de esfuerzo. Acá debe considerarse la posibilidad de variación en los costos del esfuerzo, si se requerirá trabajar horas extra o en horarios diferentes a los horarios laborales normales, en días feriados, o fines de semana.

La determinación de los demás costos del proyecto, se obtiene considerando la adquisición (ya sea por compra o alquiler) de los recursos estimados durante la estimación de recursos.

Si bien en la figura anterior, se presenta el proceso de estimación como lineal, se puede variar en la realización de los procesos de tiempo y recursos o realizarlos en paralelo. La estimación de tamaño debería ser siempre la primera y luego derivar de ella la estimación de esfuerzo. Así como también se recomienda dejar para el final la determinación de los costos del proyecto, ya que los mismos pueden tener variaciones si los resultados de las estimaciones anteriores varían.

Es importante diferenciar el costo de un producto de su valor o precio de venta. Una vez obtenido el costo por algún método como el que se describía antes, se deriva de este el precio de venta para el producto, que depende de diversos factores y situaciones, las cuales están fuera del alcance de este apunte.

Métodos de Estimación

Existen numerosos métodos de estimación software, si bien estos se pueden clasificar en dos grandes grupos: aquellos que siguen un enfoque heurístico o los que siguen un enfoque paramétrico.

Los métodos heurísticos se basan en la experiencia, y los principales son:

- **El método basado en juicio experto individual**. Más comúnmente llamado "a ojo", y que consiste básicamente en preguntar a alguien con experiencia (o sin ella) cual es en su opinión la estimación de software. Es el método más usado, aunque que tiene el problema de que se depende mucho del experto, de la experiencia que tenga y que además tiene el riesgo de que un día el experto deje la empresa y nos quedemos sin forma de estimar.
- **El método basado en juicio experto grupal**. El objetivo es lograr un consenso basado en la discusión entre expertos. El Método Delphi se basa en:
    - El anonimato de los participantes
    - La repetición y retroalimentación controlada
    - La respuesta del grupo en forma estadística

  Basados en el Método Delphi, Barry Boehm y John Farquhar elaboraron la variante conocida desde entonces como Wideband Delphi. Se trata de una técnica basada en la obtención de consensos para la estimación de esfuerzos, llamada "wideband" porque a diferencia del conocido método Delphi, esta técnica requiere de un mayor grado de interacción y discusión entre los participantes. Wideband Delphi presenta los siguientes pasos para su ejecución:

  1. Un coordinador presenta a cada experto una especificación y un formulario de estimación.
  2. El coordinador convoca a una reunión de grupo en la que los expertos debaten temas de estimación.

3. Los expertos llenan los formularios de forma anónima.
4. El coordinador prepara y distribuye un resumen de las estimaciones.
5. El coordinador convoca a una reunión de grupo, centrándose específicamente en aquellas estimaciones donde los expertos varían ampliamente.
6. Los expertos completan los formularios una vez más de forma anónima, y los pasos 4 a 6 son repetidos para tantas rondas como sea necesario.

- **El método por analogía**. Que es una importante evolución del anterior, ya que se basa en experiencias documentadas de cómo fueron los tiempos en proyectos previos. El método compara el proyecto a estimar con proyectos terminados previamente que sean similares. Aquí la importante aportación es que disponemos de un método, y de que la experiencia se va guardando en una base de datos. Para utilizar este método es importante definir previamente que características de los proyectos y productos se guardarán en la base de datos para poder estimar, por ejemplo, tecnologías utilizadas, tamaño, esfuerzo, cantidad de personas en el equipo

Los métodos paramétricos de estimación software

- COCOMO (Constructive Cost Model) II, que estima el esfuerzo (horas hombre) del proyecto. Para estimar el esfuerzo requiere previamente una estimación del tamaño (funcionalidad, líneas de código, etc.
- SLIM (Software LIfecycle Management), que de manera similar contiene un conjunto de fórmulas de estimación software. Estas fórmulas se extrajeron de estudiar grandes bases de datos de proyectos, observando cómo se comportaron las estimaciones software y distribuciones de esfuerzo.

Los dos anteriores sirven principalmente para obtener una estimación del esfuerzo y se basan en que exista previamente un cálculo del tamaño del software a desarrollar. Para determinar el tamaño del software.

Los métodos paramétricos se basan fundamentalmente en la aplicación de fórmulas algorítmicas, que tiene en cuenta parámetros diferentes del proyecto.

Incertidumbre en las estimaciones

En gestión de proyectos, el cono de la incertidumbre describe la evolución de la incertidumbre durante la ejecución de un proyecto. Al comienzo, poco se conoce sobre el producto y el resultado del trabajo, por tanto, las estimaciones están sujetas a una gran incertidumbre. A medida que avanzamos en el proyecto obtenemos mayor conocimiento sobre el entorno, la necesidad de negocio, el producto y el proyecto mismo. Esto causa que la incertidumbre tienda a reducirse progresivamente hasta desaparecer. Esto ocurre generalmente hacia el final del proyecto: no se alcanza una incertidumbre del 0% sino hasta haber finalizado. Muchos ambientes cambian tan lentamente, que la incertidumbre reinante puede ser considerada constante, durante la duración de un proyecto típico. En estos contextos, la gestión tradicional de proyectos, basada en procesos definidos, hace hincapié en lograr un entendimiento total mediante el análisis y la planificación detallada antes de comenzar a trabajar. De esta manera, los riesgos son reducidos a un nivel en el que pueden ser gestionados cómodamente. En estas situaciones, el nivel de incertidumbre decrece rápidamente al comienzo y se mantiene prácticamente constante (y bajo) durante la ejecución del proyecto.

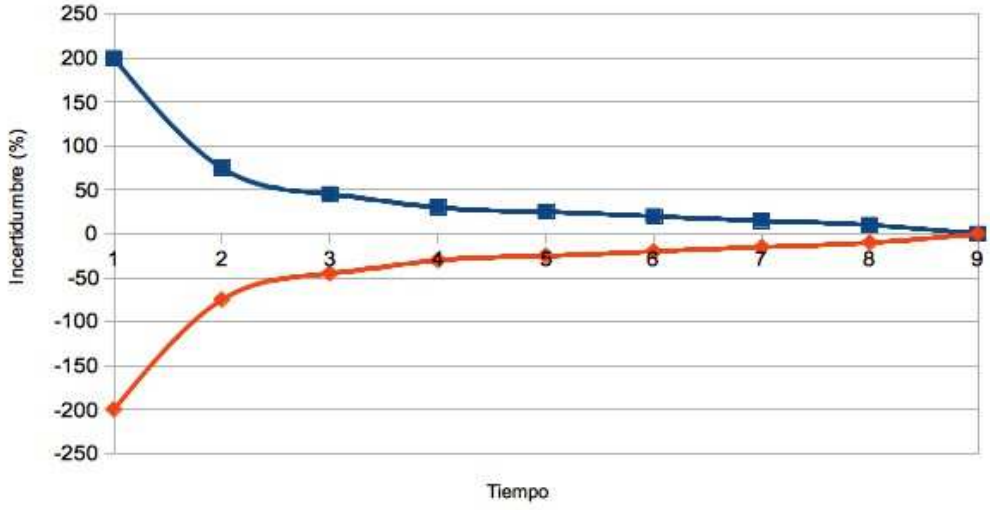


Figura 17: Cono de la Incertidumbre en un contexto estable.

El contexto del software, por el contrario, es un contexto altamente volátil donde hay muchas fuerzas externas actuando para incrementar el nivel de incertidumbre, como lo son los cambios producidos en el contexto de negocio, los cambios en los requerimientos, los cambios tecnológicos y aquellos surgidos por la mera existencia del producto construido que acontecen durante la ejecución del proyecto. Debido a esta razón, se requiere trabajar activa y continuamente en reducir el nivel de incertidumbre.

Investigaciones han demostrado que, en la industria del software, el nivel de incertidumbre al comienzo de un proyecto es del +/- 400%, esta incertidumbre tiende a decrecer durante la evolución del proyecto, pero sin garantías de ello.

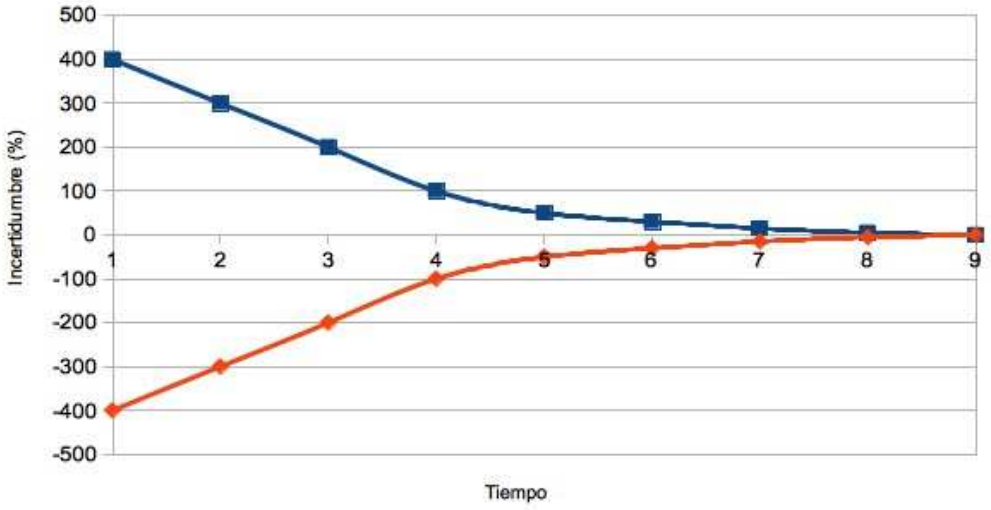


Figura 18: Cono de la Incertidumbre en desarrollo de software.

Estimaciones en contextos inciertos

Como es de esperar según los gráficos previos, proveer una estimación precisa en etapas tempranas de un proyecto tiene como consecuencia un compromiso poco probable de ser cumplido.

A medida que adquiramos conocimiento, nuestras estimaciones se harán cada vez más precisas. El problema aparece a la hora de estimar cuando muchas de las decisiones se toman en base a supuestos que probablemente no sucedan o no sean los correctos.

"La propia palabra "estimación" deja en claro que no calculamos un valor exacto, determinístico. De hecho, toda estimación tiene supuestos, y estos supuestos suman incertidumbres."

Estimaciones en ambientes Ágiles

Las estimaciones en proyectos basados en procesos de control empírico, como son los proyectos que utilizan metodologías ágiles, proponen comenzar a trabajar en un proyecto sin la necesidad de tener una estimación precisa basada en supuestos, y siendo conscientes de que la estimación inicial es de un orden de magnitud probable, para poder ganar experiencia rápidamente y así estimar con mayor certeza prescindiendo de supuestos.

Para mitigar el riesgo de proveer estimaciones incorrectas, en las metodologías ágiles se opta por reducir la precisión de las estimaciones en función de cuánto conocimiento se tiene sobre el esfuerzo que se requiere estimar. De esta manera, los "requerimientos" y sus "estimaciones" se categorizan en diferentes niveles de precisión.

Los niveles de precisión dependen de las características identificadas para el producto, contenidas en la Lista de Producto ((Product Backlog) (PBI´s) y las estimaciones recomendadas:

Podemos enumerar la siguiente escala de PBI´s y sus estimaciones:

1. **Alto Nivel**: **EPICA**, (la épica puede definirse como un bloque funcional grande, del que no se tiene información detallada. En este caso, se recomienda estimar con una técnica llamada "Talles de Remera", que consiste en seleccionar una escala de talle de remera con valores como: XS, S, M, L, XL y luego asignarle a cada Épica un talle de remera.

2. **Nivel Medio**: **Historia de Usuario (User Story)** (funcionalidad) estimada en Puntos de Historia (Story Points), utilizando la Sucesión de Fibonacci (1,1,2,3,5,8…) para asignar los puntos de historia a cada historia de usuario.

Los puntos de historia (story point) son una unidad de medida específica (de cada equipo) que mide complejidad, riesgo y esfuerzo, asociado a una historia de usuario. Es lo que "el kilo" a la unidad de nuestro sistema de medición de peso. Da idea del "peso" de cada historia y decide cuán grande (compleja) es. La complejidad de una historia tiende a incrementar exponencialmente.

Al comenzar el proyecto, nuestro Product Backlog se compone de bloques funcionales que podemos estimar según sus tamaños:

- XS – Muy Pequeño
- S – Pequeño
- M – Medio
- L – Grande
- XL – Muy Grande

Esto nos permitirá tener una primera aproximación a la problemática del negocio y a las características del producto que se desea construir. Conociendo las prioridades de dichos bloques funcionales, se toman los de mayor prioridad y se descomponen en funcionalidades más específicas, logrando de esa manera ítems de menor nivel, llamados Historias de Usuario o User Stories. A las Historias de Usuario las estimaremos utilizando la sucesión Fibonacci modificada:

- 0, 1, 2, 3, 5, 8, 13, 21, ….

Para estimar las Historias de Usuario se utiliza comúnmente, una técnica comparativa llamada Poker Planning o Poker Estimation, que describiremos un poco más adelante.

Finalmente, llegamos al nivel más bajo de estimación: la estimación de tareas o actividades.

Bajo Nivel: tareas o actividades estimadas en horas ideales[8], preferiblemente dividirlas para que dure menos de un día.

Sólo aplica a las tareas o actividades obtenidas de la descomposición de las Historias de Usuario que han sido seleccionadas para formar parte de una determinada iteración del proyecto. En la reunión de planificación de dicha iteración, estas Historias de Usuario son divididas por el Equipo, en tareas o actividades y a su vez, las tareas o actividades, estimadas en horas ideales. Lo importante es que la estimación en horas sólo se realiza para las actividades de un determinado Sprint.

Planning Poker

James Greening presentó la técnica llamada “Planning Poker” o “Poker Estimation”, que intenta evitar el análisis parálisis en la planificación del release de software”. Se basa en el método Wideband Delphi para realizar la estimación de requerimientos (o User Stories) de forma colaborativa en un Equipo. La técnica consiste en que cada integrante del Equipo posee en sus manos una baraja de cartas con los números correspondientes a la sucesión de Fibonacci y se siguen los siguientes pasos:

1. El responsable del negocio presenta una historia de usuario para ser estimada.
2. Todos los participantes proceden a realizar su estimación en forma secreta, sin influenciar al resto del Equipo, poniendo su carta elegida boca abajo sobre la mesa.
3. Una vez que todos los integrantes han estimado, se dan vuelta las cartas y se discuten principalmente los extremos.

[8] **Horas ideales**: la estimación en las horas ideales implica la asunción de que no se hará multitarea, todo lo necesario lo tendrás disponible, no habrá interrupciones.

4. Al finalizar la discusión se levantan las cartas y se vuelve a estimar, esta vez con mayor información que la que se tenía previamente.
5. Las rondas siguen hasta que se logra consenso en el Equipo y luego se continúa desde el punto número uno con una nueva historia de usuario.

Considerando que las personas no saben estimar en términos absolutos y son buenos comparando cosas, y que es generalmente más rápido, dado que se obtiene una mejor dinámica grupal y se emplea mejor el tiempo de análisis de las historias de usuario. La técnica prevé la elección de una historia base, llamada "canónica" que es la que se tomará como referencia para comparar las demás historias, y esa es la que se estima primero, el resto se estima comparándola con la canónica para determinar cuánto más grande o más pequeña es en relación a esta.

La Sabiduría de las Multitudes

Solemos favorecer la opinión de los expertos, pues consideramos que sólo una persona con experiencia y conocimientos suficientes es capaz de emitir juicios correctos en un área o materia en particular. Sin embargo, hay evidencias de que las decisiones tomadas colectivamente por un grupo de personas suelen ser más atinadas que las decisiones tomadas sobre la base del conocimiento de un experto".

La tesis detrás de la Sabiduría de las Multitudes es simple: dadas las circunstancias requeridas, un grupo de personas puede tomar una decisión más acertada que la mejor de las decisiones de la mayoría (si no todos) los integrantes del grupo individualmente.

Para que esto pueda suceder, se recomiendan las siguientes condiciones:

1. *Diversidad de opiniones*: cada persona debería tener información particular aún si es sólo una interpretación excéntrica de los hechos conocidos. El grupo debe tener diversidad de perfiles.
2. *Independencia*: las opiniones de los participantes no deberían estar influenciadas por las opiniones de los que los rodean, con el objetivo de evitar el Pensamiento de Grupo.
3. *Agregación*: El grupo debería tener la capacidad de sumar las opiniones individuales y no simplemente votar por la mejor opción.

Cómo derivar la duración de un Proyecto en ambientes Ágiles

La duración de un proyecto no se "estima", se deriva…. tomando el número total de puntos de historia (story points) de sus historias de usuario (user stories) y dividiéndolo por la velocidad del equipo. Ahora bien, ¿qué es la velocidad del equipo?

La velocidad es una métrica del progreso de un equipo. Esta se calcula sumando el número de puntos de historia asignados a cada historia de usuario como valor de estimación, que el equipo pudo completar durante la iteración.

Para el cálculo de velocidad sólo cuentan los puntos de historia de las historias de usuario que está completas, terminadas; no parcialmente completas.

La velocidad no se estima, se calcula al final de la iteración en función de las historias de usuario que el referente del cliente, en algunas metodologías ágiles se le llama Dueño de Producto, acepta.

La velocidad nos ayuda a determinar un horizonte de planificación apropiado.

La estimación en puntos de historia separa completamente la estimación de tamaño/esfuerzo de la estimación de duración.

Anteriormente, se definió la velocidad como una métrica de progreso de un equipo. ¿A qué se refiere esa expresión?

Esa definición refuerza el enfoque minimalista que la filosofía ágil tiene sobre las métricas, evidenciada en el principio del manifiesto ágil que dice que "la mejor medida de progreso es el software funcionando". El enfoque ágil de las métricas también plantea que les métricas reflejan la realidad de un equipo en un contexto de proyecto específico, y que esta experiencia no puede extrapolarse, para inferir estimaciones de otros productos, en otros proyectos con otros equipos. De hecho, no deja de ser un riesgo derivar la duración de un proyecto considerando la velocidad de un equipo, que puede no ser constante en las diferentes iteraciones.

