Ingeniería del Software I

1 de diciembre de 2012

Índice

Introducción

La ingeniería del software se enfoca en el desarrollo de cualquier software y en todo el proceso que esto conlleva, incluyendo la obtención y formalización de lo que éste debe hacer.

Concepto: la ingeniería del software es la aplicación de un enfoque sistemático, disciplinado, y cuantificable al desarrollo, operación y mantenimiento del Software (IEEE).

Software: colección de programas, procedimientos, la documentación y datos asociados que determinan la operación de un sistema de computación.

¿Cómo desarrollar Software de nivel industrial?

1. El problema del Software

Si le preguntara a cualquier estudiante que haya tenido alguna experiencia en programación lo siguiente: si les diera un programa a estudiantes que su construcción demandara aproximadamente 10.000 líneas de código, en C o en Java, trabajando a tiempo completo, ¿Cuánto tiempo les tomaría?

La respuesta de los estudiantes varía en general entre 1 a 3 meses. Y dada la experiencia de programación de los estudiantes, hay una buena probabilidad de que puedan construir el software y la demostración a su profesor en 2 meses. Tomando 2 meses como el tiempo de terminación, la productividad del estudiante será 5000 líneas de código[1] por persona por mes.

Ahora tomemos un escenario alternativo, nosotros actuamos de clientes y tenemos el mismo problema y se lo encargamos a una empresa de desarrolladores de software. Como no hay una productividad estándar, es una medida razonable suponer una productividad de 1000 LOC por persona por mes. Aunque puede ser menor a 100 líneas de código por persona por mes en sistemas integrados, entonces a un equipo de profesionales en una empresa de software le demandará 10 personas por mes construir el programa en un mes.

Nos preguntamos ¿Por qué esta diferencia entre los dos escenarios? ¿Por qué esos mismos alumnos que pueden producir software con una productividad de unas pocas miles de LOC, cuando terminan sus estudios producen sólo cerca de 1000 LOC, al trabajar en una empresa?

La respuesta, del curso, es que hay dos diferentes pensamientos para construir software en los 2 escenarios, es decir lo que el estudiante construye y lo que la industria construye son dos cosas distintas. En el primero, **el sistema de los estudiantes** se construye pensando principalmente para ese fin específico y no se espera usarlo después, como no va a ser usado en un futuro no influye la presencia de bugs ni otras cosas concernientes.

En el segundo **el sistema de software de nivel industrial** está construido para solucionar algún problema del cliente, y va a ser usado por el cliente para realizar alguna tarea dentro de su empresa, y un error del sistema puede

[1] Lines of code: LOC

	Alumno	**Industria**
Usuario	el mismo	Terceros
Tolerancia de Error	Sí	No
Intefaz del Usuario	No necesaria	Importante
Documentación	No Existe	Requerida por el Usuario y para la organización y el Proyecto
Inversión	No Existe	Fundamental
Portabilidad Cofiabilidad Robustez	No es Importante	Característica Clave
Uso Crítico	No	Sí

Cuadro 1: Comparación entre software del estudiante y la industria

traerle un gran impacto en términos financieros o perder negocios, pérdida de clientes o perder una propiedad o incluso la vida. En consecuencia el software necesita tener una alta calidad con respecto a propiedades como **reliability**: confiablidad, **usability**: usabilidad, **portability**: portabilidad y otras.

La necesidad de una alta calidad para satisfacer a los usuarios finales tiene un mayor impacto en el camino de desarrollo del software y en su costo.

- Alta calidad requiere mucho testing que consume entre el 30 % y el 50 % del esfuerzo total a desarrollar.
- Requiere la descomposición en etapas del desarrollo de manera de poder encontrar "bugs" en cada una de ellas (cada uno de diferente índole).
- Para la misma funcionalidad se incrementa el tamaño y los requerimientos: Buena interfza de Usuario, Backup, tolerancia a fallas, acatar estándares, etc.

La regla de oro se rrompe cuando se sugiere que la industria de nivel insdustial del software puede costar cerca de 10 veces más que la industria de los estudiantes.

La industria del software está largamente interesada en desarrollar un software industrial-robusto y el área de la ingeniería del software se dedica a estudiar cómo construir esos sistemas. Este es, el dominio del problema para la ingeniería del software. A partir de aquí cuando digamos software, nos estaremos refiriendo sólo a un software de nivel industrial. En el resto del capítulo aprenderemos:

- Que la calidad, el costo y la planificación son los principales agentes que conducen a una industria robusta que desarrolle proyectos de software.

- Cuánto cuesta esa productividad está definida y medida para cada proyecto, y veremos cómo se puede caracterizar y medir la calidad del software.
- La escala y cambios son atributos importantes del problema y veremos una solución cercana para manejar ambos problemas.

1.1. Costo, Planificación y Productividad

Aunque la necesidad por una alta calidad que distingue a las industrias del software de otras, el costo y la planificación son principales razones que impulsaron dicho software. En la industria del software de nivel industrial entran en juego tres fuerzas básicas: el costo, la planificación y la calidad. El software debe ser producido a un costo razonable, en un tiempo razonable y debiera ser de buena calidad. Estos tres parámetros a veces conducen y definen un proyecto de software.

El software industrial es muy caro fundamentalmente porque se desarrolla en un extremo trabajo intensivo. Para tener una idea de los costos involucrados, vamos a considerar el corriente estado en práctica de la industria. Líneas de código (LOC) o miles líneas de código (KLOC) entregado, es por mucho, la medida más usada para ver el tamaño del software en la industria. Como el principal costo de producción de software es la mano de obra empleada, el costo de desarrollar software en general se mide en términos de persona/mes de esfuerzo gastado en el desarrollo. La productividad es frecuentemente medida en la industria en términos de LOC o KLOC por persona por mes.

La productividad en la indutria del software para escribir código fresco generalmente varía entre pocas cientas líneas y 1000 líneas de código por persona por mes. Esta productividad incluye el ciclo completo de trabajo, no sólo la tarea de escribir código. Las compañías de software suelen cobrar al cliente al que están desarrollando el software entre $3.000 y $15.000 por persona por mes, con esa productivad de 1000 LOC por persona/mes, eso hace que cada línea de código cueste entre $3 y $15, incluso pequeños proyectos pueden terminar con 50.000 LOC Con esta productividad los proyectos costarían entre $150.000 y $750.000.

La relación Hardware-Software en el costo de un sistema de computación se incrementó significativamente desde su comienzo.

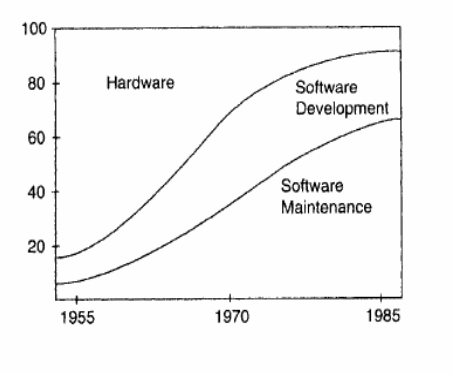

Figura 1: Relación entre el Hardware y el Software

Conclusión: el software es **muy** caro, es importante optimizar el proceso de desarrollo con el fin de disminuir su costo.

La **Planificación** es otro factor importante en muchos proyectos. Las tendencias de negocios están diciendo que el tiempo de comercialización de un producto debe ser reducido, de hecho el ciclo desde el concepto a la entrega suele ser pequeño. Para el software eso hace que se necesite desarrollar más rápido y con un tiempo específico. Desafortunadamente, la historia del software está llena de casos en que proyectos han sido terminados en un tiempo sustancialmente más tarde.

A pesar del progreso de la Ingeniería del Software, es aún un área débil. El 35 % de los proyectos del software se caracterizan como **runaway** (desbocados), es decir que tienen el presupuesto y el costo fuera de control.

Claramente, por esta razón reducir el costo y el ciclo para desarrollar el software son objetivos centrales para la ingeniería del software. La **productividad** en términos de output (KLOC) por persona/mes pueden adecuadamente capturar tanto el costo como lo que a la planificación concierne. Si la productividad es más alta, está claro que el costo en término de persona/mes bajará (el mismo trabajo puede ahora hacerse con menos personas). Similarmente, si la productividad es más alta el potencial de desarrollo de software en menor tiempo mejora, un equipo de más productividad puede terminar un trabajo en menor tiempo que uno de igual tamaño con menor productividad. (El tiempo que tomará un proyecto también depende de la cantidad de personas avocadas al mismo). Por lo tanto la búsqueda de una alta productividad es básicamente una fuerza motriz detrás de la ingeniería del software y una mayor razón para usar distintas herramientas y técnicas.

Además del costo y la planificación, otro factor de mayor impulso a la ingeniería del software es la **calidad**. Hoy, la calidad es una de las principales inspiraciones y estrategias de negocios están diseñadas a su alrededor. Desafortunadamente, una gran lista de casos se han producido en relación con la falta de confianza del software. El software a veces no hace lo que debería o hace lo que se supone no debiera. Claramente, desarrollar software de alta calidad es otro objetivo fundamental de la ingeniería del software. Sin embargo, es fácil entender la idea de costo, pero el concepto de **calidad** en el contexto del software necesita mucha más elaboración.