La derivación de la duración implica la definición de cuantas iteraciones (que son de duración fija), va a necesitar el equipo para completar todos los puntos de historia, lo que finalmente quedará plasmado en el Plan del Release que indica para cada iteración cuantos puntos de historia de quemarán y una definición inicial de que historias de usuario se incluirán en la iteración para cumplir con el objetivo del release.

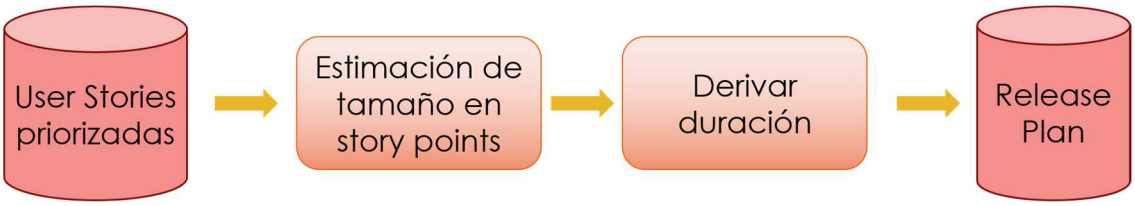


Figura 19: Derivar duración de un proyecto en Ágil

Algunas consideraciones finales sobre las estimaciones Ágiles

Muchas teorías y enfoques convergen en las siguientes características sobre estimaciones en proyectos ágiles:

- No tiene sentido presentar estimaciones certeras al comienzo de un proyecto ya que su probabilidad de ocurrencia es extremadamente baja por el alto nivel de incertidumbre.
- Intentar bajar dicha incertidumbre mediante el análisis puede llevarnos al "Análisis Parálisis". Para evitar esto debemos estimar a alto nivel con un elevado grado de probabilidad, actuar rápidamente, aprender de nuestras acciones y refinar las estimaciones frecuentemente. Este enfoque se conoce también como "Elaboración Progresiva".
- La mejor estimación es la que provee el Equipo que realizará el trabajo. Esta estimación será mucho más realista que la estimación provista por un experto ajeno al Equipo.

Revisar Técnicamente el Software

Las revisiones técnicas surgen a partir de la necesidad de producir software de alta calidad. Algunos grupos de desarrollo creen que la calidad del software es algo en lo que deben preocuparse una vez que se ha generado el código. ¡Error! El Aseguramiento de la calidad del software es una actividad de protección que se aplica a lo largo de todo el proceso de construcción del software.

Es un proceso de revisión cuyo propósito es llegar a detectar lo antes posible, errores o desviaciones en los productos que se van generando a lo largo del desarrollo. Esto se fundamenta en el hecho que el trabajo debe ser revisado; debido a que hay errores que son percibidos más fácilmente por otras personas que por los creadores.

Objetivos de las revisiones técnicas

- Descubrir errores en la función, lógica o implementación de cualquier representación del software.
- Verificar que el software bajo revisión alcanza sus requerimientos.
- Garantizar el uso de estándares predefinidos.
- Conseguir un desarrollo uniforme del software.
- Obtener proyectos que hagan más sencillo los trabajos técnicos (análisis que permitan buenos diseños, diseños que permitan implementaciones sencillas, mejores estrategias de pruebas)

Ventajas de las revisiones técnicas

- Reducción sustancial del costo del software, evitando re-trabajo.
- Gran valor educativo para los participantes.
- Sirve para comunicar la información técnica.
- Fomenta la seguridad y la continuidad.

Directrices para la revisión

- Revisar el producto y no al productor.
- Hacer foco en los problemas no en la forma de resolverlos.
- Indicar los errores con tino, tono constructivo.
- Fijar agenda y mantenerla.
- Enunciar problemas no resueltos.
- Limitar el debate y las impugnaciones.
- Limitar el número de participantes.
- Desarrollar una lista de comprobaciones para cada producto que pueda ser revisado.
- Disponer de recursos y planificación de tiempos.
- Entrenar los participantes.
- Reparar las revisiones anteriores.
- El problema debería ser resuelto por el autor.

Si bien cualquier producto de trabajo puede ser revisado técnicamente. Suele requerirse esta práctica para una muestra de artefactos, si el producto es muy grande y esto podría afectar los compromisos del proyecto. La siguiente tabla muestra ejemplos de artefactos de software que pueden ser revisados y que roles podrían participar de una revisión:

Artefacto de Software	Revisores sugeridos
Arquitectura o Diseño de alto nivel	Arquitecto, analista de requerimientos, diseñador, líder de proyecto, analistas de prueba.
Diseño detallado	Diseñador, arquitecto, programadores, analistas de prueba.
Planes de proyecto	Líder de proyecto, representante de ventas o marketing, líder técnico, representante del área de calidad.
Especificación de requerimientos	Analista de requerimientos, líder de proyecto, arquitecto, diseñador, analistas de prueba, representante de ventas y/o marketing.
Código fuente	Programador, diseñador, analistas de prueba, analista de requerimientos.
Plan de testing	Analistas de prueba, programador, arquitecto, diseñador, representante del área de calidad, analista de requerimientos.

Figura 20: Ejemplo de artefactos y revisores sugeridos

El trabajo técnico **necesita ser revisado** por una simple razón: **errar es humano**. Otra razón por la que son necesarias las revisiones técnicas es que, aunque la gente es buena descubriendo algunos de sus propios errores, algunas clases de errores se le pasan más fácilmente al que los origina que a otras personas.

El proceso de revisión es, por lo tanto, la respuesta a la plegaria de Robert Burns:

¡Qué gran regalo sería poder vernos como nos ven los demás!

Una revisión es una forma de aprovechar la diversidad de un grupo de personas para:

- Señalar la necesidad de mejoras en el producto hecho por una sola persona o un equipo.
- Confirmar las partes del producto en las que no es necesaria o no es deseable una mejora.
- Conseguir un trabajo de mayor calidad maximizando los criterios de corrección y completitud principalmente.

Existen muchos tipos diferentes de revisiones que se pueden llevar adelante como parte de la ingeniería del software. Cada una tiene su lugar. Una reunión informal durante el almuerzo o en un café es una forma de revisión, si se discuten problemas técnicos. Una presentación formal de un diseño de software a una audiencia amplia y variada es una forma de revisión. La más formal de las técnicas se llama *Inspección de Software*, se describirá en forma resumida un poco más adelante en esta sección.

Impacto de los defectos del software en el costo

Dentro del contexto de desarrollo de software, los términos "defecto" y "falla" son sinónimos. Ambos implican un problema de calidad descubierto después de entregar el software a los usuarios finales.

El objetivo primario de las revisiones técnicas es encontrar errores durante el proceso para evitar que se conviertan en defectos después de la entrega del software. El beneficio obvio de estas revisiones es el descubrimiento de errores al principio para que no se propaguen al paso siguiente del proceso de desarrollo del software.

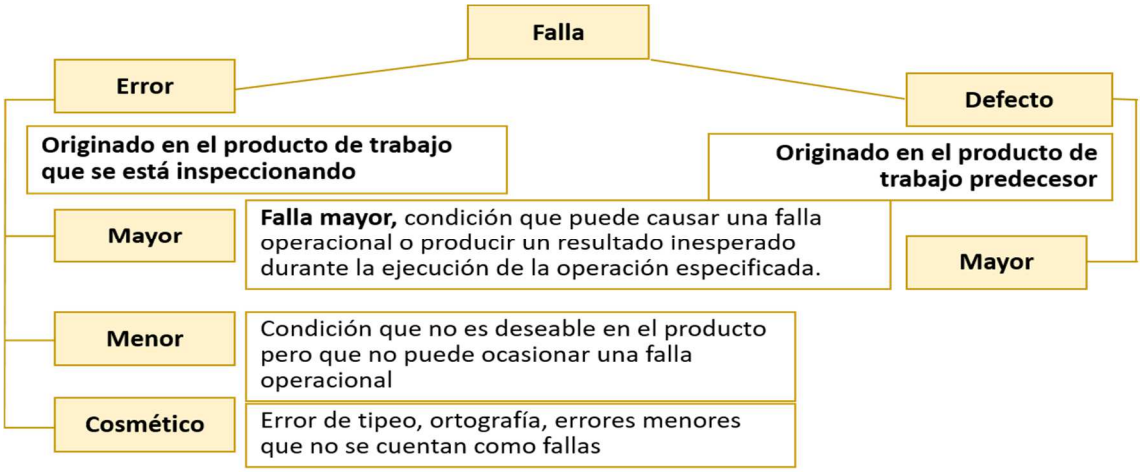


Figura 21: Clasificación de fallas en el software.

Se ha demostrado que las revisiones de software son efectivas en un 75% a la hora de detectar errores.

Con la detección y la eliminación de un gran porcentaje de errores, el proceso de revisión reduce substancialmente el costo de los pasos siguientes en las fases de desarrollo y mantenimiento.

Para ilustrar el impacto sobre el costo de la detección anticipada de errores, supongamos que un error descubierto durante el diseño cuesta corregirlo 1 unidad monetaria. De acuerdo a este costo, el mismo error descubierto justo antes de que comience la prueba costará 6,5 unidades; durante la prueba 15 unidades; y después de la entrega, entre 60 y 100 unidades.

Se puede resumir los beneficios comprobados al aplicar revisiones técnicas en:

- Reducción de los defectos en el uso del software.
- Reducción de los recursos de desarrollo, sobre todo en las etapas de codificación y prueba.
- Reducción en los costos de mantenimiento correctivo.

El proceso de inspección de software

Podemos ver a las inspecciones de software como un repaso detallado y formal del trabajo en progreso. Pequeños grupos de trabajadores (4 o 5) estudian el "producto de trabajo" independientemente y luego se reúnen para examinar el trabajo en detalle.

El "producto de trabajo" puede ser: líneas de código, requerimientos, diseño y/o cualquier otro artefacto. Los productos de trabajo son considerados “en progreso” hasta que la inspección y las correcciones necesarias estén completas.

Grupos de inspección: Se recomienda formar grupos entre 3 y 6 personas [IEEE STD 1028-1988]. Dentro de éste grupo debe incluirse al autor de los productos de trabajo.

Roles involucrados en una inspección de software:

Rol	Responsabilidad
Autor	• Creador o encargado de mantener el producto que va a ser inspeccionado. • Inicia el proceso asignando un moderador y designa junto al moderador el resto de los roles. • Entrega el producto a ser inspeccionado al moderador. • Reporta el tiempo de re trabajo y el nro. total de defectos al moderador.
Moderador	• Planifica y lidera la revisión. • Trabaja junto al autor para seleccionar el resto de los roles. • Entrega el producto a inspeccionar a los inspectores con tiempo (48hs) antes de la reunión. • Coordina la reunión asegurándose que no hay conductas inapropiadas • Hacer seguimiento de los defectos reportados.
Lector	Lee el producto a ser inspeccionado.
Anotador	Registra los hallazgos de la revisión
Inspector	Examina el producto antes de la reunión para encontrar defectos. Registra sus tiempos de preparación.

Figura 22: Roles involucrados en la Inspección de Software.

La inspección consiste en seis pasos [Fagan 1986]:

Planificación: Cuando el autor completa un "producto de trabajo", se forma un grupo de inspección y se designa un moderador. El moderador asegura que el "producto de trabajo" satisfaga el criterio de inspección. Se le asignan diferentes roles a las personas que integran el grupo de inspección, así como la planificación de tiempos y recursos necesario.

Revisión General: Si los inspectores no están familiarizados con el software que se está desarrollando, una vista general es necesaria en éste momento. Este es un paso opcional, pero no menos importante ya que en esta etapa se dará al grupo de inspección introducción y contexto requeridos para actuar durante las inspecciones.

Preparación: Los inspectores se preparan individualmente para la inspección en la reunión, estudiando los productos de trabajo y el material relacionado. Aquí es aconsejable la utilización de listas de chequeo para ayudar a encontrar defectos comunes. El tiempo que pueda llevar esta etapa va a depender de cuan familiarizado esté el inspector con el trabajo que debe analizar.

Reunión: En esta etapa, los inspectores se reúnen para analizar su trabajo individual en forma conjunta. El moderador deberá asegurarse que todos los inspectores están suficientemente preparados. La persona designada como lector presenta el "producto de trabajo", interpretando o parafraseando el texto, mientras que cada participante observa en busca de defectos. Es recomendable que este examen no dure más de 2 horas ya que la atención en busca de defectos va disminuyendo con el tiempo. Al terminar con la reunión, el grupo determina si el producto es aceptado o debe ser re trabajado para una posterior inspección. Los dos resultados principales deben ser: Una lista de defectos a corregir, y un reporte de inspección que describa que es lo que se inspeccionó, quien inspeccionó qué y el número de defectos encontrados.

Re trabajo: El autor corrige todos los defectos encontrados por los inspectores.

Seguimiento: El moderador chequea las correcciones del autor. Si el moderador está satisfecho, la inspección está formalmente completa, y el "producto de trabajo" aceptado.

El proceso de inspección debe ser llevado a cabo por personas que conozcan tanto del dominio específico, del producto de software, así como la tecnología aplicada a las soluciones que serán objeto de la inspección. A partir de éste conocimiento en el equipo de inspección, deberán respetarse las etapas planteadas precedentemente, creando las condiciones necesarias para maximizar la sinergia que se produzca sobre todo en la etapa de "examen".

Puntos Clave para tener en cuenta:

- Revisar al producto… no al productor.
- Fijar una agenda y cumplirla.
- Limitar el debate y las impugnaciones.
- Enunciar las áreas de problemas, pero no tratar resolver cualquier problema que se manifieste.
- Tomar notas escritas.
- Limitar el nro. de participantes e insistir en la preparación por anticipado.
- Desarrollar una lista de revisión.
- Disponer recursos y una agenda.
- Entrenamiento.
- Repasar revisiones anteriores.
- Aprender sobre inspecciones y convencer al proyecto de utilizarlas.
- Determinar en qué parte del proyecto deben ser utilizadas.
- Definir para cada proyecto cómo se realizarán las revisiones técnicas.

Probar el Software - Testing

La prueba de software, más conocida por su nombre en inglés: “testing de software” pertenece a una actividad o etapa del proceso de producción de software denominada Verificación y Validación –usualmente abreviada como V&V.

V&V es el nombre genérico dado a las actividades de comprobación que aseguran que el software respeta su especificación y satisface las necesidades de sus usuarios. El sistema debe ser verificado y validado en cada etapa del proceso de desarrollo utilizando los documentos (descripciones) producidas durante las etapas anteriores. Cómo vimos en la sección anterior no sólo el código debe ser sometido a actividades de V&V sino también todos los artefactos generados durante el desarrollo del software.

Si bien estos términos en su uso cotidiano pueden llegar a ser sinónimos, para la Ingeniería de Software tienen significados diferentes y cada uno tiene una definición precisa.

Validación: ¿estamos construyendo el producto correcto?

Verificación: ¿estamos construyendo el producto correctamente?

En este sentido, la verificación consiste en corroborar que el programa respeta su especificación, mientras que validación significa corroborar que el programa satisface las expectativas del usuario.

El testing es una actividad desarrollada para controlar la calidad del producto, al identificar defectos y problemas. El testing de software consiste en la **verificación dinámica** del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados, en relación con el comportamiento esperado. Es una técnica dinámica en el sentido de que el programa se verifica poniéndolo en ejecución de la forma más parecida posible a como se ejecutará cuando esté en producción.

El programa se prueba ejecutando sólo unos pocos casos de prueba dado que por lo general es física, económica y/o técnicamente imposible ejecutarlo para todos los valores de entrada posibles. De aquí la frase de Dijkstra: “*Si uno de los casos de prueba detecta un error el programa es incorrecto, pero si ninguno de los casos de prueba seleccionados encuentra un error no podemos decir que el programa es correcto (perfecto*)”.

Esos casos de prueba son elegidos siguiendo alguna regla o criterio de selección.

Se determina si un caso de prueba ha detectado un error o no comparando la salida producida con la salida esperada para la entrada correspondiente –la salida esperada debería estar documentada en la especificación del programa.

Las limitaciones antes mencionadas no impiden que el testing se base en técnicas consistentes, sistemáticas y rigurosas (e incluso, como veremos más adelante, formales). Sin embargo, en la práctica industrial, como ocurre con otras áreas de Ingeniería de Software, usualmente se considera sólo una parte mínima de dichas técnicas tornando a una actividad razonablemente eficaz y eficiente en algo útil y de escaso impacto.

Algunos conceptos básicos sobre el testing

Testear un programa significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados.

Se definirán en primer lugar algunos conceptos:

Falta (Failure): cuando un programa funciona mal.

Falla (Fault): existe en el código del programa. Puede provocar una falta.

Error: Acción humana que resulta en software que contiene una falla.

La primera lección a aprender es que no se puede probar que el sistema no tenga una *falta*, sí que esté libre de fallas. El propósito de la prueba es **encontrar fallas**

La prueba es un proceso *destructivo*, tener que indagar sobre lo que hicimos para detectar lo que hicimos mal.

Es conocido que la corrección de una falla provoca fallas adicionales, en consecuencia, si una falla aparece debemos probar todo el software nuevamente.

Principios del Testing de Software

- ✓ **Un programador debería evitar probar su propio código**.
- ✓ **Una unidad de programación no debería probar sus propios desarrollos**.
- ✓ **Examinar el software para probar que no hace lo que se supone que debería hacer.**
- ✓ **Examinar el software para detectar que hace lo que no se supone que debería hacer**.
- ✓ **No planificar el esfuerzo de testing sobre la suposición de que no se van a encontrar defectos**.
- ✓ **El testing es una tarea extremadamente creativa e intelectualmente desafiante**.
- ✓ **Testing temprano, lo más temprano posible**.
- ✓ **La paradoja del pesticida:** Si las pruebas se repiten una y otra vez, con el tiempo el mismo conjunto de casos de prueba ya no encuentran nuevos errores. Para superar esta "paradoja del pesticida", los casos de prueba deben ser examinados y revisadas periódicamente.
- ✓ **El testing es dependiente del contexto**: Las pruebas se realizan de manera diferente en diferentes contextos. Por ejemplo, la seguridad del software será testeada de forma diferente en un sitio de comercio electrónico que en uno donde se comparten fotografías. No todos los sistemas de software llevan el mismo nivel de riesgo y no todos los problemas tienen el mismo impacto cuando se producen.
- ✓ **Falacia sobre la ausencia de errores**: que no encontremos errores no significa que no estén ahí.

Tipos de Pruebas (tests)

Las pruebas (tests) se hacen para verificar y validar tanto requerimientos funcionales como requerimientos no funcionales.

El siguiente es un resumen de varios tipos de tests. Ninguno de ellos es independiente de los otros, cuando se realiza la prueba de un sistema, se usan en combinación.

Tests de Operación: es el más común. El sistema es probado en operación normal. Mide la confiabilidad del sistema y se pueden obtener mediciones estadísticas.

Tests de Escala Completa: ejecutamos el sistema al *máximo*, todos los parámetros enfocan a valores máximos, todos los equipos conectados, usados por muchos usuarios ejecutando casos de uso simultáneamente.

Tests de Performance o de Capacidad: el objetivo de esta prueba es medir la capacidad de procesamiento del sistema. Los valores obtenidos (medidos) son comparados con los requeridos.

Tests de Sobrecarga: cumple la función de determinar cómo se comporta el sistema cuando es *sobrecargado*. No se puede esperar que supere esta prueba, pero sí que no se venga abajo, que no ocurra una catástrofe. Cuántas veces se cayó el sistema es una medida interesante.

Tests Negativos: El sistema es sistemática e intencionalmente usado en forma incorrecta. Este maltrato debe ser planeado para probar casos especiales.

Tests basados en Requerimientos: estos tests son los que pueden mapearse (rastrearse), directamente desde la especificación de requerimientos.

Tests Ergonómicos: son muy importantes si el sistema será usado por gente inexperta. Se prueban cosas como: distribución de opciones de menú, consistencia de nombres entre las diferentes funcionalidades, uso de recursos de interfaz gráfica.

Tests de interfaz de usuario: Consistencia entre las interfaces de las distintas funcionalidades. Si los menús son lógicos y legibles. Si se entienden los mensajes de error.

Tests de Documentación del Usuario: con el estilo y características del anterior, se prueba la documentación del sistema, su claridad, y sobre todo la consistencia con la versión de software asociada.

Niveles de Prueba

Los niveles de prueba utilizados para probar el software son:

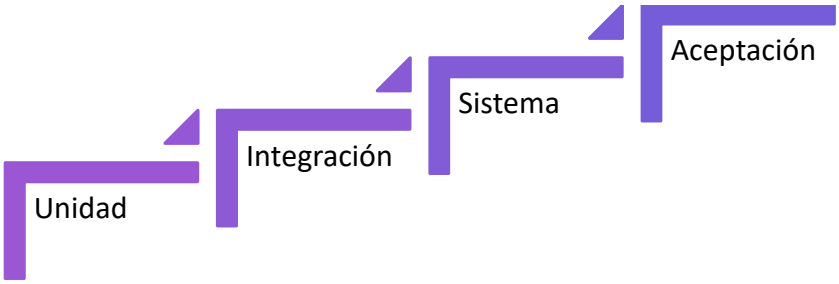


Figura 23: Niveles de Prueba en el Software.

Pruebas de Unidad

Es el nivel de prueba más bajo y normalmente lo hace el mismo desarrollador, principalmente por razones de costo. La prueba de unidad involucra: clases, bloques, paquetes de servicio. En sistemas tradicionales: procedimientos, subrutinas.

Las pruebas de unidad en sistemas orientados a objeto son más complejas; conceptos como la herencia, el polimorfismo, etc., hacen más compleja la prueba.

Pruebas de Integración

Una vez que las unidades han sido certificadas en las pruebas de unidad, estas unidades deberían integrarse en unidades más grandes y finalmente al sistema.

El propósito de las pruebas de integración es determinar si las distintas unidades que han sido desarrolladas trabajan apropiadamente, juntas. Aquí se incluyen pruebas de paquetes de servicio, de subsistemas y el sistema completo. Consecuentemente no hay una sola prueba de integración en un desarrollo, por el contrario, se realizan varias a distintos niveles.

Estas pruebas son necesarias porque:

- Al combinar las unidades pueden aparecer nuevas fallas.
- La combinación aumenta exponencialmente el número de caminos posibles.

Por lo tanto, hay fallas que no podrían detectarse de otra forma. Normalmente las pruebas de integración (desde el nivel de subsistema para arriba) se realizan con un equipo de prueba. Aquí la documentación es más formal que en las pruebas de unidad. Usualmente a las pruebas, el equipo las realiza en un entorno similar al que se el sistema se ejecutará cuando esté en operación.

Prueba de Sistema

Una vez que se han probado todas las integraciones, se probará el producto o la versión del producto, completa. Es la prueba realizada cuando una aplicación está funcionando como un

todo (Prueba de la construcción Final). Trata de determinar si el producto en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc.)

El entorno de prueba debe corresponder al entorno de producción tanto como sea posible para reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente y que no se encontraron en las pruebas. Deben investigar tanto requerimientos funcionales y no funcionales del sistema.

Prueba de Aceptación de Usuario

Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades. La meta en las pruebas de aceptación es establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema. Encontrar defectos no es el foco principal en las pruebas de aceptación. Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta).

El proceso de Prueba

Es importante planear la prueba; la prueba no es algo que debe hacerse en forma improvisada. El proceso de prueba es un proceso que en gran medida corre en paralelo con otros procesos. El proceso de prueba contiene el conjunto de actividades que se describen en la siguiente figura:

Figura 24: Proceso de Prueba

Planificación de la prueba

La actividad de prueba comienza pronto en el proceso de desarrollo del software. La planificación puede comenzar cuando comenzamos el desarrollo, en general durante los requerimientos, pero no podemos ejecutar nada hasta no comenzar la construcción.

Los lineamientos de la prueba se establecen con anticipación, determinando el método y nivel de ambición, se crean las bases de la prueba. Debería determinarse si las pruebas se harán manual o automáticamente. Hacer una estimación de recursos que se requerirán, y estas decisiones deberían reflejarse en un Plan de Prueba separado o como parte del Plan de Proyecto.

Se estudia si existen programas de prueba y datos que puedan usarse, si deberían modificarse o crearse nuevos. Usando estos lineamientos como base podemos determinar qué grado de cobertura tendrían los tests.

El plan no debe controlar los detalles de la prueba, sólo servir como base para las actividades de la prueba.

Un registro de la prueba se debe mantener durante el proceso de prueba completo. El registro debería conectarse a la versión del sistema. El propósito del registro es mantener una breve historia de las actividades de prueba. El registro es archivado al finalizar las pruebas y sirve de base para el refinamiento del proceso de prueba y para la planificación de nuevos tests.

Diseño

Cuando identificamos lo que debería probarse, se pueden estimar también los recursos requeridos. Es una estimación más detallada que la hecha anteriormente, y actúa como un principal lineamiento para la especificación y ejecución de la prueba.

Esto requiere la configuración y determinación del equipamiento que será requerido para la prueba, para que estén en condiciones en el momento que se la requiera.

Cuando los recursos de la prueba son restringidos, cada caso de prueba debe maximizar la probabilidad estadística de detección de fallas. Se debería encontrar las fallas mayores primero.

Cuando se identifica cuales pruebas se harán, se especifican a *nivel funcional*, donde describiremos la prueba y su propósito de manera general, y en un *nivel detallado*, donde describiremos exactamente cómo será ejecutado. La última parte incluye una descripción procedural completa de cada paso en la prueba.

El propósito de la especificación de la prueba es dar a las personas que no están familiarizadas con la prueba, o aún con el sistema, instrucciones detalladas para correr los casos de prueba.

Cada caso de prueba debe documentarse, para facilitar el reuso. Deberían especificarse condiciones de prueba tales como: hardware, software, equipamiento de prueba. Debe indicarse también como se debe ejecutar la prueba, en qué orden, salidas esperadas y criterios para aprobar el test.

Cuando se escriben los tests de especificación, también se preparan los reportes requeridos para informar los resultados de la prueba. El esqueleto de los reportes se prepara con anticipación.

Las pruebas ayudan a detectar faltas, si se encuentran faltas, pueden corregirse a nivel de diseño no solamente en el código.

Es interesante enfocar las pruebas asumiendo que el sistema tiene fallas, y poder determinar cuántas horas hombre son necesarias para poder detectar nuevas fallas.

Ejecución de las Pruebas

Cuando ejecutamos las pruebas usamos la especificación de pruebas y los reportes de prueba preparados.

La estrategia es probar lo que más se pueda en paralelo, aunque sea difícil. Las pruebas se realizan en forma manual o automatizada, según se haya especificado. Las especificaciones

indican el resultado esperado. Si alguna de las pruebas falla se registran todos los detalles que se tenga disponibles sobre los defectos encontrados.

Evaluación y Reportes

Al finalizar la prueba se analizan los resultados. Si está aprobado o no. Este análisis resulta en reportes de prueba. Los reportes contienen en forma resumida el resultado individual de cada prueba y uno final, los recursos gastados y si el test está aprobado a no. Si se descubrieron cuellos de botella también deben registrarse y mostrarse.

Si se detectan fallas cuando se hizo la prueba, hay que identificar la razón de la falla encontrada. La falla puede ser debido al sistema, puede haber varias causas:

¿Se ha ejecutado el test correctamente? Hay alguna falla en los datos o programas de prueba.

La falla es causada por los bancos de prueba. Si la falla no es atribuible a la prueba se debe probar nuevamente.

¿Cuánto testing es suficiente?

Responder está pregunta es uno de los aspectos más difícil de probar software. El testing exhaustivo es imposible. Decidir cuánto testing es suficiente depende de:

- Evaluación del nivel de riesgo.
- Costos asociados al proyecto.

Usamos los riesgos para determinar que probar primero, a qué dedicarle más esfuerzo de prueba y que no probar (al menos por ahora). La siguiente figura ejemplifica la imposibilidad en términos de costo beneficio de realizar una prueba exhaustiva de un producto de software. En este ejemplo, para probar en forma exhaustiva un sistema con 20 ventanas; asumiendo que cada prueba pudiera hacerse en 10 minutos, un tiempo considerablemente optimista, serían requeridos 13, 7 años.

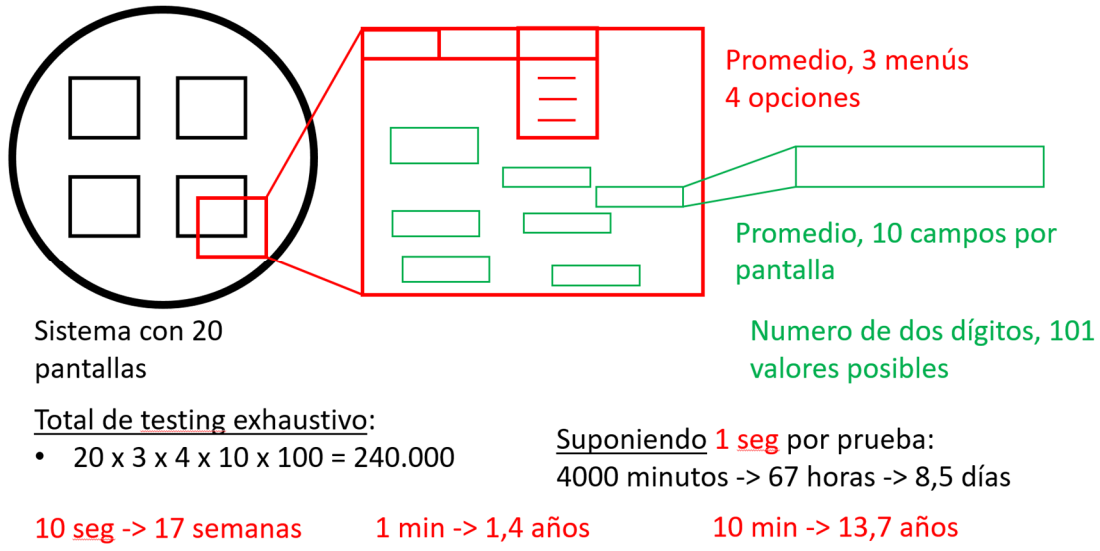


Figura 25: ¿Cuánto testing es suficiente?

El ejemplo anterior muestra que, dada la imposibilidad de probar todas las opciones posibles en un producto de software, el desafío radica en encontrar un conjunto de casos de prueba que nos permitan abarcar la mayor cantidad de situaciones posibles, que deben ser priorizadas en función de las necesidades y expectativas de los usuarios y de los recursos disponibles.

¿Cuándo dejar de probar?

Responder a esta pregunta es imposible, ya que en la práctica es imposible estimar cuando se alcanzará una determinada densidad de defectos, que justifique la finalización del proceso de prueba. Esto se basa en la necesidad de alcanzar la convergencia. Esta convergencia se logra a partir del punto en el cual, aumentar la cantidad de pruebas no incrementa la cantidad de defectos encontrados.

Algunas estrategias utilizadas:

- Dejar de probar cuando el producto pasa exitosamente el conjunto de pruebas diseñado ("No Failure"). + cobertura estructural.
- "Good Enough" (Bueno Suficiente): cierta cantidad de fallas no críticas es aceptable (umbral de fallas no críticas por unidad de testing) (Microsoft).
- Defectos detectados es similar a la cantidad de defectos estimados.

Si construimos un sistema y, al hacerlo, encontramos y corregimos defectos, no quiere decir que lo convierte en un buen sistema. Encontrar y corregir defectos no sirve de nada si el sistema integrado no se puede utilizar y no satisface las necesidades de los usuarios y sus expectativas.

Las personas y organizaciones que compran y utilizan software como ayuda en su día a día no están interesadas en los defectos o el número de defectos, salvo que sean directamente afectados por la inestabilidad del software.

La gente que usa software está más interesada en que la aplicación los apoye en la realización de sus tareas de manera eficiente y eficaz.

Por eso revisiones en las primeras etapas (requisitos y diseño) son una parte importante de las pruebas y si los verdaderos usuarios del sistema no han estado involucrados en el proceso de prueba en algún momento, entonces es probable que no obtengan lo que realmente quieren.

Administrar la Configuración del Software

La necesidad de gestionar la configuración surge del hecho que el software evoluciona constantemente.

El desarrollo del software siempre es progresivo y dadas las características de maleabilidad propias del software, es necesario mantener organizados y controlados los cambios que se realizarán al software a lo largo del tiempo.

El software: un blanco móvil

Los cambios en el software tienen su origen en:

- Cambios del negocio y nuevos requerimientos.
- Soporte de cambios de productos asociados.
- Reorganización de las prioridades de la empresa por crecimiento.
- Cambios en el presupuesto.
- Defectos encontrados, a corregir.
- Oportunidades de mejora.

Un producto de software **tiene integridad cuando**:

- satisface las necesidades del usuario;
- puede ser fácil y completamente rastreado durante su ciclo de vida;
- satisface criterios de performance y cumple con sus expectativas de costo.

El propósito de la disciplina de Administración de Configuración de Software *es establecer y mantener la **integridad de los productos** de software a lo largo de su ciclo de vida*.

Involucra para la configuración:

- Identificarla en un momento dado,
- controlar sistemáticamente sus cambios y
- mantener su integridad y origen.

La Administración de Configuración de Software es una disciplina protectora, transversal a todo el proyecto, que implica:

- Identificar y documentar las características funcionales y físicas de los ítems de configuración. ·
- Controlar los cambios a tales características.
- Reportar el proceso de tales cambios y su estado de implantación.

El objetivo es maximizar la productividad minimizando los errores. Dado a que el cambio puede ocurrir en cualquier momento, las actividades de la administración de configuración de software son desarrolladas para identificar el cambio, controlar el cambio, asegurar que el cambio está siendo apropiadamente implantado, e informar del cambio a aquellos que les interesa.

Algunos conceptos relacionados con la Administración de Configuración del Software

Configuración

Se define como configuración del software al conjunto de ítems de configuración con su correspondiente versión en un momento determinado.

Una configuración es el conjunto de todos los componentes fuentes que son compilados en un ejecutable consistente, más todos los componentes, documentos e información de su estructura que definen una versión determinada del producto en un momento de tiempo.

Una configuración cambia porque se añaden, retiran o modifican elementos. También hay que contemplar la posibilidad de que los mismos elementos se reorganicen de forma diferente, sin que cambien individualmente.

El control de configuración se refiere a la evolución de un conjunto de ítems de configuración.

La evolución del sistema consiste en: añadir, suprimir, modificar ítems de configuración; o bien, reorganizar la estructura. Se llama:

- Evolución temporal: *revisiones* º Son cambios a lo largo del tiempo.
- Evolución espacial: *variantes* º Son versiones (configuraciones) simultáneas.

Repositorio

Un repositorio es un espacio de almacenamiento que contiene los ítems de configuración. Ayuda a mantener la historia de cada ítem de configuración, con sus atributos y relaciones. Pueden estar conformado por una o varias bases de datos. El repositorio permite centralizar el almacenamiento de los ítems de configuración (IC) de un mismo sistema o producto de software, incluyendo las distintas versiones de cada IC.

El repositorio permite ahorrar espacio de almacenamiento, evitando guardar por duplicado elementos comunes a varias versiones o configuraciones. Para conseguir ese ahorro hay que disponer de un sistema de representación especializado para las versiones.

El repositorio facilita el almacenamiento de información de la evolución del sistema (historia), no sólo de los IC en sí. La siguiente figura muestra un modelo típico de trabajo con un repositorio.