Las fallas del software son distintas de las fallas mecánicas o eléctricas. En el software, las fallas **no** son consecuencia del uso y el deterioro. Las fallas ocurren como consecuencia de errores (o "bugs") **introducidos** durante el desarrollo. Es decir la falla que causa el problema existe desde el comienzo, sólo que se manifiesta tarde.

El estándar internacional de calidad del software sugiere que la calidad comprende 6 principales atributos, mostrados en la figura:

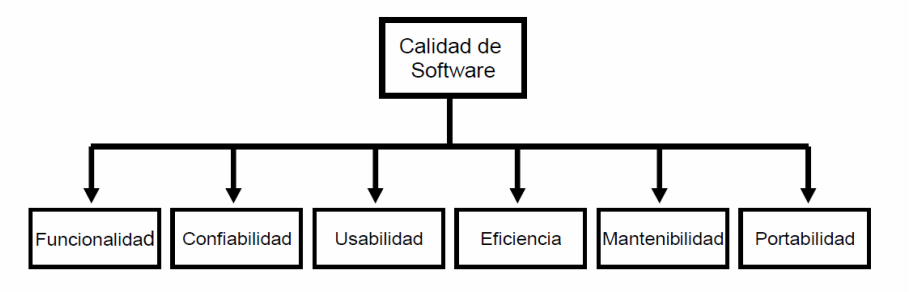

Figura 2: Atributos de la Calidad del Software

- Funcionalidad: la capacidad de proveer funciones que cumplen las necesidades establecidas o implicadas.
- Confiabilidad: la capacidad de realizar las funciones requeridas bajo las condiciones establecidas durante un tiempo específico.
- Usabilidad: la capacidad de ser comprendido, aprendido y usado.
- Eficiencia: la capacidad de proveer una apropiada performance relativa a los recursos que utiliza.
- Mantenibilidad: la capacidad de poder ser modificado con algún propósito.
    - Correcciones.
    - Mejoras (Improvements).

- Adaptaciones.

- Portabilidad: la capacidad de poder ser adaptado a diferentes ambientes sin aplicar otras acciones que las provistas a este propósito en el producto.

Con múltiples dimensiones de calidad, los diferentes proyectos enfatizan en distintos atributos y no es posible una definición simple de calidad. Sin embargo, **la confianza es aceptada como un criterio acorde de calidad**.

Una medida de calidad es el número de defectos en el software por unidad (generalmente tomado en cantidad de errores por KLOC). Las mejores prácticas intentan reducir al error por KLOC como certificado de calidad.

Entonces para determinar la calidad de un programa, debemos determinar el número de defectos escritos en el código. Este número claramente no se conoce, y muchas veces nunca se puede llegar a conocer. Un acercamiento para medir la calidad es cuántos errores se encuentran a lo largo de su uso durante 6 meses o un año.

En este punto no está clara la definición de **defecto**. Un defecto puede tomarse como un problema en el software que cause que se rrompa el programa o que devuelva un output no adecuado a los requerimientos. Una definición exacta de lo que consideramos defecto depende del proyecto o de los estándares.

Consistencia y Repetitibilidad

Algunas veces un grupo puede desarrollar un buen sistema de software. El desafío clave en la Ingeniería del Software es cómo asegurar que el éxito pueda repetirse (con el fin de mantener alguna consistencia en la calidad y la productividad). Por lo tanto un objetivo de la ingeniería del software es la **sucesiva producción** de sistemas de alta calidad y con alta productividad. La consistencia permite predecir el resultado del proyecto con certeza razonable.

Otro aspecto de importancia es la **mantenibilidad**. Una vez que un proyecto es entregado, entra en una fase de mantenibilidad. ¿Por qué el mantenimiento es necesario para el software si el software no se degrada con los años? El software necesita ser mantenido, porque siempre hay un error residual en el sistema. Defectos una vez descubiertos, necesitan ser removidos, son los que se llaman **mantenimientos correctivos (updates)**. A su vez el mantenimiento es necesario para satisfacer otras necesidades del usuario o del ambiente, a esto se conoce como **mantenimiento adaptativo (upgrades)**. Muchas veces los costos/ganancias de mantenimientos son mayores al precio original del software. Al ser necesarios es preferible tener un software fácil de mantener. Incluye la comprensión del software existente (código y documentación), comprensión de los efectos del cambio, realización de los cambios (código y documentación) testear lo nuevo y re-testear lo viejo.

1.2. Escala y Cambios

Aunque el costo, calidad y planificación son los principales actores a tener en cuenta por la ingeniería del software hay otros como la **escala** y el **cambio**.

Size(KLOC)	Software	Lenguajes
980	Gcc	ansic,cpp,yacc
320	Perl	perl,ansic,sh
200	Openssl	ansic,cpp,perl
100	Apache	ansic,sh
65	SendMail	ansic
30,000	Red Hat Linux	ansic,cpp
40,000	Windows XP	ansic,cpp

Cuadro 2: Ejemplos de Software conocido y cantidad de líneas de código.

Muchos software de las industrias grandes del software tienden a ser enormes y complejos necesitando decenas de miles de LOC, como es de esperarse, el desarrollo de un sistema grande requiere un diferente conjunto de métodos distintos a los necesarios para desarrollar un pequeño sistema, como así los métodos que son usados para desarrollar pequeños sistemas no se pueden escalar para grandes.

Podemos dar el siguiente ejemplo, "contar las personas de una habitación contra realizar el censo de un país". En ambos problemas lo esencial es el conteo. Pero los métodos usados para contar personas en una habitación no funcionarán cuando tomemos un censo.

Otro conjunto, diferente, de métodos necesitaremos para realizar un censo, y el problema del censo va a requerir mucha más administración, organización y validación sumados al tiempo del conteo.

De la misma manera métodos que uno puede usar para desarrollar programas de pocos centenares de líneas no podemos esperar que funcionen para un software de unas pocos centenas de miles de líneas. Un diferente conjunto de métodos debe usarse para desarrollar un software "largo". Cualquier proyecto de software involucra el uso de ingeniería y la gestión del proyecto. En los pequeños proyectos, métodos informales para el desarrollo y gestión pueden ser usados. Sin embargo, para grandes proyectos, ambos deben ser rigurosos como lo ilustra la figura 3.

En otras palabras, para ejecutar un proyecto satisfactoriamente un adecuado método para la ingeniería del sistema tiene que ser empleado y el proyecto tiene que ser manejado estrechamente para asegurarnos, que el costo, planificación y la calidad estén bajo control.

A gran escala una característica clave del dominio del problema y la solución cercana debe emplear herramientas y técnicas que se caracterizan por permitir construir grandes software.

El **cambio** es otra característica del problema de dominio que el enfoque para el desarrollo debe manejar. Como el completo conjunto de requerimientos del sistema en general no lo conocemos al empezar o durante el desarrollo agregamos otros requerimientos. A medida que los vamos identificando, esto

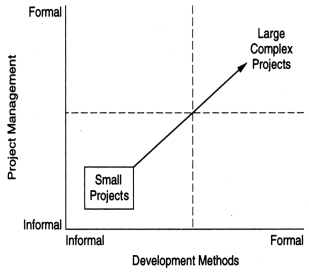

Figura 3: El problema de la Escala

hace que debamos agregarlos en el software en pleno desarrollo. Esta necesidad de cambios requiere que los métodos para el desarrollo se adapten a cambios eficientemente. El cambio de los pedidos puede ser muy perjudicial para el proyecto, y sino se manejan apropiadamente, pueden llegar a consumir por encima del 30-40 % del costo total del desarrollo.

Como se discutió anteriormente, el software tiene que poder ser cambiado incluso hasta después de haber sido entregado aunque tradicionalmente los cambios en el software durante el mantenimiento tienen que distinguirse de los cambios que ocurren mientras se está desarrollando el software esas líneas están "bluring" (desdibujadas) como fundamentalmente los cambios en ambos escenarios son similares: "código existente necesita ser cambiado debido a cambios en el requerimiento o debido a algún defecto que deba ser removido".

Universalmente como el mundo cambia más rápido, el software va a tener que adaptarse, incluso mientras esté en desarrollo. Los cambios en el requerimiento son por lo tanto una característica del dominio del problema. Actualmente, software que no pueda aceptar y acomodarse a cambios es poco usado, ya que sólo puede resolver un pequeño conjunto de problemas que resistan el cambio.

El cambio también afecta a las empresas/instituciones.

Ya comprendemos el dominio del problema y los factores que motivan la ingeniería del software.

- Consistentemente desarrollar software de alta calidad y con alta productividad (C&P) para problemas de gran escala que se adapten a cambios.
- C&P son los objetivos básicos a perseguir bajo gran escala y tolerancia a cambios.
- A su vez C&P son consecuencia de la gente, los procesos y la tecnología.