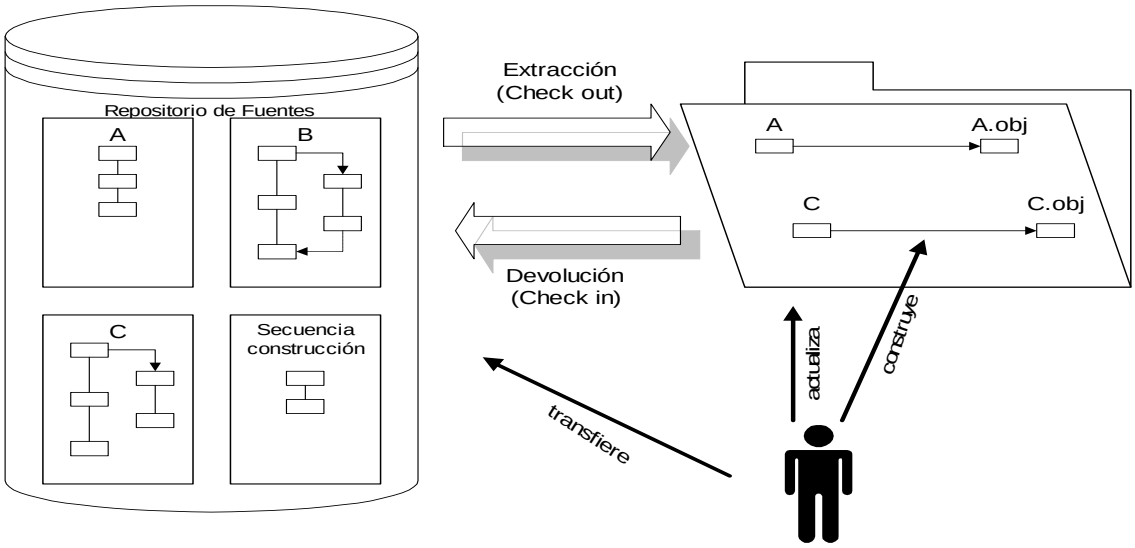


Figura 26: Modelo Check in – Check out de un repositorio

Puede haber dos tipos base de repositorios:

Repositorio Centralizado

- Un servidor contiene todos los archivos con sus versiones.
- Los administradores tienen mayor control sobre el repositorio.
- Falla el servidor y nadie puede trabajar.

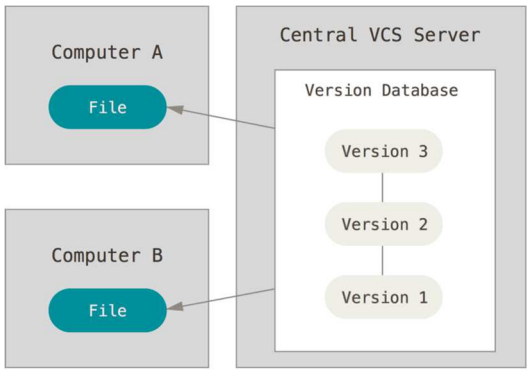


Figura 27: Repositorio Centralizado

Repositorio Distribuido

- Cada cliente tiene una copia exactamente igual del repositorio completo.
- Si un servidor falla sólo es cuestión de "copiar y pegar".
- Posibilita otras formas de trabajo no disponibles en el modelo centralizado.

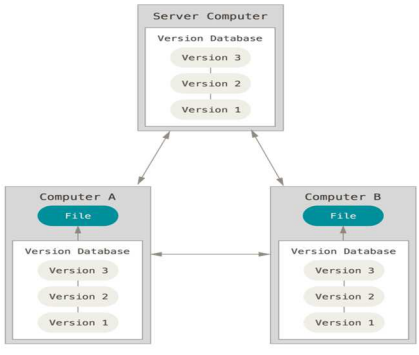


Figura 28: Repositorio Distribuido

Ítem de Configuración

Se llama **ítem de configuración** (IC) a todos y cada uno de los artefactos que forman parte del producto o del proyecto, que pueden sufrir cambios o necesitan ser compartidos entre los miembros del equipo y sobre los cuales necesitamos conocer su estado y evolución.

Ejemplos de ítems de configuración pueden ser: documentos de requerimientos, documentos de diseño, código fuente, código ejecutable, plan de proyecto, etc.

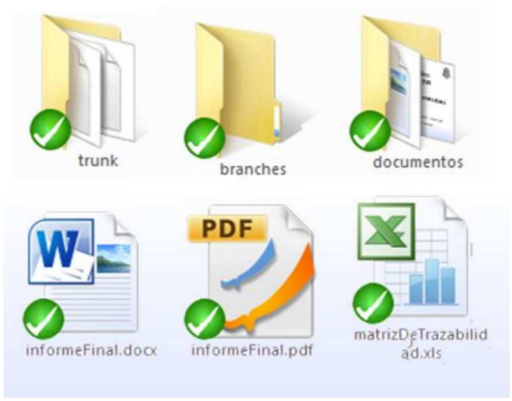


Figura 29: Ejemplo de ítem de configuración

Cambio

El cambio en este contexto se define como el paso de una versión de la línea base a la siguiente. Puede incluir modificaciones del contenido de algún componente, o modificaciones de la estructura del sistema, añadiendo, eliminando y/o reorganizando componentes.

Versión

Una versión se define, desde el punto de vista de la evolución, como la forma particular de un artefacto en un instante o contexto dado.

También hay que contemplar la posibilidad de que coexistan versiones alternativas de un mismo artefacto un instante dado. Es necesario disponer de un método para identificar unívocamente las diferentes versiones de manera sistemática u organizada.

El control de versiones se refiere a la evolución de un único ítem de configuración (IC), o de cada IC por separado.

La evolución puede representarse gráficamente en forma de grafo, en el que los nodos son las versiones y los arcos corresponden a la creación de una nueva versión a partir de otra ya existente. El siguiente gráfico muestra las revisiones sucesivas de un componente, que dan lugar a una simple secuencia lineal.

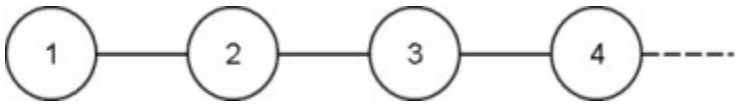


Figura 30: Ejemplo de evolución lineal de un ítem de configuración

Esta forma de evolución no presenta problemas desde el punto de vista de organización del repositorio. Las versiones se pueden designar simplemente mediante números correlativos, como en la figura.

Variante

Una variante es una versión de un ítem de configuración (o de la configuración) que evoluciona por separado. Las variantes representan configuraciones alternativas. Un producto de software puede adoptar distintas formas (configuraciones) dependiendo del lugar donde se instale. Por ejemplo, dependiendo de la plataforma (máquina + S.O.) que la soporta, o de las funciones opcionales que haya de realizar o no.

Cuando hay variantes, es decir, cuando existen simultáneamente varias versiones de un ítem de configuración, el grafo de evolución ya no es una secuencia lineal, sino que adopta la forma de un árbol. Si queremos seguir numerando las versiones se necesitará ahora una numeración a dos niveles. El primer número designa la variante (línea de evolución), y el segundo la versión particular (revisión) a lo largo de dicha variante, como podemos ver en la siguiente figura.

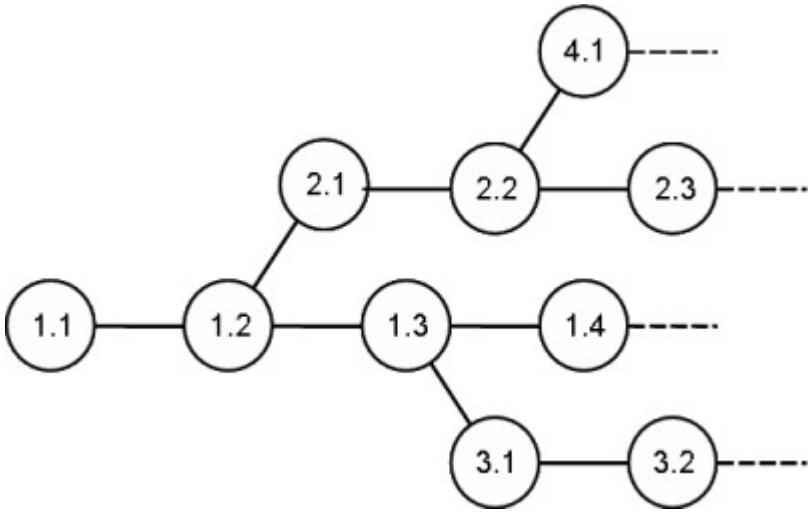


Figura 31: Variante de un ítem de configuración

Ejemplo de evolución de una configuración

Se presenta un ejemplo de evolución simple (secuencia temporal lineal). Las revisiones del conjunto se han numerado correlativamente (Rev.1, Rev.2, ...). Cada configuración contiene una colección de elementos, no siempre los mismos. Pueden crearse o eliminarse elementos entre una revisión y la siguiente. Los cambios individuales de un componente se indican con flechas. Los componentes que se mantienen sin cambios se marcan con dos líneas paralelas (como el signo = en vertical).

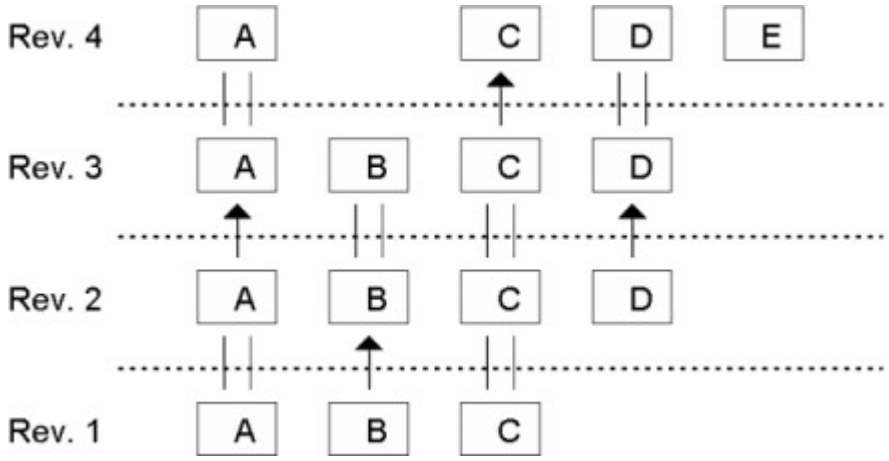


Figura 32: Ejemplo de evolución de una Configuración

Línea Base

Una **línea** base es una configuración que ha sido revisada formalmente y sobre la que se ha llegado a un acuerdo. Sirve como base para desarrollos posteriores y puede cambiarse sólo a través de un procedimiento formal de control de cambios.

El propósito principal de la creación y administración de líneas es la posibilidad de ir atrás en el tiempo y recuperar una versión de un producto en un momento dado del proyecto

Una definición más formal, dada por el Estándar IEEE 610.12-1990, dice:

"Línea Base es una especificación o producto que se ha revisado formalmente y sobre los que se ha llegado a un acuerdo, y que de ahí en adelante sirve como base para un desarrollo posterior y que puede cambiarse solamente a través de procedimientos formales de control de cambios"

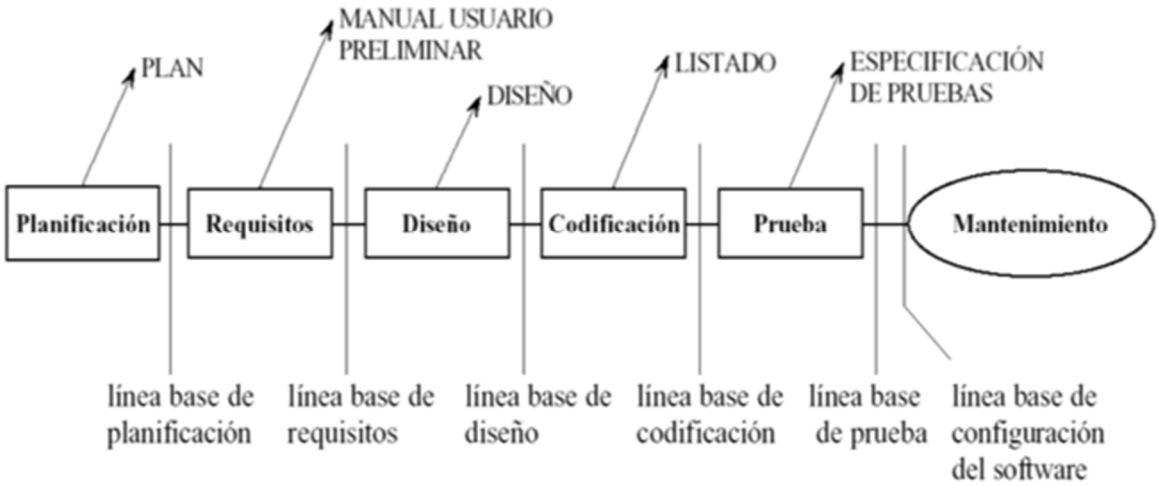


Figura 33: Representación de Líneas Base

Las Líneas Base pueden ser:

- **De especificación:** es decir contienen sólo documentación (Requerimientos, diseño, planes)
- **Operacional**: es decir que contiene una versión del producto que han pasado por un control de calidad previamente definido.

Comité de Control de Cambios

El Comité de Control de Cambios es la autoridad responsable de la aprobación de la incorporación de cambios a una Línea Base. Y posteriormente debe verificar que los cambios implementados sean los que fueron autorizados.

El Comité de Control de Cambios debe tener definido el procedimiento que se utilizará para realizar esta tarea, de manera de garantizar que todos los interesados que deban ser informados, se enteren de la ocurrencia de un cambio.

Está formado por representantes de todas las áreas que se vean afectadas por el cambio propuesto, ejemplos de roles de un equipo de desarrollo que pueden formar parte de un Comité de Control de Cambios:

- Líder de Proyecto
- Arquitecto
- Analista Funcional
- Gestor de Configuración de Software
- Desarrolladores
- Analistas de Prueba
- Representante del Cliente

Actividades Fundamentales de la Administración de Configuración de Software

El siguiente esquema presenta las 4 actividades principales que conforman la disciplina de Administración de Configuración de Software (SCM: Software Configuration Management).

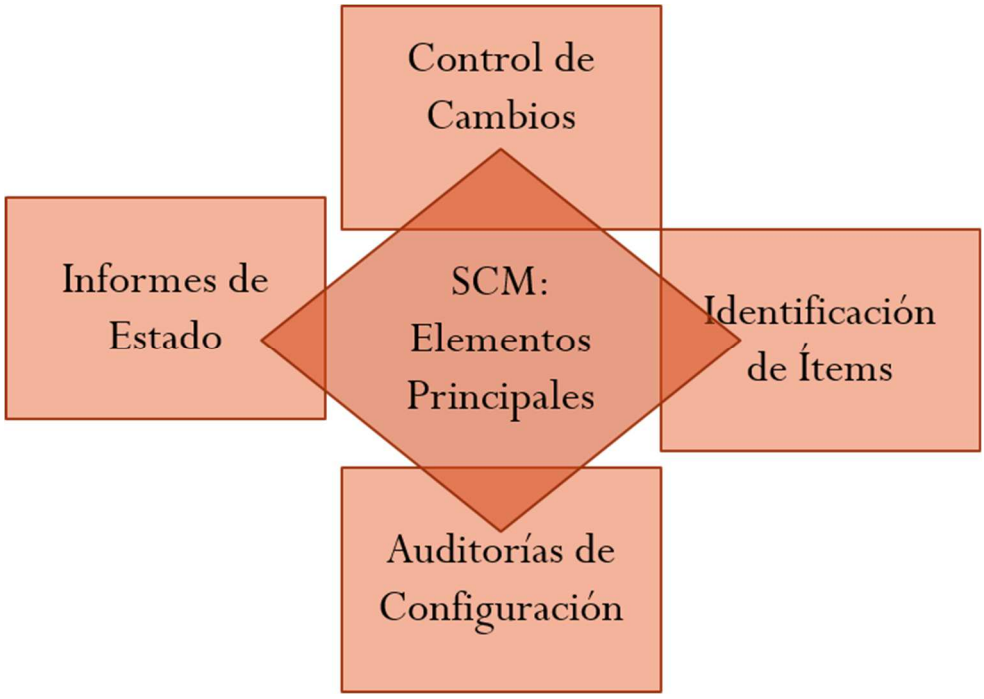


Figura 34: Actividades Principales de la Administración de Configuración de Software

Identificación de ítems

Se trata de establecer estándares de documentación y un esquema de identificación unívoca para los ítems de configuración que se administrarán.

Control de cambios

La ingeniería de software recomienda realizar el desarrollo de manera disciplinada. Las herramientas de control de versiones no garantizan un desarrollo razonable si cualquier miembro del equipo puede realizar los cambios que quiera e integrarlos en el repositorio sin ningún tipo de control.

El control de cambios consiste en la evaluación y registro de todos los cambios que se hagan a la configuración del software. Está relacionado con la naturaleza evolutiva del desarrollo de software; creando por medio de cambios sucesivos realizados de una manera disciplinada. La evolución de un producto de software, puede verse como la evolución de sus líneas base.

Auditorías de configuración

Sirven, junto con las revisiones técnicas para garantizar que el cambio se ha implementado correctamente

Hay dos tipos de auditorías a la configuración del software

- **Auditoría de configuración física**
  Asegura que lo que está indicado para cada IC en la línea base o en la actualización se ha alcanzado realmente. Evaluación independiente de los IC para verificar que el software y su documentación son internamente consistentes y están listos para ser entregados al cliente.
- **Auditoría configuración funcional**
  Evaluación independiente de los productos de software, verificando que la funcionalidad y performance reales de cada ítem de configuración sean consistentes con la especificación de requerimientos de software.

Ambos tipos de auditoría se hacen a una línea base específica del producto de software en un momento de tiempo determinado. Uno de los aspectos principales que se controla en estas auditorías es la trazabilidad del producto de software, razón por la cual la Matriz de Trazabilidad es una herramienta muy útil al momento de auditar, tal como se representa en el siguiente esquema.

La trazabilidad plantea la incorporación de vínculos entre los ítems de configuración que nos permiten detectar si todos los requerimientos que fueron comprometidos están siendo satisfechos, el estado de avance de cada requerimiento, como así también vínculos hacia el origen y la razón de la incorporación de esos requerimientos en el producto de software.

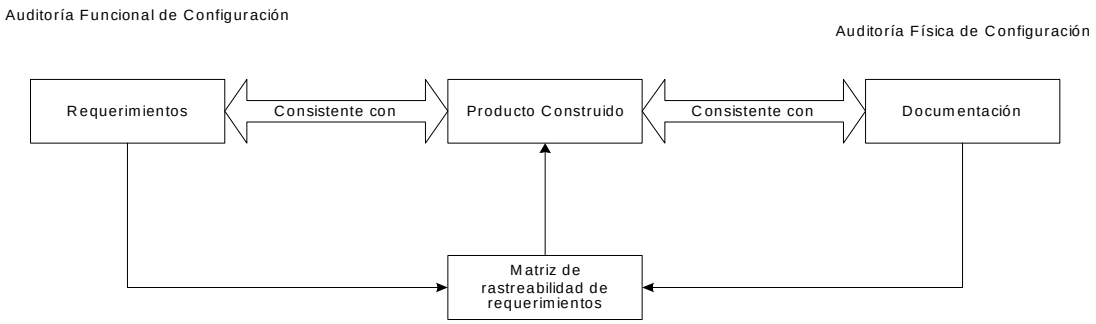


Figura 35: Tipos de Auditorías de Configuración

Generación de informes

Se ocupa de mantener los registros de la evolución del sistema. Maneja mucha información y salidas por lo que se suele implementar dentro de procesos automáticos, cuyo soporte depende de la herramienta de administración de configuraciones que se utilice.

Algunos reportes que se pueden obtener son:

- Momento de incorporación o actualización de una línea base.
- Estado de cada cambio propuesto.

- Momento en que se incorpora un cambio de configuración de software.
- Estado de la documentación administrativa o técnica.
- Deficiencias detectadas durante la auditoría de configuración.
- Información descriptiva de cada cambio propuesto.
- Reportes de rastreabilidad de todos los cambios realizados a las líneas base durante el ciclo de vida.

Plan de Gestión de Configuración

Como todas las actividades vinculadas al desarrollo de software, las actividades vinculadas con la administración de configuración de software también deben planificarse. En el caso particular de estas actividades es deseable que la planificación se haga lo más tempranamente posible.

El Plan de Administración de Configuración debería incluir al menos los siguientes ítems:

- Definición de los ítems de configuración (IC) que se administrarán.
- Reglas de nombrado de los IC.
- Herramientas a utilizar para la administración de configuración.
- Responsabilidades e integrantes del Comité de Control de Cambios.
- Procedimiento formal de cambios.
- Procesos de Auditoría.

Algunas herramientas para Administración de Configuración de Software

- MicroFocus - PVCS Dimensions
- Platinum - CCC/Harvest
- Rational – ClearCase
- Microsoft – Team Fundation
- Accurev – Accurev
- Borland – StarTeam

Herramientas de software libre para control de configuración:

- TortoiseSVN
- Subversion
- Git

Para terminar, algunos Tips relacionados con la Administración de Configuración de Software

- Hacer de la Administración de Configuración de Software (ACS), el trabajo de todos.
- Crear un ambiente y un proceso de ingeniería que permita la Administración de Configuración.
- Definir y documentar el proceso de Administración de Configuración/Ingeniería, luego seleccionar la/las herramientas que le den soporte al proceso.
- El personal de ACS debe contar con Individuos con conocimientos técnicos para dar soporte al desarrollo y mantenimiento del producto.
- Los procedimientos y el Plan de ACS deben realizarse en las etapas iniciales del proyecto.

Metodologías Ágiles

En febrero de 2001 se reunieron en Utah (EEUU) un grupo de diecisiete profesionales reconocidos del desarrollo de software, representantes de nuevas metodologías y críticos de los modelos en procesos predictivos de control; convocados por Kent Beck, para discutir sobre el desarrollo de software con el objetivo de determinar los valores y principios que les permitirían a los equipos desarrollar software de forma más rápida y responder mejor a los cambios que pudieran surgir a lo largo de un proyecto de desarrollo. Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, basado en procesos de control predictivos; caracterizados por la rigidez y dominados por la documentación.

En esta reunión, se creó la Agile Alliance[9], una organización sin fines de lucro cuyo objetivo es el de promover los valores y principios de la filosofía ágil y ayudar a las organizaciones en su adopción. La piedra angular del movimiento ágil es conocida como Manifiesto Ágil (Agile Manifesto[10]) y está apoyada con la firma de más de 100 referentes de la industria de software mundial.

Manifiesto Ágil

El Manifiesto Ágil se compone de 4 valores y 12 principios.

Valores

Los valores del manifiesto ágil destacan y reconocen la importancia de los elementos de la derecha, no obstante, privilegian aún más los elementos de la izquierda de cada declaración de valor.

Valorar a las personas y las interacciones entre ellas por sobre los procesos y las herramientas

Las personas son el principal factor de éxito de un proyecto de software. Es más importante construir un buen equipo que construir el contexto. Muchas veces se comete el error de construir primero el entorno de trabajo y esperar que el equipo se adapte automáticamente. Por el contrario, se propone crear el equipo y que éste construya su propio entorno y procesos en base a sus necesidades.

Valorar el software funcionando por sobre la documentación detallada

La regla a seguir es "no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante". Estos documentos deben ser cortos y centrarse en lo esencial. La documentación (diseño, especificación técnica de un sistema), no es más que un resultado intermedio y su finalidad no es dar valor en forma directa al usuario o cliente del

[9] http://www.agilealliance.org
[10] http://www.agilemanifesto.org

proyecto. Medir avance en función de resultados intermedios se convierte en una simple "ilusión de progreso".

Valorar la colaboración con el cliente por sobre la negociación de contratos

Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta mutua colaboración será la que dicte la marcha del proyecto y asegure su éxito. La expectativa de este valor es que el cliente asuma una participación protagónica en el producto a desarrollar, y que se haga cargo de la definición y priorización de las características que el producto deberá satisfacer. Esto se privilegia frente a la alternativa de una negociación donde el cliente y el equipo de desarrollo pareciera que pertenecen a diferentes bandos.

Valorar la respuesta a los cambios por sobre el seguimiento estricto de los planes

La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también su éxito o fracaso. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta. El equipo debe tener una actitud de aceptación de los cambios, en lugar de una actitud de cumplimiento estricto de los planes fijados completamente al inicio del proyecto, cuando las condiciones existentes y la información disponible pueden ser muy diferentes.

Principios

Los valores anteriores son los pilares sobre los cuales se construyen los doce principios del Manifiesto Ágil. De estos doce principios, los dos primeros son generales y resumen gran parte del espíritu ágil del desarrollo de software, mientras que los siguientes son más específicos y orientados al proceso o al equipo de desarrollo:

1. Nuestra mayor prioridad es satisfacer al cliente a través de entregas tempranas y frecuentes de software con valor.
2. Aceptar el cambio incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan los cambios para darle al cliente ventajas competitivas.
3. Entregar software funcionando en forma *frecuente*, desde un par de semanas a un par de meses, prefiriendo el periodo de tiempo más corto.
4. Expertos del negocio y desarrolladores deben trabajar juntos diariamente durante la ejecución del proyecto.
5. Construir proyectos en torno a personas motivadas, generándoles el ambiente necesario, atendiendo sus necesidades y confiando en que ellos van a poder hacer el trabajo.
6. La manera más eficiente y efectiva de compartir la información dentro de un equipo de desarrollo es la conversación cara a cara.
7. El *software funcionando* es la principal métrica de progreso.
8. Los procesos ágiles promueven el desarrollo sostenible. Los sponsors, desarrolladores y usuarios deben poder mantener un ritmo constante indefinidamente.
9. La atención continua a la excelencia técnica y buenos diseños incrementan la agilidad.

10. La simplicidad –el arte de maximizar la cantidad de trabajo no hecho- es esencial.
11. Las mejores arquitecturas, requerimientos y diseños emergen de equipos auto-organizados.
12. A intervalos regulares, el equipo reflexiona acerca de cómo convertirse en más efectivos, luego mejora y ajusta su comportamiento adecuadamente.

A continuación, se describen de manera general las siguientes metodologías: Scrum, XP (Extreme Programming, Programación Extrema) y TDD (Test Driven Development, Desarrollo Guiado por Pruebas)

SCRUM®

Scrum[11] se basa en la teoría de control de procesos empírica o empirismo. El empirismo, tal como se mencionó en secciones anteriores, asegura que el conocimiento procede de la experiencia *y* de tomar decisiones basándose en lo que se conoce. Scrum emplea un ciclo de vida iterativo e incremental para optimizar la predictibilidad y el control del riesgo.

Tres pilares soportan toda la implementación del control de procesos empírico: transparencia, inspección y adaptación.

Transparencia

Los aspectos significativos del proceso deben ser visibles para aquellos que son responsables del resultado. La transparencia requiere que dichos aspectos sean definidos por un estándar, de tal modo que los observadores compartan un entendimiento común de lo que se está viendo.

Inspección

Los usuarios de Scrum deben inspeccionar frecuentemente los artefactos de Scrum y el progreso hacia un objetivo, para detectar variaciones. Su inspección no debe ser tan frecuente como para que interfiera en el trabajo. Las inspecciones son más beneficiosas cuando se realizan de forma diligente por inspectores expertos, en el mismo lugar de trabajo.

Adaptación

Si un inspector determina que uno o más aspectos de un proceso se desvían de límites aceptables, y que el producto resultante no será aceptable, el proceso o el material que está siendo procesado deben ser ajustados. Dicho ajuste debe realizarse cuanto antes para minimizar desviaciones mayores.

La siguiente figura presenta la representación del framework de Scrum:

[11] **SCRUM**: toda la sección que describe el framework de Scrum se ha desarrollado tomando como referencia la Guía de Scrum, desarrollado por Ken Schwaber y Jeff Sutherland, sus creadores; liberada en julio de 2013.

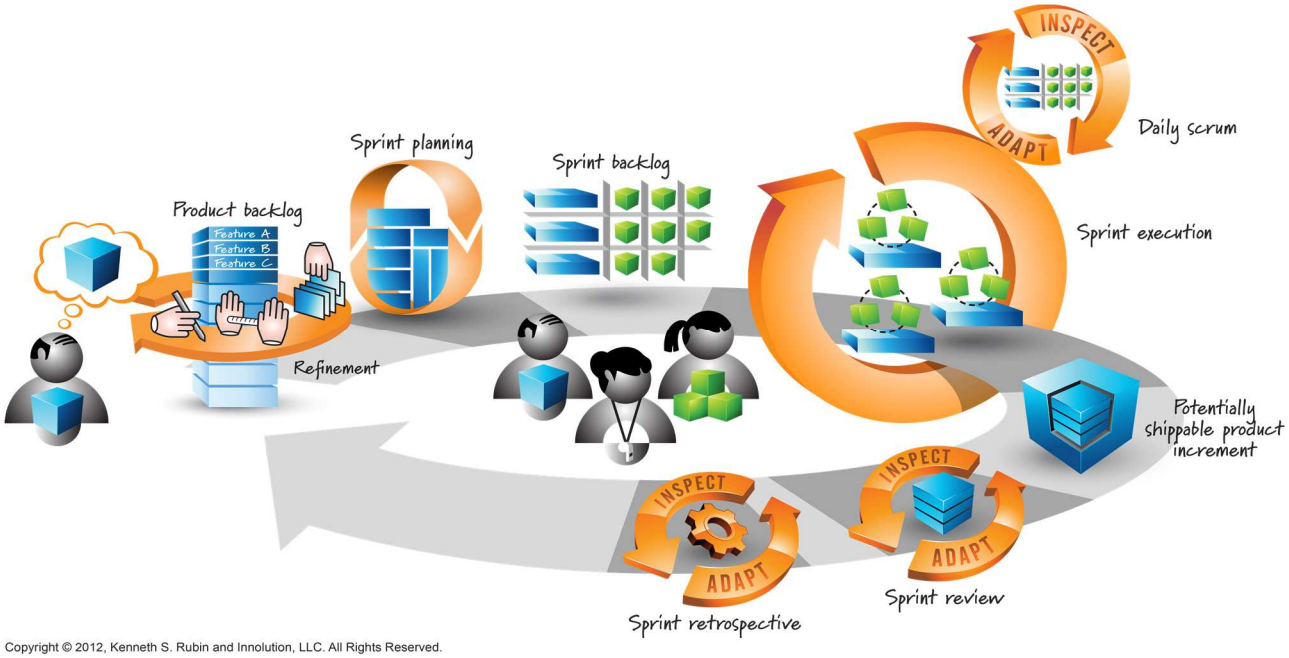


Figura 36: Framework de SCRUM

Roles de Scrum

El Equipo Scrum (*Scrum Team*)

El Equipo Scrum consiste en un **Dueño de Producto** (*Product Owner*), el **Equipo de Desarrollo** (*Development Team*) y un **Scrum Master**. Los Equipos Scrum son auto-organizados y multifuncionales. Los equipos auto-organizados eligen la mejor forma de llevar a cabo su trabajo y no son dirigidos por personas externas al equipo. Los equipos multifuncionales tienen todas las competencias necesarias para llevar a cabo el trabajo sin depender de otras personas que no son parte del equipo. El modelo de equipo en Scrum está diseñado para optimizar la flexibilidad, la creatividad y la productividad.

- El Dueño de Producto (Product Owner)

  El Dueño de Producto es el responsable de maximizar el valor del producto y del trabajo del Equipo de Desarrollo. Cómo se lleva a cabo esto podría variar ampliamente entre distintas organizaciones, Equipos Scrum e individuos.

  El Dueño de Producto es la única persona responsable de gestionar la Lista del Producto (*Product Backlog*). La gestión de la Lista del Producto incluye:

- ✓ Expresar claramente los elementos de la Lista del Producto;
- ✓ Ordenar los elementos en la Lista del Producto para alcanzar los objetivos y misiones de la mejor manera posible;
- ✓ Optimizar el valor del trabajo desempeñado por el Equipo de Desarrollo;
- ✓ Asegurar que la Lista del Producto es visible, transparente y clara para todos, y que muestra aquello en lo que el equipo trabajará a continuación; y,
- ✓ Asegurar que el Equipo de Desarrollo entiende los elementos de la Lista del Producto al nivel necesario.

El Dueño de Producto podría hacer el trabajo anterior, o delegarlo en el Equipo de Desarrollo. Sin embargo, en ambos casos el Dueño de Producto sigue siendo el responsable de dicho trabajo.

El Dueño de Producto es una única persona, no un comité. El Dueño de Producto podría representar los deseos de un comité en la Lista del Producto, pero aquellos que quieran cambiar la prioridad de un elemento de la Lista deben hacerlo a través del Dueño de Producto.

Para que el Dueño de Producto pueda hacer bien su trabajo, toda la organización debe respetar sus decisiones. Las decisiones del Dueño de Producto se reflejan en el contenido y en la priorización de la Lista del Producto.

- El Equipo de Desarrollo (Development Team)

El Equipo de Desarrollo consiste en los profesionales que desempeñan el trabajo de entregar un Incremento de producto "Terminado", que potencialmente se pueda poner en producción, al final de cada Sprint. Solo los miembros del Equipo de Desarrollo participan en la creación del Incremento.
Los Equipos de Desarrollo son estructurados y empoderados por la organización para organizar y gestionar su propio trabajo. La sinergia resultante optimiza la eficiencia y efectividad del Equipo de Desarrollo.

Los Equipos de Desarrollo tienen las siguientes características:

- ✓ Son auto-organizados. Nadie (ni siquiera el Scrum Master) indica al Equipo de Desarrollo cómo convertir elementos de la Lista del Producto en Incrementos de funcionalidad potencialmente entregables;
- ✓ Los Equipos de Desarrollo son multifuncionales, contando como equipo con todas las habilidades necesarias para crear un Incremento de producto;
- ✓ Scrum no reconoce títulos para los miembros de un Equipo de Desarrollo, todos son Desarrolladores, independientemente del trabajo que realice cada persona; no hay excepciones a esta regla;
- ✓ Scrum no reconoce sub-equipos en los equipos de desarrollo, no importan los dominios particulares que requieran ser tenidos en cuenta, como pruebas o análisis de negocio; no hay excepciones a esta regla; y,
- ✓ Los Miembros individuales del Equipo de Desarrollo pueden tener habilidades especializadas y áreas en las que estén más enfocados, pero la responsabilidad recae en el Equipo de Desarrollo como un todo.

El tamaño óptimo del Equipo de Desarrollo es lo suficientemente pequeño como para permanecer ágil y lo suficientemente grande como para completar una cantidad de trabajo significativa. Tener menos de tres miembros en el Equipo de Desarrollo reduce la interacción y resulta en ganancias de productividad más pequeñas. Los Equipos de Desarrollo más pequeños podrían encontrar limitaciones en cuanto a las habilidades necesarias durante un Sprint, haciendo que el Equipo de Desarrollo no pudiese entregar un Incremento que potencialmente se pueda poner en producción. Tener más de nueve miembros en el equipo requiere demasiada coordinación. Los Equipos de Desarrollo grandes generan demasiada complejidad como para que pueda gestionarse mediante un proceso empírico. Los roles de Dueño de

Producto y Scrum Master no cuentan en el cálculo del tamaño del equipo a menos que también estén contribuyendo a trabajar en la Lista de Pendientes de Sprint (*Sprint Backlog*).

- El Scrum Master

El Scrum Master es el responsable de asegurar que Scrum es entendido y adoptado. Los Scrum Masters hacen esto asegurándose de que el Equipo Scrum trabaja ajustándose a la teoría, prácticas y reglas de Scrum.

El Scrum Master es un líder que está al servicio del Equipo Scrum. El Scrum Master ayuda a las personas externas al Equipo Scrum a entender qué interacciones con el Equipo Scrum pueden ser de ayuda y cuáles no. El Scrum Master ayuda a todos a modificar estas interacciones para maximizar el valor creado por el Equipo Scrum.

El Scrum Master da servicio al Dueño de Producto de varias formas, incluyendo:

- ✓ Encontrar técnicas para gestionar la Lista de Producto de manera efectiva;
- ✓ Ayudar al Equipo Scrum a entender la necesidad de contar con elementos de Lista de Producto claros y concisos;
- ✓ Entender la planificación del producto en un entorno empírico;
- ✓ Asegurar que el Dueño de Producto conozca cómo ordenar la Lista de Producto para maximizar el valor;
- ✓ Entender y practicar la agilidad; y,
- ✓ Facilitar los eventos de Scrum según se requiera o necesite.