1.3. Resumen

- El dominio de problema para la ingeniería del software es el software robusto de nivel industrial. Este software significa resolver algún problema para un conjunto de usuarios, y se espera que la solución sea de alta calidad.

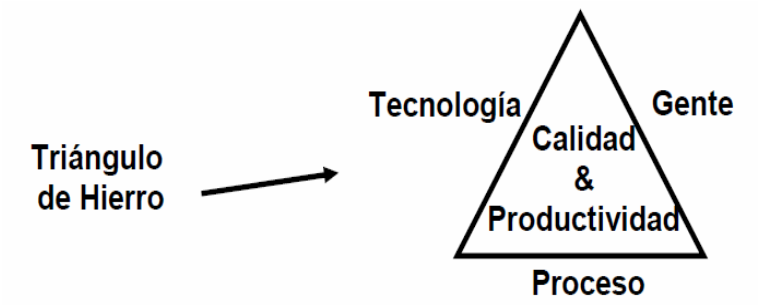

Figura 4: Relación entre Tecnología, Proceso y Gente

- En el dominio de este problema, el costo, la planificación y la calidad son las fuerzas motrices básicas. Por lo tanto, métodos y herramientas que serán usadas para resolver problemas de nuestro dominio deben garantizar alta productividad y alta calidad.
- La productividad se mide como la cantidad de producción por unidad de recurso de entrada. En el software, la producción puede medirse en términos de líneas de código y el tiempo humano es el principal recurso, el recurso de entrada puede medirse como persona/mes. La productividad puede por esto ser medida como las líneas de código producidas por persona por mes.
- Un software de calidad tiene muchos atributos entre los cuales incluye a:
  - Funcionalidad
  - Confiabilidad
  - Usabilidad
  - Eficiencia
  - Mantenibilidad
  - Portabilidad

  Confiabilidad (Reliability) es a veces considerada como el principal atributo de la calidad y como la desconfianza en el software está causada por defectos en el mismo, la calidad puede ser caracterizada por el número de defectos por KLOC.
- El problema en este dominio suele a veces tender a ser grande y donde las necesidades del cliente cambian rápido. Por lo tanto las técnicas usadas para desarrollar un software de nivel industrial deben ser capaces para construir un gran sistema de software y tienen que ser capaces de manejar los cambios.

Ejercicios

1. ¿Cuál es la principal diferencia entre un software de un estudiante y un software de nivel industrial?

2. Si el desarrolo para solucionar un problema requiere un esfuerzo E, está estimado que la industria del software va a necesitar 10E para resolverlo. ¿Dónde crees que este esfuerzo extra es gastado?

3. ¿Qué lineamientos tomarías para medir la productividad? ¿Cómo determinarías la productividad con esas medidas?

4. ¿Cuáles son los diferentes atributos de calidad del software? Si por alguna razón necesitaríamos focalizarnos en obtener un software sin sorpresas en cuál atributo deberíamos focalizar?

5. ¿Qué diferencia un pequeño software de uno grande? ¿Cómo crees que la ejecución de una misma tarea cambia?

6. Suponga que los cambios que requerimos en un software. ¿Por qué son mucho más costosos si sólo cambio líneas de código existentes?

2. El proceso del Software

La ingeniería del software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.

Por enfoque sistemático nos referimos a metodologías y prácticas existentes para solucionar un problema dentro de un dominio determinado. Esto permite repetir el proceso y da la posibilidad de predecir los resultados (independientemente del grupo de personas que lo lleve a cabo).

Ahora que entendemos un poco más acerca del problema que focaliza la ingeniería del software y cómo intenta solucionarlo, vamos a orientar la discusión a la ingeniería del software propiamente dicha.

La ingeniería del software está definida como lineamientos sistematizados para el desarrollo, operación, mantenimiento y entrega del software.

Hemos visto que dentro del desarrollo del software, la alta calidad, el bajo costo y el corto ciclo de trabajo son referencias que la ingeniería del software debe atender. En otras palabras la sistematización debe ayudar a llegar a una alta calidad y productividad (Q&P). En el software, hay principalmente 3 factores que influyen en la calidad y productividad: la gente, el proceso y la tecnología. Es decir la calidad y la productividad final obtenida depende de las habilidades de las personas involucradas en el proyecto del software, los procesos de personas usadas para mejorar la performance de las diferentes tareas del proyecto, y de las herramientas usadas.

Como se trata de personas que en última instancia desarrollan y entregan un producto, (y la productividad está medida con respecto al esfuerzo de ellos frente al input), el principal trabajo del proceso es ayudar a la gente a lograr un Q&P mejor especificando qué tareas hacer y cómo hacerlas.

Las herramientas se brindan para permitir a las personas realizar una tarea más eficientemente y con menor cantidad de errores. Debiera por esta razón ser claro el objetivo a satisfacer hacia el software con alta Q&P, el procedimiento ser el núcleo. Consecuentemente, en la ingeniería del software, enfoca principalmente en los procesos que están referidos a lograr una sistematización. Es este enfoque en donde los procesos distinguen a la ingeniería del software de otras disciplinas de la computación (generalmente focalizados en el producto). Entonces podemos sintetizar diciendo que la ingeniería del software focaliza en los procesos para producir los productos.

Como los procesos son el corazón de la ingeniería del software con herramientas y tecnologías provistas como soporte para una ejecución eficiente de los procesos, este libro focaliza primariamente en los procesos. En este capítulo discutiremos acerca de:

- El rol de los procesos y modelos de procesos en un proyecto.
- Varias componentes del proceso del software y el rol del desarrollo y su administración.
- Varios modelos para desarrollar procesos.
    - “Waterfall”: cascada.

- "Prototyping": prototipado.
- "Iterative": iterativo.
- "RUP": Rational Unified Process.
- "Timeboxing": Cajas de tiempo.
- "XP": Extreme Programming.

- La estructura general de la administración del proceso y sus principales fases.

2.1. Proceso y Proyecto

Un proceso es una secuencia de pasos realizados con algún propósito determinado. Como se mencionó anteriormente, mientras desarrollamos software de nivel industrial, la finalidad es el desarrollo de software para satisfacer la necesidad de algunos usuarios o clientes como lo muestra la figura:

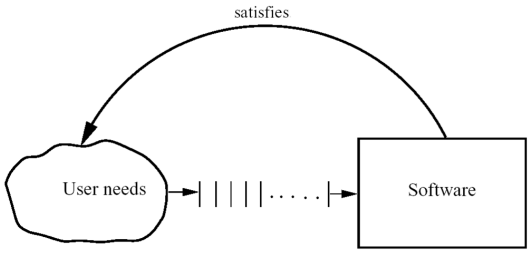

Figura 5: El problema Básico

Un proyecto de software es una instancia del problema y el desarrollo del proceso es lo que usamos para alcanzar ese propósito.

Entonces, para desarrollar un proyecto el proceso juega un rol clave. Es decir, siguiendo el proceso deseado al final de la meta el software se consigue. Sin embargo, una vieja discución, no es suficiente para llegar justamente al software deseado, pero queremos que el proyecto termine a un bajo costo en un corto ciclo de trabajo y entregar un software de alta calidad. El rol del proceso se implemente debido a esas metas, y sin embargo muchos procesos pueden alcanzar sólo lo básico de desarrollar un software de la figura. Para alcanzar Q&P necesitamos un proceso óptimo. Es este objetivo que hace del diseño del proceso un desafío.

Debemos distinguir la especificación del proceso o descripción del proceso mismo. El **proceso** es una entidad dinámica que englobará todas las acciones realizadas. La **especificación del proceso**, por otra parte, es una descripción del proceso que probablemente puede ser seguido en algún proyecto para alcanzar el objetivo para el cual fue diseñado.

En un proyecto, la especificación del proceso puede ser tomada como el plan a seguir. El proceso actual es lo que se está haciendo en el proyecto. Se observa que el actual proceso puede ser diferente al proceso planeado, y les aseguro que seguir la especificación del proceso no es un problema trivial. Sin embargo, en este libro, vamos a asumir que el proceso planeado y el actual son los mismos sin distinciones entre ambos y emplearemos la palabra proceso para referirnos a ambos.

Un modelo de proceso especifica un proceso general, que es "óptimo" para una clase de proyectos. Es decir, en situaciones para los que el modelo es aplicable, aplicar el proceso nos llevará a cumplir la meta de desarrollar un software con alta Q&P. Un modelo de proceso es esencialmente una recopilación de las mejores prácticas dentro de un "recipiente" para terminar satisfactoriamente un proyecto. En otras palabras, un proceso es el medio para alcanzar la meta de la alta calidad, bajo costo y en un corto ciclo de trabajo; y un modelo de proceso provee una estructura bien catalogada para una clase de proyectos.

Un proceso es a veces especificado a un alto nivel como una secuencia de etapas. La secuencia de pasos dentro de una etapa, a veces suele ser considerado como un subproceso.

2.2. Componentes del Proceso del Software

Por la definición anterior, un proceso es una secuencia de pasos ejecutados para alcanzar el objetivo. Desde que hay muchos objetivos y diferentes por satisfacer mientras desarrollamos software, muchos procesos son necesarios. Muchos de los cuales no entran dentro de la ingeniería del software, sin embargo impactan en el desarrollo del software, pueden ser clasificados como "**nonsoftware process**". Procesos de negocios, sociales y de entrenamiento, son ejemplos de esta clase. Los procesos que tratan con técnicas y la administración de reglas del desarrollo de software son llamados los procesos del software.

Un proyecto exitoso es el que satisface las expectativas en costo, tiempo y calidad. Al planear y ejecutar un proyecto de software, las decisiones se toman con el fin de reducir costos y tiempos e incrementar la calidad.

Como un proyecto de software va a tener una ingeniería para manejar y solucionar adecuadamente el proyecto, están dados los dos mayores componentes en un proceso de software "**development process**" y un "**project managment process**". El desarrollo de los procesos especifica todas las actividades de la ingeniería a ser realizadas, y el proceso de Managment especifica cómo planear y controlar las actividades, a qué costo, calidad y otros objetivos. Un efectivo desarrollo y proyecto de managment son claves para alcanzar los objetivos de la realización de software que responda a las necesidades del cliente con una alta calidad y productividad.

Durante la ejecución del proceso muchos productos están compuestos de lo que típicamente se conoce como items (por ejemplo el código fuente generalmente está constituido por varios archivos). Estos items van evolucionando junto con el proceso, creando muchas versiones en el camino. Existen **software**

configuration control que son los encargados de administrar dichas versiones, ejmplos: svn y trac.

Estos tres procesos constituyentes se centran en los proyectos y los productos y se puede considerar que comprenden el proceso de la ingeniería del producto, y su principal objetivo es producir el deseado producto. Si el proceso del software puede ser visto como una entidad estática, estas tres componentes del proceso serán suficientes. Sin embargo, el proceso del software propiamente dicho es una entidad dinámica, ya que deben cambiar para adaptarse a nuestra comprensión acerca del desarrollo del software y la posibilidad de nuevas tecnologías y herramientas. Debido a esto es necesario un proceso para manejar el proceso del software.

El objetivo básico del proceso de managment es mejorar el proceso del software, con mejorar nos referimos a la capacidad del proceso de producir una buena calidad a un bajo costo. Por esta razón se estudia el proceso actual del software, frecuentemente estudiando los proyectos que han usado algún proceso.

La relación entre los principales componentes del proceso se muestran en la figura:

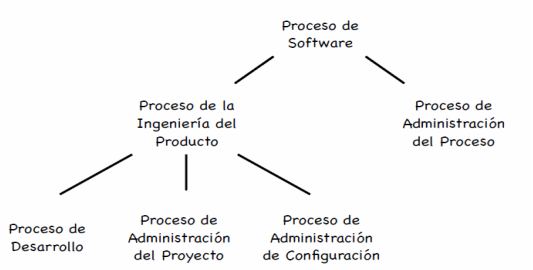

Figura 6: Los procesos del Software

Estos componentes del proceso se distinguen no sólo en el tipo de actividades que realizan, sino también en las personas que trabajan específicamente en el proceso. En un típico proyecto las actividades de desarrollo son realizadas por programadores, diseñadores, testers, etc. En cuanto al proyecto de Managment, las actividades de configuration control process y las actividades de managment son realizadas por el Software Enginering Process Group (SEPG).

En este libro nos enfocaremos fundamentalmente en los procesos relacionados con productos de la ingeniería, particularmente los procesos de desarrollo y el manejo del proyecto. Mucho de este libro discute diferentes formas de desarrollar el proceso en fases y los subprocesos o metodologías usadas para ejecutar estas fases. Por el resto del libro, usaremos el término proceso del software para referirnos al producto de la ingeniería de los procesos.

2.2.1. El enfoque ETVX

ETVX: Entry Task Verification Exit

- Criterio de entrada: qué condiciones deben cumplirse para iniciar la fase.
- Tarea: lo que debe realizar esa fase.
- Verificación: las inspecciones/controles/revisiones/verificaciones que deben realizarse a la salida de la fase (es decir al producto de trabajo).
- Criterio de salida: cuándo puede considerarse que la fase fue realizada exitosamente.

Además cada fase produce información para la administración del proceso.

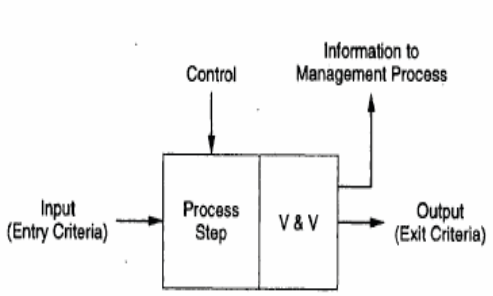

Figura 7: ETVX

El criterio de entrada de una fase debe ser consistente con el criterio de salida de la fase anterior.

2.2.2. Características deseadas del proceso del software

La idea central incluye alcanzar una alta calidad y productividad, para esto debe:

- Producir software testeable: testing es la tarea más cara dentro del proceso de desarrollo, entre 30 % y 50 % del esfuerzo total del desarrollo.
- Producir software mantenible: el mantenimiento puede ser más caro que el desarrollo, hasta 80 % del costo total durante la vida del software.
- Eliminar defectos en etapas tempranas: el costo de eliminar un defecto se incrementa a medida que perdura en el proceso de desarrollo.
- Ser predecible y repetible: los procesos deben conseguir repetir el desempeño cuando se utilizan en distintos proyectos. Es decir, el resultado de utilizar un proceso debe poder predecirse.

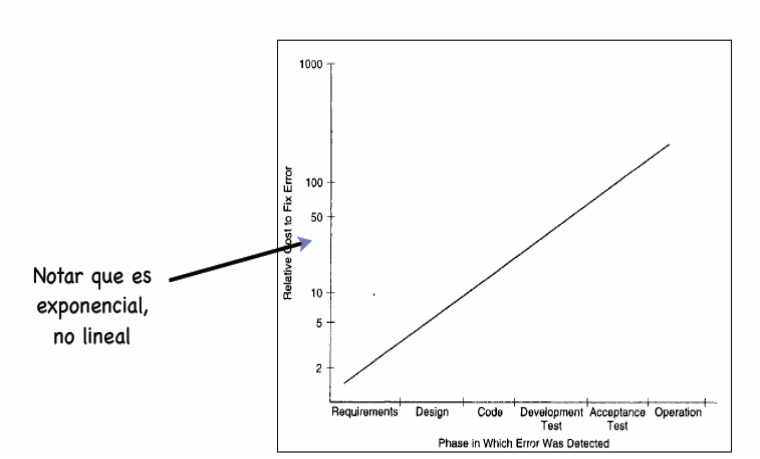

Figura 8: Relación Costo Correción - fase

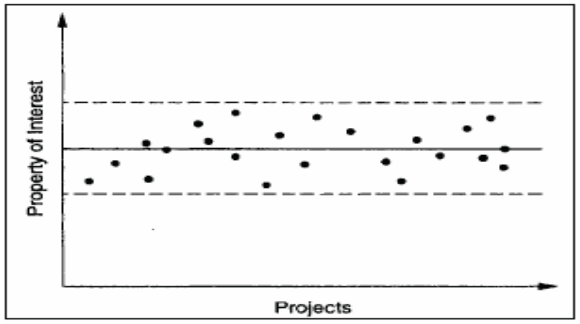

Figura 9: Control estadístico de los procesos

"Procesos similares desarrollados bajo el mismo proceso tendrán costos similares"

Un proceso predecible se dice que está bajo control estadístico.

- Soportar cambios y producir software que se adapte a cambios: el cambio de requerimientos (por diversos motivos) es una razón prominente. Los cambios en los requerimientos son esperables y no pueden tratarse como algo "malo". Además de cambiar software en operación, los cambios pueden tomar lugar durante el desarrollo entonces todo proceso de desarrollo de software debe dejar lugar para cambios y tratarlos apropiadamente.

2.3. Modelos de desarrollo del proceso del Software

Para el desarrollo del proceso del software, el objetivo es producir un software de alta calidad. Es por esto que enfoca sus actividades directamente a lo relacionado con la producción de software, por ejemplo: diseño, codificación y testing. La administración del proceso a veces se decide basada en el desarrollo del proceso.

El proceso de desarrollo define las tareas que el proyecto debe hacer y el orden en que deben terminar. El proceso limita el grado o la libertad para un proyecto especificando el tipo de actividad que debe hacerse y en qué orden, de manera tal que se logre el más corto/eficiente camino para satisfacer las necesidades del cliente. El proceso **impulsa** un proyecto que **influye fuertemente** en el resultado.

El proceso de desarrollo de software es un conjunto de fases, cada fase es a su vez una secuencia de pasos que definen la metodología de la fase. ¿Por qué utilizar fases? "divide y vencerás", es decir cada fase ataca distintas partes del problema esto ayuda a validar continuamente el proyecto.