El Scrum Master da servicio al Equipo de Desarrollo de varias formas, incluyendo:

- ✓ Guiar al Equipo de Desarrollo en ser auto-organizado y multifuncional;
- ✓ Ayudar al Equipo de Desarrollo a crear productos de alto valor;
- ✓ Eliminar impedimentos para el progreso del Equipo de Desarrollo;
- ✓ Facilitar los eventos de Scrum según se requiera o necesite; y,
- ✓ Guiar al Equipo de Desarrollo en el entorno de organizaciones en las que Scrum aún no ha sido adoptado y entendido por completo.

El Scrum Master da servicio a la organización de varias formas, incluyendo:

- ✓ Liderar y guiar a la organización en la adopción de Scrum;
- ✓ Planificar las implementaciones de Scrum en la organización;
- ✓ Ayudar a los empleados e interesados a entender y llevar a cabo Scrum y el desarrollo empírico de producto;
- ✓ Motivar cambios que incrementen la productividad del Equipo Scrum; y,
- ✓ Trabajar con otros Scrum Masters para incrementar la efectividad de la aplicación de Scrum en la organización.

Eventos de Scrum

En Scrum existen eventos predefinidos con el fin de crear regularidad y minimizar la necesidad de reuniones no definidas en Scrum. Todos los eventos deben respetar la característica de "time box", de tal modo que todos tienen una duración máxima, prefijada. Una vez que comienza un Sprint, su duración es fija y no puede acortarse o alargarse. Los demás eventos pueden terminar siempre que se alcance el objetivo del evento, asegurando que se emplee una cantidad apropiada de tiempo sin permitir desperdicio en el proceso.

Además del propio Sprint, que es un contenedor del resto de los eventos, cada uno de los eventos de Scrum constituye una oportunidad formal para la inspección y adaptación de algún aspecto. Estos eventos están diseñados específicamente para facilitar y permitir las vitales transparencia e inspección. La falta de alguno de estos eventos da como resultado una reducción de la transparencia y constituye una oportunidad perdida para inspeccionar y adaptarse.

- El Sprint

El corazón de Scrum es el Sprint, es un bloque de tiempo (*time-box*) de un mes o menos durante el cual se crea un incremento de producto "Terminado", utilizable y potencialmente desplegable. Es más conveniente si la duración de los Sprints es consistente a lo largo del esfuerzo de desarrollo. Cada nuevo Sprint comienza inmediatamente después de la finalización del Sprint previo.

Los Sprints contienen y consisten de la Reunión de Planificación del Sprint (*Sprint Planning Meeting*), los Scrums Diarios (*Daily Scrums*), el trabajo de desarrollo, la Revisión del Sprint (*Sprint Review*), y la Retrospectiva del Sprint (*Sprint Retrospective*).

Durante el Sprint:

- ✓ No se realizan cambios que puedan afectar al Objetivo del Sprint (*Sprint Goal*);
- ✓ Los objetivos de calidad no disminuyen; y,
- ✓ El alcance puede ser clarificado y renegociado entre el Dueño de Producto y el Equipo de Desarrollo a medida que se va aprendiendo más.

Cada Sprint puede considerarse un proyecto con un horizonte no mayor de un mes. Al igual que los proyectos, los Sprints se usan para lograr algo. Cada Sprint tiene una definición de qué se va a construir, un diseño y un plan flexible que guiará la construcción, el trabajo y el producto resultante.

Los Sprints están limitados a un mes calendario. Cuando el horizonte de un Sprint es demasiado grande la definición de lo que se está construyendo podría cambiar, la complejidad podría elevarse y el riesgo podría aumentar. Los Sprints habilitan la predictibilidad al asegurar la inspección y adaptación del progreso al menos en cada mes calendario. Los Sprints también limitan el riesgo al costo de un mes calendario.

- Reunión de Planificación de Sprint (Sprint Planning Meeting)

El trabajo a realizar durante el Sprint se planifica en la Reunión de Planificación de Sprint. Este plan se crea mediante el trabajo colaborativo del Equipo Scrum completo.

La Reunión de Planificación de Sprint tiene un máximo de duración de ocho horas para un Sprint de un mes. Para Sprints más cortos, el evento es usualmente más corto. El Scrum Master se asegura que el evento se lleve a cabo y que los asistentes entiendan su propósito.

La Reunión de Planificación de Sprint responde a las siguientes preguntas:

Tema Uno: ¿Qué puede ser terminado en este Sprint?

El Equipo de Desarrollo trabaja para proyectar la funcionalidad que se desarrollará durante el Sprint. El Dueño de Producto discute el objetivo que el Sprint debería lograr y los elementos de la Lista de Producto que, si se completan en el Sprint, lograrían el Objetivo del Sprint. El Equipo Scrum completo colabora en el entendimiento del trabajo del Sprint.

Después de que el Equipo de Desarrollo proyecta qué elementos de la Lista de Producto entregará en el Sprint, el Equipo Scrum elabora un Objetivo del Sprint (*Sprint Goal*). El Objetivo del Sprint debería lograrse durante el Sprint a través de la implementación de la Lista de Producto, y provee una guía al equipo de desarrollo de por qué se está construyendo el incremento.

Tema Dos: ¿Cómo se conseguirá completar el trabajo seleccionado?

Una vez que se ha establecido el objetivo y seleccionado los elementos de la Lista de Producto para el Sprint, el Equipo de Desarrollo decide cómo construirá esta funcionalidad para formar un Incremento de producto "Terminado". Los elementos de la Lista de Producto seleccionados para este Sprint, más el plan para terminarlos, recibe el nombre de Lista de Pendientes del Sprint (*Sprint Backlog*).

El Equipo de desarrollo se auto-organiza para asumir el trabajo de la Lista de Pendientes de Sprint, tanto durante la reunión de Planificación de Sprint como a lo largo del Sprint.

El Dueño de Producto puede ayudar a clarificar los elementos de la Lista de Producto seleccionados y hacer concesiones. Si el Equipo de Desarrollo determina que tiene demasiado trabajo o que no tiene suficiente trabajo, podría renegociar los elementos de la Lista de Producto seleccionados con el Dueño de Producto. El Equipo de Desarrollo podría también invitar a otras personas a que asistan con el fin de que proporcionen asesoría técnica o relacionada con el dominio.

Al finalizar la Reunión de Planificación de Sprint, el Equipo de Desarrollo debería ser capaz de explicar al Dueño de Producto y al Scrum Master cómo pretende trabajar como un equipo auto-organizado para lograr el Objetivo del Sprint y crear el Incremento esperado.

A medida que el equipo de desarrollo trabaja, se mantiene el objetivo del Sprint en mente. Con el fin de satisfacer el objetivo del Sprint se implementa la funcionalidad y la tecnología. Si el trabajo resulta ser diferente de lo que el Equipo de Desarrollo espera, ellos colaboran con el Dueño del Producto para negociar el alcance de la Lista de pendientes del Sprint (*Sprint Backlog*).

- Scrum Diario (Daily Scrum)

El Scrum Diario es una reunión con un bloque de tiempo de 15 minutos para que el Equipo de Desarrollo sincronice sus actividades y cree un plan para las siguientes 24 horas. Esto se lleva a cabo inspeccionando el trabajo avanzado desde el último Scrum Diario y haciendo una proyección acerca del trabajo que podría completarse antes del siguiente.

El Scrum Diario se realiza a la misma hora y en el mismo lugar todos los días para reducir la complejidad. Durante la reunión, cada miembro del Equipo de Desarrollo explica:

- ✓ ¿Qué hizo el día anterior que ayudó al Equipo de Desarrollo a lograr el Objetivo del Sprint?
- ✓ ¿Qué hará hoy para ayudar al Equipo de Desarrollo a lograr el Objetivo del Sprint?
- ✓ ¿Ve algún impedimento que evite que el Equipo de Desarrollo logre el Objetivo del Sprint?

El Equipo de Desarrollo usa el Scrum Diario para evaluar el progreso hacia el Objetivo del Sprint y para evaluar qué tendencia sigue este progreso hacia la finalización del trabajo contenido en la Lista del Sprint. El Scrum Diario optimiza las posibilidades de que el Equipo de Desarrollo cumpla el Objetivo del Sprint. Cada día, el Equipo de Desarrollo debería entender cómo intenta trabajar en conjunto como un equipo auto-organizado para lograr el Objetivo del Sprint y crear el Incremento esperado hacia el final del Sprint.

El Scrum Master se asegura que se cumpla la regla de que sólo los miembros del Equipo de Desarrollo participan en el Scrum Diario.

Los Scrum Diarios mejoran la comunicación, eliminan la necesidad de mantener otras reuniones, identifican y eliminan impedimentos relativos al desarrollo, resaltan y promueven la toma de decisiones rápida, y mejoran el nivel de conocimiento del Equipo de Desarrollo. El Scrum Diario constituye una reunión clave de inspección y adaptación.

- Revisión de Sprint (Sprint Review)

Al final del Sprint se lleva a cabo una Revisión de Sprint para inspeccionar el Incremento y adaptar la Lista de Producto si fuese necesario. Durante la Revisión de Sprint, el Equipo Scrum y los interesados discuten acerca de lo que se hizo durante el Sprint. Basándose en esto, y en cualquier cambio a la Lista de Producto durante el Sprint, los asistentes colaboran para determinar las siguientes cosas que podrían hacerse para optimizar el valor. Se trata de una reunión informal, no una reunión de *seguimiento*, y la presentación del Incremento tiene como objetivo facilitar la retroalimentación de información y fomentar la colaboración.

Se trata de una reunión restringida a un bloque de tiempo de cuatro horas para Sprints de un mes. Para Sprints más cortos, se reserva un tiempo proporcionalmente menor. El Scrum Master se asegura que el evento se lleve a cabo y que los asistentes entiendan su propósito. El Scrum Master enseña a todos a mantener el evento dentro del bloque de tiempo fijado.

La Revisión de Sprint incluye los siguientes elementos:

- ✓ Los asistentes son el Equipo Scrum y los interesados clave invitados por el Dueño de Producto;
- ✓ El Dueño de Producto explica qué elementos de la Lista de Producto se han "Terminado" y cuales no se han "Terminado";
- ✓ El Equipo de Desarrollo habla acerca de qué fue bien durante el Sprint, qué problemas aparecieron y cómo fueron resueltos esos problemas;
- ✓ El Equipo de Desarrollo demuestra el trabajo que ha "Terminado" y responde preguntas acerca del Incremento;
- ✓ El Dueño de Producto habla acerca de la Lista de Producto en el estado actual. Proyecta fechas de finalización probables en el tiempo basándose en el progreso obtenido hasta la fecha (si es necesario);
- ✓ El grupo completo colabora acerca de qué hacer a continuación, de modo que la Revisión del Sprint proporcione información de entrada valiosa para Reuniones de Planificación de Sprints subsiguientes.
- ✓ Revisión de cómo el mercado o el uso potencial del producto podría haber cambiado lo que es de más valor para hacer a continuación; y,
- ✓ Revisión de la línea de tiempo, presupuesto, capacidades potenciales y mercado para la próxima entrega prevista del producto.

El resultado de la Revisión de Sprint es una Lista de Producto revisada, que define los elementos de la Lista de Producto posibles para el siguiente Sprint. Es posible además que la Lista de Producto reciba un ajuste general para enfocarse en nuevas oportunidades.

- Retrospectiva de Sprint (Sprint Retrospective)

La Retrospectiva de Sprint es una oportunidad para el Equipo Scrum de inspeccionarse a sí mismo y crear un plan de mejoras que sean abordadas durante el siguiente Sprint.

La Retrospectiva de Sprint tiene lugar después de la Revisión de Sprint y antes de la siguiente Reunión de Planificación de Sprint. Se trata de una reunión restringida a un bloque de tiempo de tres horas para Sprints de un mes. Para Sprints más cortos se reserva un tiempo proporcionalmente menor. El Scrum Master se asegura que el evento se lleve a cabo y que los asistentes entiendan su propósito. El Scrum Master enseña a todos a mantener el evento dentro del bloque de tiempo fijado. El Scrum Master participa en la reunión como un miembro del equipo ya que la responsabilidad del proceso Scrum recae sobre él. El propósito de la Retrospectiva de Sprint es:

- ✓ Inspeccionar cómo fue el último Sprint en cuanto a personas, relaciones, procesos y herramientas;
- ✓ Identificar y ordenar los elementos más importantes que salieron bien y las posibles mejoras; y,
- ✓ Crear un plan para implementar las mejoras a la forma en la que el Equipo Scrum desempeña su trabajo.

El Scrum Master alienta al equipo para que mejore, dentro del marco de proceso Scrum, su proceso de desarrollo y sus prácticas para hacerlos más efectivos y amenos para el siguiente Sprint. Durante cada Retrospectiva de Sprint, el Equipo Scrum planifica formas

de aumentar la calidad del producto mediante la adaptación de la Definición de "Terminado" (*Definition of "Done"*) según sea conveniente.

Para el final de la Retrospectiva de Sprint, el Equipo Scrum debería haber identificado mejoras que implementará en el próximo Sprint. El hecho de implementar estas mejoras en el siguiente Sprint, constituye la adaptación subsecuente a la inspección del Equipo de Desarrollo a sí mismo. Aunque las mejoras pueden implementarse en cualquier momento, la Retrospectiva de Sprint ofrece un evento dedicado para este fin, enfocado en la inspección y la adaptación.

Artefactos de Scrum

Los artefactos de Scrum representan trabajo o valor en diversas formas que son útiles para proporcionar transparencia y oportunidades para la inspección y adaptación. Los artefactos definidos por Scrum están diseñados específicamente para maximizar la transparencia de la información clave, que es necesaria para asegurar que todos tengan el mismo entendimiento del artefacto.

- Lista de Producto (Product Backlog)

La Lista de Producto es una lista ordenada de todo lo que podría ser necesario en el producto, y es la única fuente de requisitos para cualquier cambio a realizarse en el producto. El Dueño de Producto (*Product Owner*) es el responsable de la Lista de Producto, incluyendo su contenido, disponibilidad y ordenación.

Una Lista de Producto nunca está completa. El desarrollo más temprano de la misma sólo refleja los requisitos conocidos y mejor entendidos al principio. La Lista de Producto evoluciona a medida que el producto y el entorno en el que se usará también lo hacen. La Lista de Producto es dinámica; cambia constantemente para identificar lo que el producto necesita para ser adecuado, competitivo y útil. Mientras el producto exista, su Lista de Producto también existe.

La Lista de Producto enumera todas las características, funcionalidades, requisitos, mejoras y correcciones que constituyen cambios a ser hechos sobre el producto para entregas futuras. Los elementos de la Lista de Producto tienen como atributos la descripción, la ordenación, la estimación y el valor.

A medida que un producto es utilizado y se incrementa su valor, y el mercado proporciona retroalimentación, la Lista de Producto se convierte en una lista más larga y exhaustiva. Los requisitos nunca dejan de cambiar, así que la Lista de Producto es un artefacto vivo. Los cambios en los requisitos de negocio, las condiciones del mercado o la tecnología podrían causar cambios en la Lista de Producto.

A menudo, varios Equipos Scrum trabajan juntos en el mismo producto. Para describir el trabajo a realizar sobre el producto, se utiliza una única Lista de Producto. En ese caso podría emplearse un atributo de la Lista de Producto para agrupar los elementos.

El refinamiento de la Lista de Producto es el acto de añadir detalle, estimaciones y orden a los elementos de la Lista de Producto. Se trata de un proceso continuo, en el cual el Dueño de Producto y el Equipo de Desarrollo colaboran acerca de los detalles de los elementos de

la Lista de Producto. Durante el refinamiento de la Lista de Producto, se examinan y revisan sus elementos. El Equipo Scrum decide cómo y cuándo se hace el refinamiento. Este usualmente consume no más del 10% de la capacidad del Equipo de Desarrollo. Sin embargo, los elementos de la Lista de Producto pueden actualizarse en cualquier momento por el Dueño de Producto o a criterio suyo.

Los elementos de la Lista de Producto de orden más alto son generalmente más claros y detallados que los de menor orden. Se realizan estimaciones más precisas basándose en la mayor claridad y detalle; cuanto más bajo es el orden, menor es el detalle. Los elementos de la Lista de Producto de los que se ocupará el Equipo de Desarrollo en el siguiente Sprint tienen una granularidad mayor, habiendo sido descompuestos de forma que cualquier elemento puede ser "Terminado" dentro de los límites del bloque de tiempo del Sprint. Los elementos de la Lista de Producto que pueden ser "Terminados" por el Equipo de Desarrollo en un Sprint son considerados "preparados" o "accionables" para ser seleccionados en una reunión de Planificación de Sprint. Los elementos de la Lista de Producto normalmente adquieren este grado de transparencia mediante las actividades de refinamiento descriptas anteriormente.
El Equipo de Desarrollo es el responsable de proporcionar todas las estimaciones. El Dueño de Producto podría influenciar al Equipo ayudándoles a entender y seleccionar soluciones de compromiso, pero las personas que harán el trabajo son las que hacen la estimación final.

- Lista de Pendientes del Sprint (Sprint Backlog)

La Lista de Pendientes del Sprint es el conjunto de elementos de la Lista de Producto seleccionados para el Sprint, más un plan para entregar el Incremento de producto y conseguir el Objetivo del Sprint. La Lista de Pendientes del Sprint es una predicción hecha por el Equipo de Desarrollo acerca de qué funcionalidad formará parte del próximo Incremento y del trabajo necesario para entregar esa funcionalidad en un Incremento "Terminado".

La Lista de Pendientes del Sprint hace visible todo el trabajo que el Equipo de Desarrollo identifica como necesario para alcanzar el Objetivo del Sprint.

La Lista de Pendientes del Sprint es un plan con un nivel de detalle suficiente como para que los cambios en el progreso se puedan entender en el Scrum Diario. El Equipo de Desarrollo modifica la Lista de Pendientes del Sprint durante el Sprint y esta Lista de Pendientes del Sprint emerge a lo largo del Sprint. Esto ocurre a medida que el Equipo de Desarrollo trabaja sobre el plan y aprende más acerca del trabajo necesario para conseguir el Objetivo del Sprint.