Usualmente está compuesto de las siguientes actividades:

- Análisis de requerimientos y especificaciones.
  - Objetivo: comprender precisamente el problema.
  - Formar las bases del acuerdo entre el cliente y el desarrollador.
  - Especificar el **qué** y **no** el cómo.
  - Salida: SRS (especificación de los requisitos del software).
- Arquitectura y diseño.
  - Objetivo: paso fundamental para moverse del dominio del problema al dominio de la solución.
  - Involucra 3 etapas:
    1. Diseño arquitectónico.
    2. Diseño de alto nivel.
    3. Diseño detallado.
  - Salida: documentos correspondientes.
- Codificación.
  - Convierte el diseño en código escrito en un lenguaje específico.
  - Objetivo: implementar el diseño con código simple y fácil de comprender (legible!).
  - Incluye el testing.
  - Salida: el código.
- Testing.

- Objetivo: identificar la mayor cantidad de defectos.
- Es una tarea muy cara: debe planearse y ejecutarse apropiadamente.
- Salida: plan de test conjuntamente con los resultados, y el código final testeado (y confiable).

- Entrega e instalación.

Distribución de esfuerzos

Análisis de requisitos	10-20 %
Diseño	10-20 %
Codificación	20-30 %
Testing	30-50 %

Cuadro 3: El testing es la fase más demandante.

Discutimos anteriormente, que un modelo de proceso especifica un proceso general, usualmente como un conjunto de etapas en las que debe dividirse el proyecto, el orden en que esas etapas deben ser ejecutadas y cualquier otra restricción y condición en la ejecución de las etapas. La premisa básica detrás de un modelo de proceso es "en situaciones en las que el modelo es aplicable hacerlo nos llevará a un bajo costo, alta calidad y a reducir el ciclo de trabajo, o nos proveerá otros beneficios". En otras palabras, un modelo de proceso provee lineamientos generales para desarrollar un adecuado proceso para un proyecto.

Debido a la importancia del development process, se han propuesto varios modelos, en esta sección discutiremos algunos de los más importantes y conocidos.

2.3.1. Modelo Cascada (Waterfall)

El modelo de proceso más simple es el modelo cascada, en el cual las fases están organizadas en un orden lineal. En este modelo el proyecto empieza con un análisis práctico. Una vez llevado a cabo la practicidad del proyecto con éxito, empieza el análisis de los requerimientos y empieza la planificación. El diseño empieza después de que acaba el análisis de los requerimientos, y la codificación luego de que concluye el diseño. Una vez que el programa es completado, se lleva a cabo una fase de testing, luego de que se testea satisfactoriamente el sistema es instalado. Luego de esto regularmente comienza una fase de mantenimiento.

La idea básica detrás de la separación de las fases es separar las preocupaciones, cada fase trata con un conjunto distinto y separado de preocupaciones. Haciendo esto la larga y compleja tarea de construir el software se divide en pequeñas tareas (que por si mismas son complejas). Separando los problemas y enfocados en una selección de pocas tareas se facilita a los ingenieros y Managers a tratar con la complejidad de problema.

La fase de análisis de los requerimientos se mencionó como **análisis y Planificación**. La planificación es una actividad crítica en el desarrollo del software.

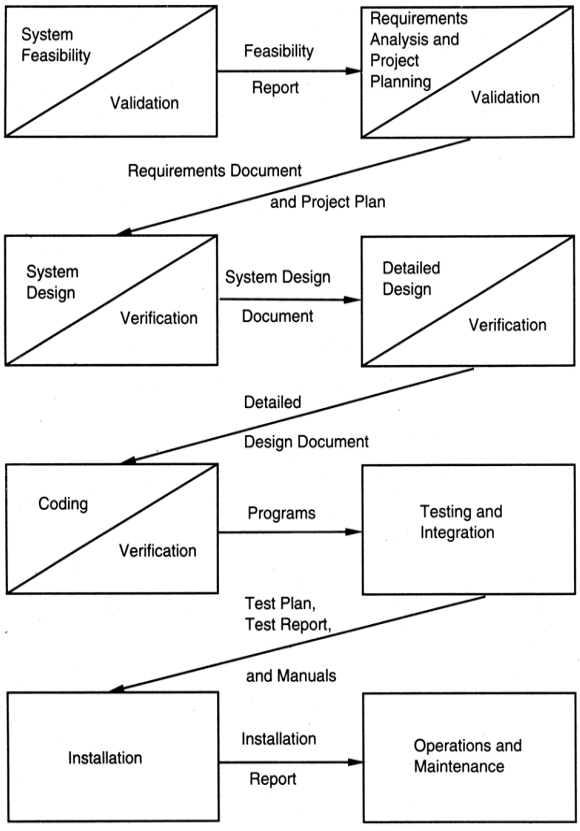

Figura 10: El modelo Cascada

Un buen plan está basado en los requerimientos del sistema y debe ser realizado antes de que la siguiente fase empiece. Sin embargo, en la práctica, una lista detallada de los requerimientos no es necesaria para la planificación. En consecuencia la planificación usualmente se superpone con el análisis de los requerimientos y el plan está listo antes de que la siguiente fase empiece. Este plan es un input adicional para todas las siguientes fases. El orden lineal de las actividades trae importantes consecuencias.

Primero se identifica claramente el comienzo y fin de cada fase, cierta certificación debe realizarse para terminar una fase, generalmente se realiza una certificación y validación que asegure que el output de una fase sea consistente con el input, y que el output de la fase sea consistente con todos los requerimientos del sistema.

La consecuencia de la necesidad de una certificación es que cada fase debe

tener un definido output que pueda ser evaluado y certificado. Es decir, cuando las actividades de una fase están completas, debe haber algún producto para cada fase, los datos de las fases tempranas a veces suelen ser llamados **work products**.

Documentación realizada

1. Documentación de los Requerimientos.
2. Plan del proyecto.
3. Documentos de diseño (Arquitectura, detalles del sistema).
4. Plan de tests y reporte de los tests.
5. Código final.
6. Manuales del Software (usuario, instalación , ...).

La ventaja más importante que tiene es la simplicidad, podría pagarse por la finalización de cada etapa y tengo una detección simplificada de los culpables en caso de error y puedo remunerar fácilmente a los desarrolladores de cada fase, por lo que es fácil de administrar.

Limitaciones

1. Asume que los requerimientos se van a congelar.
2. Congelar los requerimientos implica una elección de hardware.
3. Todo o Nada, trae problemas para financiamientos inestables.
4. Necesita fuertemente de los requerimientos.
5. Está manejado por la documentación, para empezar o terminar una fase.

Conclusión

Es el modelo más usado, es muy útil para proyectos en los cuales los requerimientos están bien entendidos, fases claras, es un modelo familiar y funciona bien para procesos de requerimientos claros y entendidos se concluye en un modelo de proceso eficiente.

2.3.2. Prototipado

El objetivo del modelo prototipado es tener en cuenta la primer limitación del modelo cascada. Aquí la idea básica es construir un prototipo descartable para ayudar a entender los requerimientos. El prototipo está basado en los requerimientos que se conocen. El desarrollo del prototipo se somete a diseño, codificación y testing, pero cada una de las fases no se hace muy formalmente o a fondo. Usando un prototipo, el cliente puede obtener una sensación real del

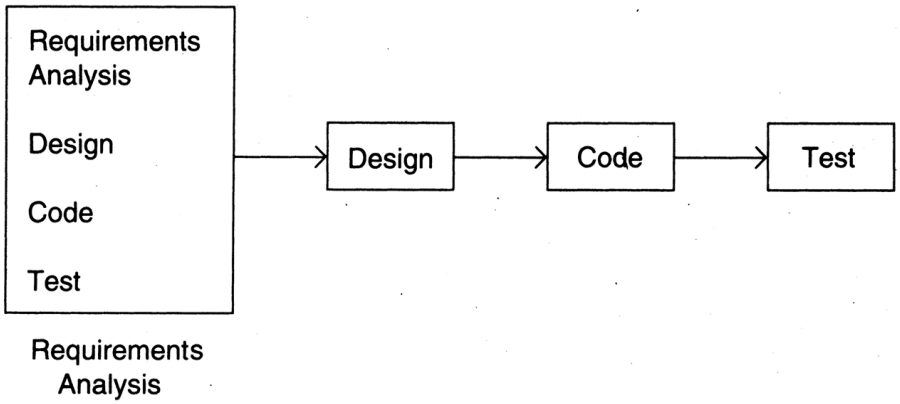

Figura 11: El modelo prototipado

sistema, que puede permitirle un mejor entendimiento de lo que quiere para el sistema. El resultado es más estable cuando cambian con frecuencia.

Prototipado es un buen modelo cuando no hay manuales o elementos pre-existentes, o no están claros los requerimientos.

Ayuda al cliente dejándolo "jugar" con el prototipo, también es efectivo para demostrar la confiabilidad con un acercamiento al sistema.

Luego de la realización del prototipo se le da la oportunidad de explicar al usuario final que no tenía en claro lo que quería.

Basado en la experiencia, y la respuesta observada en el prototipo, se puede determinar qué es correcto, qué debe ser modificado o qué falta o es innecesario.

Luego de las sugerencias el proyecto puede realizarse de manera más simple.