Según se requiere nuevo trabajo, el Equipo de Desarrollo lo añade a la Lista de Pendientes del Sprint. A medida que el trabajo se ejecuta o se completa, se va actualizando la estimación de trabajo restante. Cuando algún elemento del plan pasa a ser considerado innecesario, es eliminado. Solo el Equipo de Desarrollo puede cambiar su Lista de Pendientes del Sprint durante un Sprint. La Lista de Pendientes del Sprint es una imagen visible en tiempo real del trabajo que el Equipo de Desarrollo planea llevar a cabo durante el Sprint, y pertenece únicamente al Equipo de Desarrollo. En cualquier momento durante un Sprint, es posible sumar el trabajo restante total en los elementos de la Lista de Pendientes del Sprint. El Equipo de Desarrollo hace seguimiento de este trabajo restante

total al menos en cada Scrum Diario para proyectar la posibilidad de conseguir el Objetivo del Sprint. Haciendo seguimiento del trabajo restante a lo largo del Sprint, el Equipo de Desarrollo puede gestionar su progreso.

- Incremento

El Incremento es la suma de todos los elementos de la Lista de Producto completados durante un Sprint y el valor de los incrementos de todos los Sprints anteriores. Al final de un Sprint, el nuevo Incremento debe estar "Terminado", lo cual significa *que está en condiciones de ser utilizado* y que cumple la Definición de "Terminado" del Equipo Scrum. El *incremento debe estar en condiciones de utilizarse sin importar si el Dueño de Producto decide liberarlo o no*.

Transparencia de los Artefactos

Scrum se basa en la transparencia. Las decisiones para optimizar el valor y controlar el riesgo se toman basadas en el estado percibido de los artefactos. En la medida en que la transparencia sea completa, estas decisiones tienen unas bases sólidas. En la medida en que los artefactos no son completamente transparentes, estas decisiones pueden ser erróneas, el valor puede disminuir y el riesgo puede aumentar.

El Scrum Master debe trabajar con el Dueño de Producto, el Equipo de Desarrollo y otras partes involucradas para entender si los artefactos son completamente transparentes. Hay prácticas para hacer frente a la falta de transparencia; el Scrum Master debe ayudar a todos a aplicar las prácticas más apropiadas si no hay una transparencia completa. Un Scrum Master puede detectar la falta de transparencia inspeccionando artefactos, reconociendo patrones, escuchando atentamente lo que se dice y detectando diferencias entre los resultados esperados y los reales.

La labor del Scrum Master es trabajar con el Equipo Scrum y la organización para mejorar la transparencia de los artefactos. Este trabajo usualmente incluye aprendizaje, convicción y cambio. La transparencia no ocurre de la noche a la mañana, sino que es un camino.

Definición de "Terminado" (Definition of "Done")

Cuando un elemento de la Lista de Producto o un Incremento se describe como "Terminado", todo el mundo debe entender lo que significa "Terminado". Aunque esto varía significativamente para cada Equipo Scrum, los miembros del Equipo deben tener un entendimiento compartido de lo que significa que el trabajo esté completado, para asegurar la transparencia. Esta es la definición de "Terminado" para el Equipo Scrum y se utiliza para evaluar cuándo se ha completado el trabajo sobre el Incremento de producto.

Esta misma definición guía al Equipo de Desarrollo en saber cuántos elementos de la Lista de Producto puede seleccionar durante una reunión de Planificación de Sprint. El propósito de cada Sprint es entregar Incrementos de funcionalidad que potencialmente se puedan poner en producción, y que se ajustan a la Definición de "Terminado" actual del Equipo Scrum.

Los Equipos de Desarrollo entregan un Incremento de funcionalidad de producto en cada Sprint. Este Incremento es utilizable, de modo que el Dueño de Producto podría elegir liberarlo inmediatamente. Si la definición de “Terminado” para un incremento **es** parte de las convenciones, estándares o guías de la organización de desarrollo, al menos todos los Equipos Scrum deben seguirla. Si “Terminado” para un incremento **no** es una convención de la organización de desarrollo, el Equipo de Desarrollo del Equipo Scrum debe crear una definición de “Terminado” apropiada para el producto. Si hay múltiples Equipos Scrum trabajando en la entrega del sistema o producto, los equipos de desarrolladores en todos los Equipos Scrum deben acordar conjuntamente la definición de “Terminado”.

Cada Incremento se integra con todos los Incrementos anteriores y es probado exhaustivamente, asegurando que todos los Incrementos funcionan en conjunto.

A medida que los Equipos Scrum maduran, se espera que su definición de “Terminado” se amplíe para incluir criterios más rigurosos para una mayor calidad. Cualquier producto o sistema debería tener una definición de “Terminado” que es un estándar para cualquier trabajo realizado sobre él.

Extreme Programming (XP)[12]

También conocido como programación extrema, esta metodología enfatiza las prácticas de ingeniería de software. La programación extrema es una metodología ágil, basada en control de procesos empírico.

XP adhiere a los principios del manifiesto Ágil. XP considera que los cambios de requerimientos durante el ciclo de vida de un proyecto son algo natural y hasta deseable en el desarrollo de software: poder incorporar cambios en cualquier momento durante el desarrollo de un sistema es una aproximación más realista que aquellas que intentan definir todo desde un principio y condensar el esfuerzo en controlar y evitar los cambios.

Valores de XP

Los valores originales de la programación extrema son: simplicidad, comunicación, retroalimentación (feedback) y coraje. Un quinto valor, el respeto, fue añadido en la segunda edición del libro Extreme Programming Explained (Kent Beck). Los cinco valores[13] se explican así:

Simplicidad

Descripto por la agilidad como "el arte de maximizar la cantidad de trabajo no realizado", la simplicidad es el valor principal de la programación extrema. *Simplificar los diseños y arquitecturas agiliza el desarrollo y facilita el mantenimiento*. Un diseño complejo del código, junto con sucesivas modificaciones realizadas por diferentes desarrolladores, aumenta la complejidad en forma exponencial. Una de las prácticas fundamentales a la hora de mantener la simplicidad en el diseño es la llamada refactorización (la que se tratará más adelante).

Otro punto importante sobre la simplicidad es la documentación, donde se busca que el código se comente lo justo y necesario, y que, por medio de técnicas de nomenclatura de variables, métodos y clases, el código se transforme en un activo autodocumentado.

Comunicación

La comunicación se realiza de diferentes formas. Para los programadores el código comunica mejor cuanto más simple sea. Si el código es complejo hay que esforzarse para hacerlo legible. El código autodocumentado es más fiable que los comentarios ya que estos últimos pronto quedan desfasados con el código a medida que es modificado. Por ello debe comentarse sólo aquello que no va a variar, por ejemplo, el objetivo de una clase o la funcionalidad de un método.

Las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de cómo utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la programación por parejas y la comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo. El cliente es quien decide qué características tienen prioridad y él siempre debe estar disponible para despejar dudas.

[12] Kent Beck, *Extreme Programming Explained: Embrace Change,* 1999.

[13] http://es.wikipedia.org/wiki/Programaci%C3%B3n_extrema

Feedback

Al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real. Como los ciclos son muy cortos, los resultados se muestran con frecuencia y se minimiza la necesidad de rehacer partes que no cumplan con los requisitos, ayudando a los programadores a centrarse en lo que es más importante.

Considérense los problemas derivados de tener ciclos muy largos. Meses de trabajo pueden tirarse por la borda debido a cambios en los criterios del cliente o malentendidos por parte del equipo de desarrollo. El código también es una fuente de retroalimentación gracias a las herramientas de desarrollo. Por ejemplo, las pruebas unitarias informan sobre el “estado de salud” del código. Ejecutar las pruebas unitarias frecuentemente permite descubrir fallas debido a cambios recientes producidos en el código.

Coraje

Los puntos anteriores parecen tener sentido común, entonces ¿por qué coraje? Para la alta gerencia, la programación en parejas puede ser difícil de aceptar porque a primera vista pareciera que la productividad disminuye a la mitad ya que sólo la mitad de los programadores está escribiendo código. Hay que ser valiente para confiar en que la programación por parejas beneficia la calidad del código sin repercutir negativamente en la productividad.

La simplicidad es uno de los principios más difíciles de adoptar. Se requiere coraje para implementar las características que el cliente quiere ahora sin caer en la tentación de optar por un enfoque más flexible que permita futuras modificaciones. No se debe emprender el desarrollo de grandes marcos de trabajo (frameworks) mientras el cliente espera. En ese tiempo el cliente no recibe noticias sobre los avances del proyecto y el equipo de desarrollo no recibe retroalimentación para saber si va en la dirección correcta. La forma de construir marcos de trabajo es mediante la refactorización del código en sucesivas aproximaciones.

Respeto

El respeto se manifiesta de varias formas. Los miembros del equipo se respetan los unos a los otros porque los programadores no pueden realizar cambios que hagan que las pruebas existentes fallen o que demore el trabajo de sus compañeros. Respetan su trabajo porque siempre están luchando por la alta calidad en el producto y buscando el diseño óptimo o más eficiente para la solución a través de la refactorización del código. Finalmente, respetan el trabajo del resto no menospreciando a otros, sino orientándolos a mejorar, obteniendo como resultado una mayor autoestima en el equipo y elevando su ritmo de producción.

XP es una metodología ágil que integra un conjunto de prácticas, que se muestran en la siguiente figura y que se explicarán brevemente a continuación de la misma:

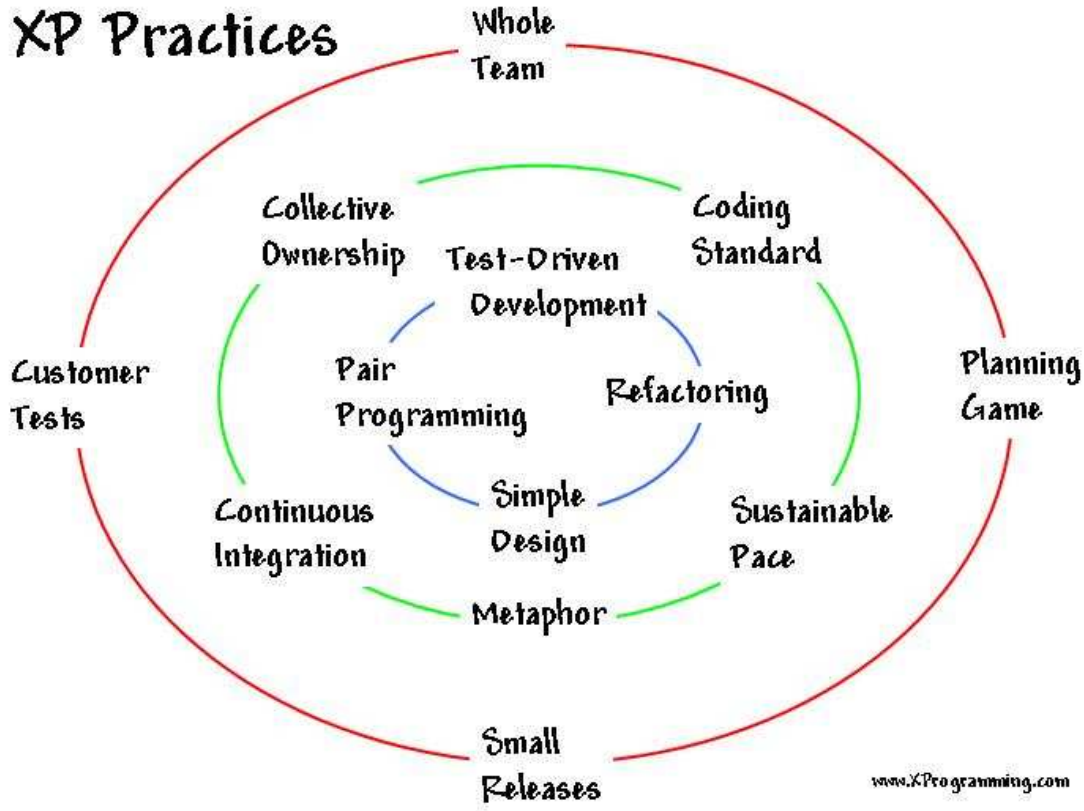


Figura 37: Prácticas de XP

Prácticas de XP

- **Programación por parejas (Pair Programming)**

Esta práctica basada en el principio de "cuatro ojos ven más que dos", consiste en ubicar en cada puesto de trabajo dos desarrolladores. Uno de ellos programa y el otro revisa el trabajo que el primero hace, mientras los está haciendo. Los roles se intercambian en ciclos de aproximadamente 2 horas. Está demostrado que la productividad y la calidad del código producidos por una pareja de programadores es mucho mayor al resultado obtenido por la suma de los logros de los programadores en forma aislada. Algunas conclusiones hasta ahora respecto a la programación por parejas:

- ✓ La programación por parejas realmente mejora la calidad del código.
- ✓ La programación por parejas realmente mejora la concentración del equipo (por ejemplo, cuando la persona que está sentada contigo dice "hey, ¿realmente necesitamos eso para este Sprint?"
- ✓ Sorprendentemente, muchos de los desarrolladores que están totalmente en contra de la programación por parejas ni siquiera la han probado, y aprenden rápidamente a apreciarla una vez que la prueban.
- ✓ La programación por parejas es agotadora y no debería hacerse durante todo el día.
- ✓ Es bueno cambiar de parejas frecuentemente.
- ✓ La programación por parejas realmente mejora la distribución de conocimiento entre el equipo. Sorprendentemente rápido, además.

- ✓ Algunas personas simplemente no se sienten a gusto haciendo programación por parejas. No prescindas de un programador excelente simplemente porque no se sienta a gusto programando por parejas.
- ✓ La revisión de código es una alternativa aceptable a la programación por parejas.
- ✓ El "navegante" (la persona que no usa el teclado) debería tener una computadora propia también. No para desarrollar, pero sí para hacer pequeñas tareas cuando sea necesario, consultar documentación cuando el "piloto" (persona que usa el teclado) se atasque, etc.

- **Desarrollo guiado por pruebas (Test Driven Development)**

Desarrollo guiado por pruebas significa que escribes un test automático y, a continuación, escribes el código suficiente para pasar dicho test y después re factorizas el código, principalmente para mejorar la legibilidad y eliminar duplicaciones. Aclarar y repetir.

Si bien hay una sección diferenciada para esta práctica, que ya se ha convertido en una metodología de desarrollo en sí misma, se presentan algunas reflexiones sobre el desarrollo guiado por pruebas:

- ✓ TDD es duro. Los programadores tardan un tiempo en aprenderlo. De hecho, en muchos casos no importa cuanto lo expliques, lo demuestres y los animes: en muchos casos la única forma que un programador lo entienda es emparejarlo con otro programador que sea bueno en TDD. Una vez que un programador lo incorpora, no querrá trabajar de otra forma.
- ✓ TDD tiene un efecto profundamente positivo en del diseño del sistema.
- ✓ Se tarda un tiempo en conseguir que TDD funcione en un nuevo producto, especialmente con pruebas de integración tipo "caja negra",
- ✓ Asegurar que se invierte suficiente tiempo en hacer que a la gente le resulte fácil escribir pruebas. Esto significa conseguir las herramientas adecuadas, educar a las personas, proporcionarles las clases de utilidad, clases básicas adecuadas, etc.
- ✓ TDD también actúa como una red de seguridad, proporcionando a los desarrolladores suficiente confianza como para re factorizar a menudo, lo que significa que el diseño se mantiene limpio y simple incluso mientras el sistema crece.

- **Re factorización frecuente (Refactoring)**

La re factorización ("refactoring") consiste en escribir nuevamente parte del código de un programa, sin cambiar su funcionalidad, a los efectos de hacerlo más simple, conciso y/o entendible. Muchas veces, al terminar de escribir un código de un programa, pensamos que, si lo comenzáramos de nuevo, lo hubiéramos hecho en forma diferente, más clara y eficientemente. Sin embargo, como ya está listo y "funciona", rara vez es reescrito. XP sugiere recodificar cada vez que sea necesario. Si bien, puede parecer una pérdida de tiempo innecesaria en el plazo inmediato, los resultados de ésta práctica tienen sus frutos en las siguientes iteraciones, cuando sea necesario ampliar o cambiar la funcionalidad. La filosofía que se persigue es, como ya se mencionó, tratar de mantener el código más simple posible que implemente la funcionalidad deseada.

- **Diseño Simple**

  Mantener los diseños y arquitecturas simples. No sobredimensionar y evitar el hecho de caer en la predicción de funcionalidades.

  Un diseño simple se implementa más rápidamente que uno complejo. Por ello, XP propone implementar el diseño más simple posible que funcione. Se sugiere nunca adelantar la implementación de funcionalidades que no correspondan a la iteración en la que se esté trabajando.

- **Integración continua (Continuous Integration)**

  La Integración continua es una práctica de desarrollo donde los miembros de un equipo integran su trabajo frecuentemente, generalmente una persona íntegra por lo menos diariamente.

  Cada integración es verificada por una herramienta de build automático (incluido un test) para detectar errores de integración tan pronto como sea posible.

  Es una práctica extremadamente valiosa y ahorra muchísimo tiempo. Es la solución definitiva al viejo problema de “hey, en mí máquina sí funciona”. Nuestro servidor de compilación continua, actúa como el “juez” o punto de referencia desde el que determinar la salud de nuestro código.

  Cada vez que alguien chequea algo en el sistema de control de versiones, el servidor de compilación continua arranca, compila todo desde cero en un servidor compartido, y corre todas las pruebas. Si algo va mal, manda un correo notificando a todo el equipo que la compilación ha fallado, incluyendo información sobre qué parte del código falló la compilación exactamente, enlaces a los informes de pruebas, etc.

  Todas las noches el servidor de compilación continúa reconstruye el producto desde cero y publica los binarios, documentación, informes de pruebas, informes de cobertura de pruebas, informes de dependencias, etc., a un portal interno de documentación. Algunos productos también se instalarán automáticamente en un entorno de pruebas.

- **Propiedad colectiva del código (Collective Ownership)**

  Los equipos que tienen un alto nivel de propiedad colectiva del código han probado ser muy robustos. Por ejemplo, sus Sprints no fallan simplemente porque una persona clave esté enferma.