En el modelo prototipado, como el prototipo es descartado, sólo se focaliza en las partes que no entendemos correctamente (los grises).

El prototipo se hace rápido y "suciamente" se focaliza en hacerse rápido y no en la calidad.

Se reduce ampliamente en el prototipado el testing porque suele ser la parte más demandante.

1. Se intenta hacer el prototipo lo más barato posible y de él extraer gran parte de experiencia para reducir el costo del trabajo final.

2. Logramos unos requerimientos más estables, por el feedback del cliente.

3. Se obtiene un mejor código final basado en la experiencia y gusto.

4. Se elimina cualquier tipo de riesgo donde los objetivos no son claros.

Conclusión

En general prototipado es adecuado para proyectos donde los requerimientos son difíciles de codificar y la confianza en los requerimientos establecidos es baja. En proyectos donde los requerimientos no son correctamente establecidos en el comienzo, usar el modelo prototipado puede ser el método más efectivo para el desarrollo del software. También es una técnica excelente para reducir algún tipo de riesgos asociados con el proyecto.

2.3.3. Desarrollo iterativo

El modelo de desarrollo iterativo tiene en cuenta la tercer y cuarta limitación del modelo cascada ("todo o nada", "necesita fuertemente de los requerimientos") e intenta combinar los beneficios de ambos (prototipado y cascada).

La idea básica es que el software debe ir desarrollándose de manera incremental donde cada incremento agregue una funcionalidad o una capacidad hasta que el sistema final es implementado.

La idea es crear una **lista de control** del proyecto con distintas funcionalidades en orden, esto da una idea de:

- Lo siguiente que debe hacerse.
- Cuánto falta para el trabajo final.

Cada paso consiste en remover la siguiente tarea de la lista.
En 3 fases:

1. Diseño.
2. Implementación.
3. Análisis.

El ciclo continúa hasta que la lista de control se vacía.

El análisis de cada paso puede llevar a rediseñar el sistema o alguna componente, pero a medida que vaciando la lista debemos llegar a un sistema más estable y no hay posibilidad de rediseñar el sistema.

Un paso sólo debe terminarse cuando se comprenda y realice íntegramente.

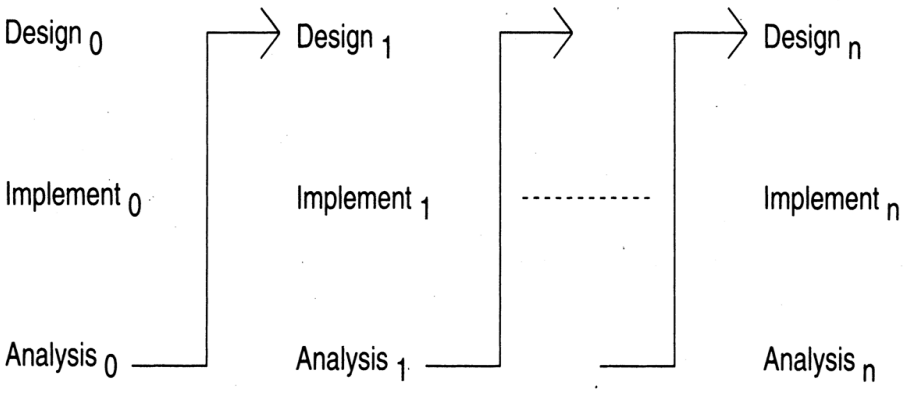

Figura 12: El modelo iterativo

Beneficios:

- Permite cambiar fácilmente los requerimientos.
- No tiene el riesgo de todo o nada.

Costos:

- El diseño del sistema no es robusto.
- Volver a trabajar sobre una zona o descartar trabajo.

Permite ir mostrando al cliente distintas versiones y obtener feedback.

Otro enfoque común para el modelo de desarrollo iterativo es hacer los requerimientos y el diseño con cascada o algo más cercano al prototipado, entregando software iterativamente. Es decir, la construcción del sistema, que es la tarea que más esfuerzo consume, se realiza iterativamente.

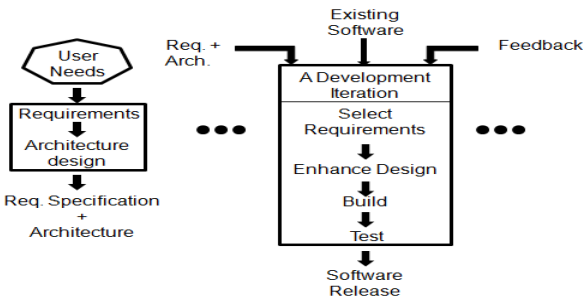

Figura 13: Modelo de entregas Iterativas

La ventaja de esto es que si la mayoría de los requerimientos son conocidos por adelantado, una estructura general del sistema puede ser desarrollada agregando funcionalidad a algo estable. A su vez cada entrega del software al finalizar un paso de iteración puede recibir un feedback del cliente lo que a posteriori puede ser agregado al sistema.

Modelo en espiral

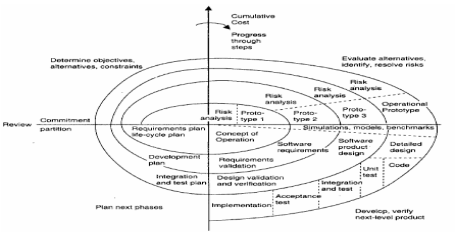

Figura 14: Modelo espiral

En cada ciclo de la espiral se realizan los siguientes pasos:

1. Identificar objetivos, distintas alternativas para conseguirlos y restricciones.
2. Evaluar alternativas en base a objetivos y restricciones (considerar riesgos). Desarrollar estrategias para resolver incertidumbres y reducir riesgos.
3. Desarrollar y verificar el software.
4. Planear el próximo paso.

Conclusión desarrollo iterativo

Razones por la que este modelo se está popularizando:

1. Los clientes no quieren invertir sin obtener nada, actualmente prefieren ir viendo continuamente los cambios y avances.
2. Como los trabajos o negocios cambian rápidamente, los clientes nunca conocen en realidad por completo lo que quieren, y esta necesidad constante de agregar nuevas capacidades para adaptarse a las cuestiones de negocios, el modelo iterativo permite esto.
3. Cada iteración recibe una respuesta del cliente que nos ayuda a desarrollar requerimientos más estables para la siguiente iteración.

2.3.4. Rational Unified Process

Es otro modelo iterativo que propone que el desarrollo de software está dividido en ciclos, cada ciclo construye un completo sistema.

Generalmente, cada ciclo es ejecutado como un proyecto separado cuya meta es agregarle una capacidad a un sistema existente (construido por el ciclo anterior).

Cada ciclo propiamente dicho se divide en 4 fases consecutivas:

- Fase de Incepción.
- Fase de Elaboración.
- Fase de Construcción.
- Fase de Transición.

Cada fase tiene un propósito distinguido, y cada fase se completa con outputs claramente definidos.

El propósito de la fase de incepción es establecer los objetivos y el alcance del proyecto y se completa con el lifecycle objective milestone. Este milestone debe especificar una visión general de la capacidad del sistema, qué beneficios se esperan agregar, algunos casos de uso para el sistema, los riesgos del proyecto, y un plan básico basado en el costo y la planificación. Basado en este output se toma una decisión de Go/No Go del proyecto, si el proyecto continúa es porque las partes están interesadas y de acuerdo en la visión del proyecto, el costo, los usos, ...

En la fase de elaboración, se diseña la arquitectura del sistema basada en los detalles del análisis de los requerimientos. Esta fase se completa con lifecycle arquitectute milestone. Al finalizar la fase de elaboración, se espera que la mayoría de los requerimientos sean identificados y especificados, con el fin de esta fase se tomarán medidas como la elección de la tecnología y arquitectura.

En la fase de construcción, se construye y testea el software de esta fase resulta el software a entregar incluyendo manuales de usuario, una vez terminada satisfactoriamente la fase resulta la initial operational capability.

La fase de transición se encarga de mover el software desde el ambiente de desarrollo al ambiente del cliente, fase que puede requerir un test adicional o la actualización de algún software.

Esta fase finaliza satisfactoriamente con the milestone product release.

Aunque las tareas se realizan en orden puede ser que una fase se realice iterativamente, para ir clarificando algún objetivo o requerimiento.

Los nombres de las fases de RUP son elegidos cuidadosamente ya que una fase no implica necesariamente la realización de una tarea, es decir permite gran flexibilidad en ese aspecto. El esfuerzo de cada fase depende de la elaboración de cada proyecto.