  El principio de la propiedad colectiva intenta evitar que ciertas porciones de código o del sistema en sí pertenezcan en la práctica a un programador o grupo de programadores específicos sin dejar a otros acceder y/o modificar dicho código. En XP, todos son dueños de todo y todos están autorizados a revisar y cambiar el código de las aplicaciones cuando lo crean conveniente, sin importar quien lo haya escrito. Este principio debe estar reforzado por medio de la simplicidad y los estándares de codificación.

Espacio informativo

Todos los equipos tienen acceso a pizarras y espacios vacíos en las paredes, y hacen buen uso de ellos. En la mayoría de las salas encontrarás las paredes empapeladas de toda clase de información sobre el producto y el proyecto. Es recomendable el uso de tableros de tareas.

- **Estandarización de código**

  El objetivo de un estándar de codificación es que todos los programadores escriban el código siguiendo una serie común de parámetros para que parezca escrito en su totalidad por una única persona. En la práctica se recomienda comenzar con algo simple, que luego crecerá con el tiempo. La mayoría de los programadores tienen su propio y distintivo estilo de programación. Pequeños detalles como la forma en la que tratan las excepciones, cómo comentan el código, cuándo devuelven un valor null, etc. En algunos casos esta diferencia no importa, pero en otros puede conducir a una severa inconsistencia del diseño del sistema y a un código difícil de leer. Un estándar de código ayuda a que esto no ocurra, siempre que te concentres en las cosas que importan.

- **Ritmo sostenible / trabajo enérgico**

La metodología XP indica que debe llevarse un ritmo sostenido de trabajo. Anteriormente, ésta práctica se denominaba “Semana de 40 horas”. Sin embargo, lo importante no es si se trabajan, 35, 40 o 42 horas por semana. El concepto que se desea establecer con esta práctica es el de planificar el trabajo de manera de mantener un ritmo constante y razonable, sin sobrecargar al equipo.

Cuando un proyecto se retrasa, trabajar tiempo extra puede ser más perjudicial que beneficioso. El trabajo extra desmotiva inmediatamente al grupo e impacta en la calidad del producto. En la medida de lo posible, se debería renegociar el plan de entregas (“Release Plan”), realizando una nueva reunión de planificación con el cliente, los desarrolladores y los gerentes. Adicionalmente, agregar más desarrolladores en proyectos ya avanzados casi nunca resuelve el problema.

- **Metáforas de Sistema**

Una “metáfora” es algo que todos entienden, sin necesidad de mayores explicaciones. XP sugiere utilizar este concepto como una manera sencilla de explicar el propósito del proyecto, y guiar la estructura y arquitectura del mismo. Por ejemplo, puede ser una guía para la nomenclatura de los métodos y las clases utilizadas en el diseño del código. Tener nombres claros, que no requieran de mayores explicaciones, redunda en un ahorro de tiempo.

Es muy importante que el cliente y el grupo de desarrolladores estén de acuerdo y compartan esta “metáfora”, para que puedan dialogar en un “mismo idioma”. Una buena metáfora debe ser fácil de comprender para el cliente y a su vez debe tener suficiente contenido como para que sirva de guía a la arquitectura del producto. Sin embargo, ésta práctica resulta, muchas veces, difícil de realizar.

- **Release Pequeños y Frecuentes**

Producir rápidamente versiones del sistema que sean operativas, aunque no cuenten con toda la funcionalidad del sistema. Esta versión ya constituye un resultado de valor para el negocio. Un release no debería tardar más 3 meses.

- **Planificar el Juego**

La planificación en XP responde dos preguntas clave del desarrollo de software: predecir qué se habrá terminado para la fecha de entrega, y determinar qué hacer después. Se hace énfasis en guiar al proyecto -que es bastante directo- en vez de predecir exactamente lo que se necesitará y cuánto tiempo tomará hacerlo -que es bastante difícil. Hay dos pasos claves en la planificación de XP, que responde estas dos preguntas:

  - *Planificación de la Entrega*, es una práctica en donde el Cliente presenta las características deseadas a los programadores, y los programadores estiman la dificultad. Teniendo las estimaciones de costo, y sabiendo la importancia de las características, el Cliente establece un plan para el proyecto. Los planes iniciales de entregas son necesariamente imprecisos: ni las prioridades ni las estimaciones son sólidas, y tampoco sabremos qué tan rápido trabaja el equipo hasta que empiece a trabajar. Sin embargo, incluso el primer plan de entrega es lo suficientemente preciso como para tomar decisiones, y el equipo XP revisa de forma regular el plan.

  - *Planificación de la Iteración*, es la práctica en donde el equipo establece el rumbo cada un par de semanas. Los equipos XP construyen software en iteraciones de dos semanas, y entregan software útil al finalizar cada iteración. Durante la Planificación de la Iteración, el Cliente presenta las características deseadas para las siguientes dos semanas. Los programadores las descomponen en tareas, y estiman su costo (a un nivel de detalle más fino que durante la Planificación de la Entrega). El equipo entonces se compromete a terminar ciertas características basándose en la cantidad de trabajo que pudieron terminar en la iteración anterior.

Estos pasos de planificación son muy simples, y le brindan al cliente muy buena información y excelente flexibilidad para guiar al proyecto. Cada dos semanas se hace completamente visible el progreso. No existe el "90% terminado" en XP: una historia está terminada, o no lo está. Este foco en la transparencia resulta en una bonita paradoja: por un lado, con tanta visibilidad, el Cliente está en la posición de cancelar el proyecto si el progreso no es suficiente. Por otro lado, como el progreso es tan visible, y hay completa libertad para decidir qué se hará después, los proyectos XP tienden a entregar más de lo necesario, con menos presión y estrés.

- **Pruebas de Cliente**

  Pruebas unitarias
  Las pruebas unitarias son una de las piedras angulares de XP. Todos los módulos deben pasar las pruebas unitarias antes de ser liberados o publicados. Por otra parte, como se mencionó anteriormente, las pruebas deben ser definidas antes de realizar el código ("Test-driven programming"). Que todo código liberado pase correctamente las pruebas unitarias es lo que habilita que funcione la propiedad colectiva del código. En este sentido, el sistema y el conjunto de pruebas debe ser guardado junto con el código, para que pueda ser utilizado por otros desarrolladores, en caso de tener que corregir, cambiar o recodificar parte del mismo.

  Detección y corrección de errores
  Cuando se encuentra un error ("bug"), éste debe ser corregido inmediatamente, y se deben tener precauciones para que errores similares no vuelvan a ocurrir. Asimismo, se generan nuevas pruebas para verificar que el error haya sido resuelto.

  Pruebas de aceptación
  Las pruebas de aceptación son creadas en base a las historias de usuarios, en cada ciclo de la iteración del desarrollo. El cliente debe especificar uno o diversos escenarios para comprobar que una historia de usuario ha sido correctamente implementada.
  Las pruebas de aceptación son consideradas como "pruebas de caja negra". Los clientes son responsables de verificar que los resultados de éstas pruebas sean correctos. Asimismo, en caso que fallen varias pruebas, deben indicar el orden de prioridad de resolución.
  Una historia de usuario no se puede considerar terminada hasta tanto pase correctamente todas las pruebas de aceptación.
  Dado que la responsabilidad es grupal, es recomendable publicar los resultados de las pruebas de aceptación, de manera que todo el equipo esté al tanto de esta información.

- **El equipo completo trabajando junto**

  El equipo completo -desarrolladores y clientes- trabajando juntos en un espacio común asignado para el proyecto. Uno o más clientes están junto al equipo casi de forma permanente. Se espera que sean expertos en el tema y con capacidad de tomar decisiones respecto de los requerimientos y su prioridad.

  La contribución del cliente incluye explicaciones detalladas de las características que están resumidas en el tablero como tarjetas de usuario (user stories). Participar en la planificación, clarificar dudas, escribir pruebas de aceptación en colaboración con los desarrolladores.

  El propósito de la práctica es en respuesta a investigaciones que indican que el involucramiento del cliente es fundamental para el éxito de los proyectos.

TDD (Test Driven Development)

El Desarrollo Dirigido por Pruebas (Test Driven Development), referido como TDD, es una técnica de diseño e implementación de software incluida originalmente dentro de la metodología XP. TDD se centra en tres pilares fundamentales:

- La implementación de las funciones justas que el cliente necesita y no más.
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

TDD es una buena técnica que convierte al programador en desarrollador. TDD es la respuesta a grandes preguntas:

¿Cómo lo hago?, ¿Por dónde empiezo?, ¿Cómo sé qué es lo que hay que implementar y lo que no?, ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente?

Pasamos, de pensar en implementar tareas, a pensar en ejemplos certeros que eliminen la ambigüedad creada por la prosa en lenguaje natural (nuestro idioma). Hasta ahora estábamos acostumbrados a que las tareas, o lo casos de uso, eran las unidades de trabajo más pequeñas sobre las que ponerse a desarrollar código. Con TDD intentamos traducir el caso de uso o tarea en X ejemplos, hasta que el número de ejemplos sea suficiente como para describir la tarea sin lugar a malinterpretaciones de ningún tipo.

TDD permite que la propia implementación de pequeños ejemplos, en constantes iteraciones, hagan emerger la arquitectura que necesitamos usar. Ni más ni menos. No es que nos despreocupemos por completo de las características técnicas de la aplicación a priori, es decir, lógicamente tendremos que saber si el desarrollo será para un teléfono móvil, para una web o para un pc de escritorio; más que nada porque tenemos que elegir unas herramientas de desarrollo conformes a las exigencias del guión. Sin embargo, nos limitamos a escoger el framework correspondiente y a usar su arquitectura como base. TDD produce una arquitectura que emerge de la no-ambigüedad de los tests automatizados, lo cual no exime de las revisiones de código entre compañeros ni de hacer preguntas a los desarrolladores más experimentados del equipo.

A continuación, se describen algunas de las razones que da Kent Beck (uno de los creadores de XP) y otras destacadas figuras de la industria, respecto de utilizar TDD:

- La calidad del software aumenta.
- Se obtiene código altamente reutilizable.
- El trabajo en equipo se hace más fácil, une a las personas.
- Nos permite confiar en nuestros compañeros, aunque tengan menos experiencia.
- Multiplica la comunicación entre los miembros del equipo.
- Las personas encargadas del aseguramiento de calidad adquieren un rol más inteligente e interesante.
- Escribir el ejemplo (test) antes que el código obliga a escribir el mínimo de funcionalidad necesaria, evitando sobre-diseñar.
- Cuando se revisa un proyecto desarrollado mediante TDD, se advierte que los tests son la mejor documentación técnica que podemos consultar a la hora de entender qué misión cumple cada pieza de software.
- Incrementa la productividad.

Ahora bien, como cualquier técnica, no es una varita mágica y no dará el mismo resultado a un experto arquitecto de software que a un programador junior que está empezando. Sin embargo, es útil para ambos y para todo el rango de integrantes del equipo que hay entre uno y otro.

El algoritmo TDD

La esencia de TDD es sencilla, pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD soló tiene tres pasos:

- Escribir la especificación del requisito (el ejemplo, el test).
- Implementar el código según dicho ejemplo.
- Re factorizar para eliminar duplicidad y hacer mejoras.

La siguiente figura representa en forma genérica la técnica de TDD:

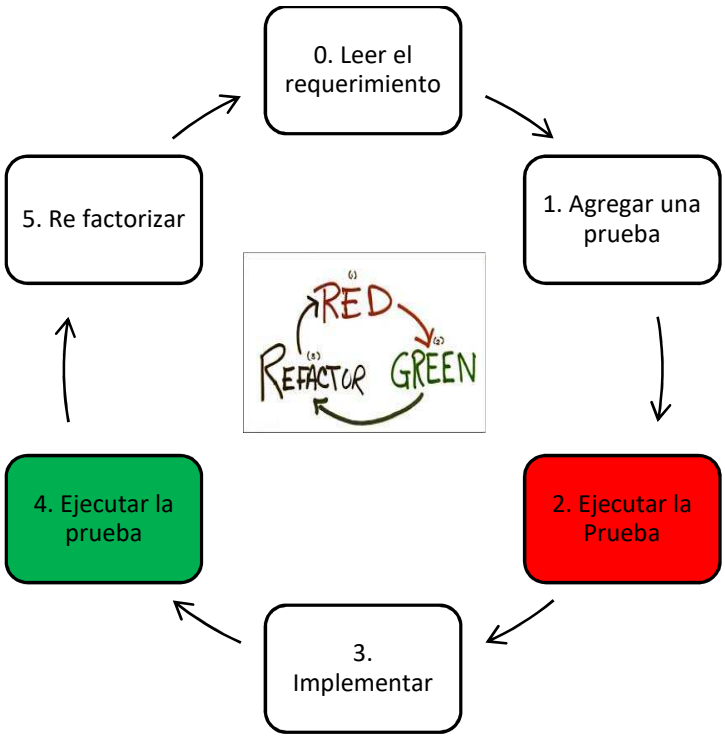


Figura 38: Técnica de TDD

Escribir la especificación primero

Una vez que tenemos claro cuál es el requisito, lo expresamos en forma de código. ¿Cómo escribimos una prueba para un código que todavía no existe? Respondamos con otra pregunta ¿Acaso no es posible escribir una especificación antes de implementarla? Una prueba, no es inicialmente una prueba, sino un ejemplo o especificación. La palabra especificación podría tener la connotación de que es inamovible, algo preestablecido y fijo, pero no es así. Una prueba se puede modificar. Tenemos que hacer el esfuerzo de imaginar cómo sería el código si ya estuviera implementado y cómo comprobaríamos que, efectivamente, hace lo que le pedimos que haga. El hecho de tener que usar una funcionalidad antes de haberla escrito le da un giro de 180 grados al código resultante.

Implementar el código que hace funcionar el ejemplo

Teniendo el ejemplo escrito, codificamos lo mínimo necesario para que se cumpla, para que la prueba pase. Comúnmente, el mínimo código es el de menor número de caracteres, porque mínimo quiere decir el que menos tiempo nos llevó escribirlo. No importa que el código parezca feo, eso lo vamos a enmendar en el siguiente paso y en las siguientes iteraciones. En este paso, la máxima es no implementar nada más que lo estrictamente obligatorio para cumplir la especificación actual. Y no se trata de hacerlo sin pensar, sino concentrados para ser eficientes. Parece fácil, pero, al principio, no lo es; veremos que siempre escribimos más código del que hace falta. Si estamos bien concentrados, nos vendrán a la mente dudas sobre el comportamiento ante distintas entradas, es decir, los distintos flujos condicionales que pueden entrar en juego; el resto de especificaciones de este bloque de funcionalidad. Estaremos tentados de escribir el código que los gestiona sobre la marcha y, en ese momento, solo la atención nos ayudará a contener el impulso y a anotar las preguntas que nos han surgido en un lugar al margen para convertirlas en especificaciones que retomaremos después, en iteraciones consecutivas.

Re factorizar

Re factorizar no significa reescribir el código; reescribir es más general que re factorizar. Según Martın Fowler, re factorizar es modificar el diseño sin alterar su comportamiento. En este tercer paso del algoritmo TDD, rastreamos el código (también el de la prueba) en busca de líneas duplicadas y las eliminamos refactorizando. Además, revisamos que el código cumpla con ciertos principios de diseño y re factorizamos para que así sea. Más allá de la duplicidad, durante la refactorización podemos permitirnos darle una vuelta de tuerca al código para hacerlo más claro y fácil de mantener. La clave de una buena refactorización es hacerlo en pasitos muy pequeños. Se hace un cambio, se ejecutan todos los tests

Cuando hemos dado los tres pasos de la especificación que nos ocupa, tomamos la siguiente y volvemos a repetirlos.

¿Y TDD sirve para proyectos grandes? Un proyecto grande no es sino la agrupación de pequeños subproyectos y es ahora cuando toca aplicar aquello de “divide y vencerás”. El tamaño del proyecto no guarda relación con la aplicabilidad de TDD. La clave está en saber dividir, en saber priorizar.

Índice de Figuras

Referencias Bibliográficas

- Sommerville, Ian - INGENIERÍA DE SOFTWARE - Novena Edición (Editorial Addison-Wesley Año 2011).
- Steve Mc Connell., DESARROLLO Y GESTIÓN DE PROYECTOS INFORMÁTICOS (Editorial McGraw Hill – Año 1996).
- Pressman, Roger - INGENIERÍA DE SOFTWARE, UN ENFOQUE PRÁCTICO. –(Sexta Edición -Editorial McGraw Hill – Año 2005)

- Myers, Glenford- El arte de Probar el Software. (Editorial El Ateneo, 1983).
- Cohn, Mike – Agile Estimation and Planning – Editorial Prentice Hall 2006.
- McConnell, Steve, Software Estimation: Demystifying the Black Art (Editorial Microsoft Press – Año 2006).
- Freeman, P. (1987). Software Perspectives: The System Is the Message. Addison Wesley.
- Jacobson, I. (1994). Object Oriented Software Engineering. Estados Unidos: Addison Wesley.
- Jacobson, I. (2000). El Proceso Unificado de Desarrollo. Madrid: Addison Wesley.
- No Silver Bullet (http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html)
- Dean Leffingwell and Pete Behrens – A user story primer (2009)
- Manifiesto Ágil http://agilemanifesto.org/iso/es/
- http://pgpubu.blogspot.com.ar/2007/01/tcnica-de-estimacin-wideband-delphi.html
- http://people10.com/blog/software-sizing-for-agile-transformation
- Software Program Manager Network - The Little Book of Software Configuration Management, (AirLie Software Council, 1998)- Sitio: http://www.spmn.com
- Gilb, T. and Graham, D., - Software Inspection, Addison-Wesley, 1993
- IEEE Std 1028-1988 –Standard for Software Reviews and Audits
- Bersoff, E.H., "Elements of Software Configuration Management", IEEE Transactions on Software Engineering, vol 10, nro. 1, enero 1984, pp 79-87
- Fowler, Martin – The new methodology http://martinfowler.com/articles/newMethodology.html
- Ken Schwaber & Jeff Sutherland - Scrum Guide - http://www.scrumguides.org/download.html
- Kent Beck, Extreme Programming Explained: Embrace Change, 1999.