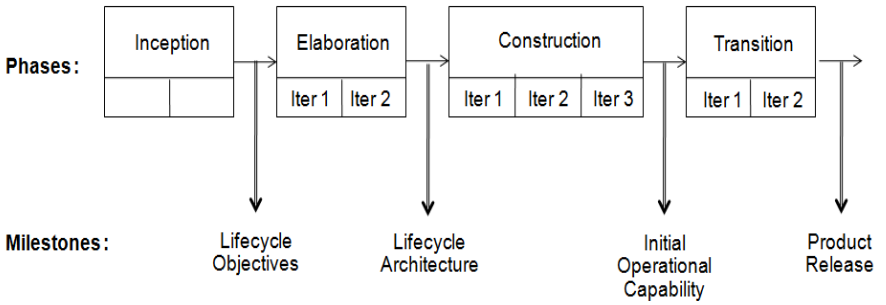

Figura 15: El Modelo RUP

PHASES	Inception	Elaboration	Construction	Transition
Requirements	High	High	Low	Nil
Anal. & Design	Low	High	Medium	Nil
Implementation	Nil	Low	High	Low
Test	Nil	Low	High	Medium
Deployment	Nil	Nil	Medium	High
Proj. Mgmt	Medium	Medium	Medium	Medium
Config. Mgmt	Low	Low	High	High

Time

Figura 16: Nivel de actividad de los subprocesos en diferentes fases de RUP

Conclusión

En general, RUP proporciona un modelo de proceso flexible que sigue un modelo iterativo no sólo a alto nivel sino en cada fase, y en cada fase permite realizar distintas tareas según las necesidades particulares del proyecto.

2.3.5. Timeboxing Model

Para aumentar la velocidad de desarrollo puede usarse el paralelismo entre diferentes iteraciones. Es decir, una nueva iteración comienza antes de que el sistema sea producido por la actual iteración. Porque se comienza la nueva antes que termine la anterior se puede reducir el tiempo de ciclo de trabajo. Sin embargo, para soportar una ejecución en paralelo, cada iteración tiene que estar estructurada adecuadamente y los equipos deben estar organizados **suitably** (adecuadamente).

La unidad básica del modelo es una **caja de tiempo**, que tiene una duración fija, esto hace que lo que pueda realizarse durante ese tiempo fijo esté acotado a un propósito determinado. Timeboxing cambia la perspectiva de desarrollo y hace a la planificación no negociable y una alta prioridad de commit.

Cada caja de tiempo se divide en una secuencia de etapas como en el modelo de cascada. Cada etapa realiza alguna clase de tarea por iteración y produce cla-

ramente un output. El modelo también requiere que la duración de cada etapa, sea aproximadamente el mismo. El modelo requiere gente especializada en cada etapa, hay una diferencia clara con el resto de los modelos ya que explícitamente hay equipos especializados en una etapa, en los otros implícitamente se supone que todos trabajan en la misma etapa.

Timeboxing es el mejor para agregar gente que no sabés donde sitiarlo para reducir tiempo. En otras palabras, el modelo provee el camino de menor entrega usando mano de obra (**manpower**) adicional.

Para ilustrar el uso de este modelo, considere un time box constituido por 3 etapas:

1. Especificación de los requerimientos.
2. Construcción.
3. Entrega.

La etapa de requerimientos es realizada por un equipo de analistas y termina con una lista de prioridades de requerimientos a construirse en esta iteración con un diseño de alto nivel.

El equipo a cargo de la etapa de construcción desarrolla el código para implementar los requerimientos, y realiza el testing.

El equipo de entrega realiza los testing de pre-entrega y luego instala el sistema para ser usado.

Estas 3 etapas pueden realizarse en aproximadamente el mismo tiempo de duración.

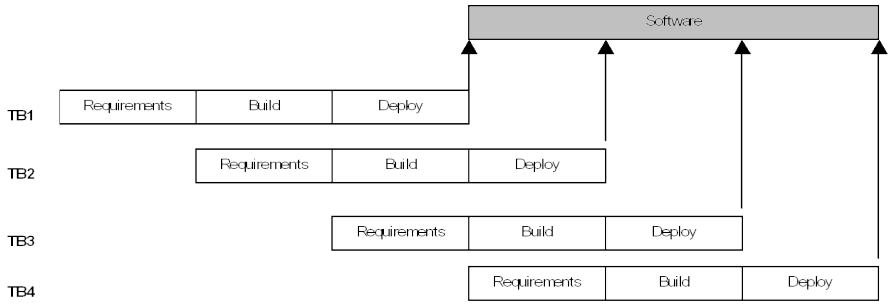

Figura 17: El modelo de TimeBox

A su vez cada equipo realiza distintas tareas:

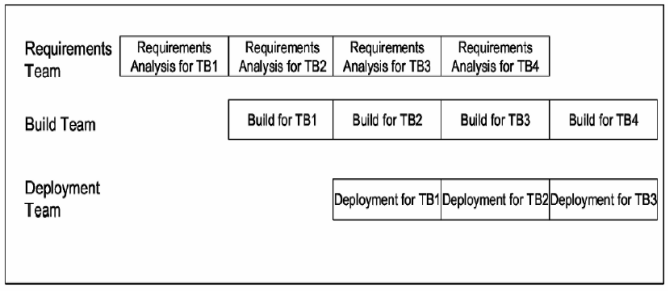

Figura 18: Ciclo Timeboxing

Conclusión

El modelo de timeboxing se acomoda a proyectos que requieren un gran número de características que deben ser desarrolladas en una arquitectura estable usando tecnologías estables.

Esas características deben ser significativas para el sistema y el usuario.

El principal costo del modelo es que incrementa la complejidad del proyecto de managment (y el manejo de entrega de productos) al haber múltiples desarrolladores trabajando activamente.

Asimismo el impacto de **una iteración inusal puede ser muy perjudicial**.

2.3.6. Programación Extrema y Procesos Ágiles

Creado para evitar los procesos basados en la burocracia fundamentalmente cascada:

- Trabajar el software es la clara medida del progreso en un proyecto.
- Para progresar en un proyecto, el software debe ser desarrollado y entregado rápidamente en pequeños incrementos, esto facilita acomodarse a cambios en los requerimientos.
- Face to Face está por encima de la documentación.
- Continuo Feedback para mediar la calidad del software.
- Un diseño simple rápido que se pueda mejorar con el tiempo en lugar de un gran diseño que contemple todos los posibles cambios.
- La entrega de datos se decide por los empleados según talentos individuales.

Un proyecto de programación extrema empieza con la historia del usuario que son pocas líneas de descripción acerca de que situaciones los usuarios desean que el sistema soporte. Luegos los empleados a cargo estiman cuánto tomará implementar la **historia del usuario**, esta estimación es aproximada generalmente situada en pocas semanas, luego de esto comienza a realizarse el plan que define lo que será construido de la historia.

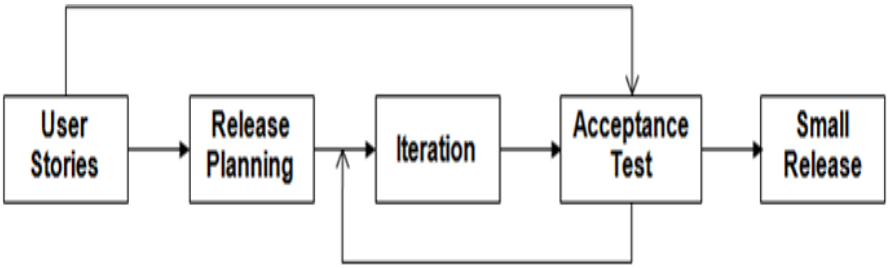

Figura 19: Estructura de un proceso con Programación Extrema

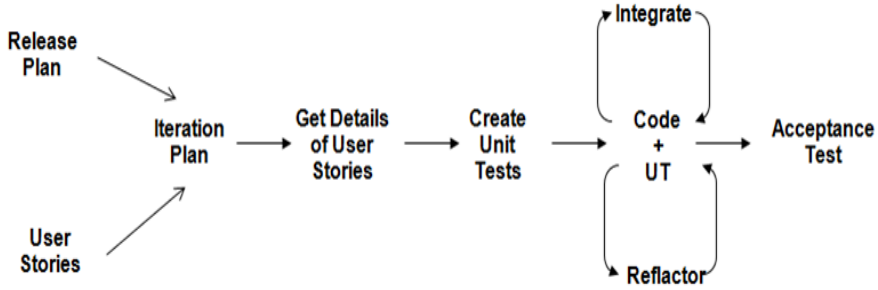

Figura 20: Una iteración en XP

Conclusión

XP, y otros métodos ágiles, son adecuados para situaciones en donde el volumen y ritmo de cambios es alto y donde los riesgos en los requerimientos son considerables.

Este modelo funciona fundamentalmente cuando el grupo de trabajadores está focalizado en el objeto y dispuesto al trabajo durante toda la fase de desarrollo, trabajando fuertemente en equipo.

2.3.7. Usando un modelo de proceso en un Proyecto

Según las restricciones del proyecto, vamos a optar por emplear algún modelo de proceso para maximizar las posibilidades de entregar el software y alcanzar la mayor calidad y productividad (Q&P).

Ejemplos

- Una universidad va a tardar en terminar de explicitar todos lo requerimientos y tenemos un pequeño grupo de desarrolladores, un sitio para una universisdad, 4 meses para el proyecto entonces debieramos usar RUP por la velocidad de entrega.
- Según la competencia se quiere mejorar una WebPage entonces para reducir costos contratan a otro equipo de otro país, timeboxing es el que mejor se acomoda.
- Universidad quiere registrar a los estudiantes en una base de datos, como los requerimientos son claros modelo cascada es el más adecuado.

2.4. Project Managment Process

Mientras la selección del proceso de desarrollo decide las fases y las tareas a realizarse, no especifica cuándo debiera terminar una fase, cuántos recursos deben asignarse por fase, o si una fase debiera ser monitoreada. Está claro que la calidad y productividad también dependen de esas decisiones críticas. Para alcanzar los objetivos del costo, calidad y la planificación los recursos deben ser asignados para cada actividad del proyecto, y el progreso de las diferentes actividades tiene que ser monitoreado y pueden ser necesarias acciones correctivas.

Todas estas actividades son parte del **project Managment Process**. Por lo tanto el proyecto de proceso de Managment es necesario para asegurar que la ingeniería del proceso termine encontrando los costos reales, planificables y la localidad.

El **proceso de managment de un proyecto** especifica todas las actividades necesarias para realizarse por el proyecto de Managment asegurar que los objetivos de costo y calidad sean alcanzados. Es una tarea básica que hace que una vez que se elija el proyecto de desarrollo, se implemente de manera óptima. Es decir, la tarea básica consiste en planear los detalles de la implementación del

proceso para el caso particular del proyecto y luego asegurar que el plan sea ejecutado apropiadamente. Para los grandes proyectos, un apropiado Managment del proceso es clave para el éxito.

Las actividades de un proceso de Managment para un proyecto pueden ser agrupadas generalmente en 3 fases:

1. Planeamiento.
2. Monitoreo y control.
3. Análisis de terminación.

El proyecto de Managment comienza con el planeamiento que es la actividad más crítica del Managment.

El objetivo de esta fase es desarrollar un plan que el desarrollo del software debe seguir y qué objetivos debe encontrarse el proyecto en cuanto a la calidad y productividad.

El plan de software se produce generalmente antes del comienzo de la actividad de desarrollo y es actualizado según el development process vaya progresando.

En el planeamiento, la mayor tarea es dar una estimación del costo, planificación y **milestone determination**, el staff, planes de control de calidad, y el control y monitoreo de los plazos.

El proyecto de plan es indudablemente la actividad más importante del Managment y forma las bases para el monitoreo y control. Luego vamos a gastar un capítulo entero en desarrollar extensamente: Planificación del Proyecto.

El proyecto de monitoreo y la fase de control del proceso de Managment, es la más larga en cuanto a la duración, acompañando la mayoría del proceso de desarrollo. Esta fase incluye todas las actividades que el proyecto de Managment tiene que realizar mientras se está realizando la fase de desarrollo para asegurar que se logren los objetivos del proyecto y que el desarrollo marche de acuerdo al plan de desarrollo (incluyendo la actualización del plan si es necesario). Como el costo, la planificación y la calidad son los principales agentes del proceso, la mayoría de las actividades de esta fase se realizan para monitorear factores que afecten a estos agentes. Monitorear los posibles riesgos del desarrollo del proceso es una actividad fundamental para prevenir que se alcancen los objetivos, y si con la información obtenida del monitoreo sugiere que posiblemente los objetivos no sean encontrados, necesariamente deben tomarse acciones para exhortar un adecuado control en las fases de desarrollo.

El monitoreo del desarrollo del proceso requiere una apropiada información acerca del proyecto, esta información es típicamente obtenida por el proceso de Managment del proceso de desarrollo. En consecuencia, la implementación del desarrollo del modelo del proceso debe asegurar que el proceso de Managment necesita para ese paso. Es decir, el proceso de desarrollo provee la información que el poceso de Managment necesita. Sin embargo la interpretación de esa información forma parte del monitoreo y control. Mientras que el monitoreo y control se debe realizar enteramente durante el proyecto, la última fase del

Managment Process - **Termination Analysis** se realiza cuando el desarrollo a concluído.

La idea básica para realizar un análisis de terminación es proveer información acerca del desarrollo del proceso y **aprender** sobre el proyecto realizado para mejorar el proceso. Esta fase también es llamada Post morten analysis (Vulgarmente: enterrar el muerto). En el modelo iterativo esta fase puede realizarse luego de cada iteración para mejorar sobre la marcha.

La siguiente figura muestra una idealización del proceso:

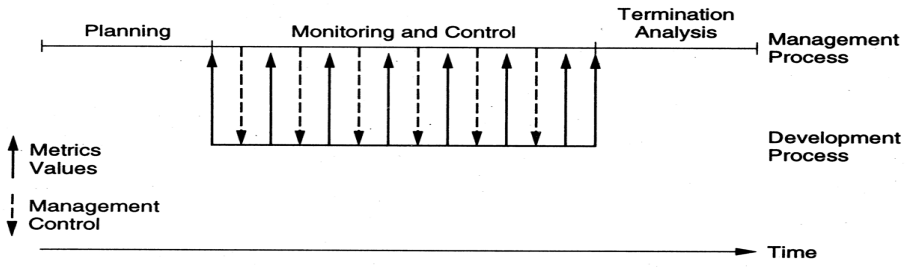

Figura 21: Relación temporal entre el development y Managment Process

Como la figura lo muestra fluye mucha información entre el monitoreo y control y el desarrollo para ir mejorando la ejecución del plan del proceso.

2.4.1. El proceso de inspección

El objetivo principal es detectar los defectos en los productos de trabajo. Inicialmente utilizada para el código, actualmente usada en todos los tipos de productos de trabajo. Está reconocido como una de las mejores prácticas de la industria. Mejora tanto la calidad como la productividad.

Los defectos pueden introducirse en el software en cualquier etapa entonces deben eliminarse en cada etapa.

Las inspecciones pueden realizarse sobre cualquier documento, incluidos requerimientos, diseños y planificaciones. Este proceso se enfoca en **encontrar problemas, no solucionarlos**.

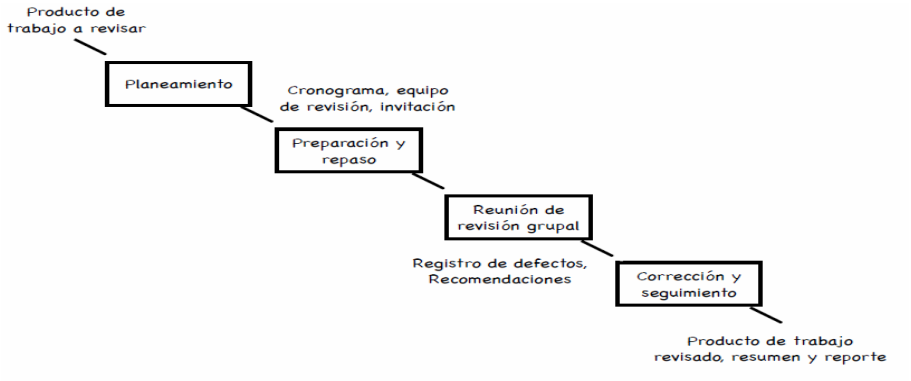

Figura 22: El proceso de inspección

Project name and code :
Work product name and ID:
Reviewer name:
Effort spent for preparation (hrs):
Defect List:

Sl	Location	Description	Criticality / Seriousness

Figura 23: Planilla de la revisión individual

2.4.2. Proceso de Adminitración de Procesos

Un proceso no es una entidad estática: éste debe cambiar para mejorar la calidad y productividad. La administración del proceso se enfoca en la evaluación y mejora del proceso.

Importante: es diferente a la administración del proyecto, en el cual se administra el **proyecto**. La administración de proceso es un tópico avanzado.

Para mejorar el proceso, una organización debe comprender el proceso actual: requiere que el proceso esté bien documentado, que sea apropiadamente ejecutado en los proyectos, recolectar datos de los proyectos para comprender el desempeño del proceso en los proyectos.

Es mejor que los cambios realizados al proceso se hagan incrementalmente, de a pasos pequeños. Estos deben ser cuidadosamente seleccionados: decidir qué cambios hacer y cuándo.

Existen marcos que sugieren formas de proceder en la mejora del proceso. Ejemplo: CMM Capability Maturity Model. Tiene 5 niveles para el proceso del software (el primero y más bajo es ad-hoc).

En cada nivel el proceso tiene ciertas capacidades y establece las bases para

Cuadro 4: Pautas para la revisión de los productos de trabajo

Producto de Trabajo	Enfoque de la Inspección	Participantes
Especificación de los Requisitos	-Cumple con las necesidades del cliente -Es implementable -Omisiones, inconsistenias , ambigüedades	Cliente Analista, Diseñador Desarrollador
Diseño de alto nivel	- El diseño implementa los Requisitos -El diseño es implementable -Omisiones, calidad del diseño	Autor de la SRS, Diseñador de alto nivel Desarrollador
Código	- El código implementa el diseño. - El código es completo y correcto. - Defectos en el código. - Otras características de calidad	Desarollador Diseñador Tester
Casos de Tests	- Conjunto de casos de Tests verifica los requerimientos en la SRS. - Los casos de tests son ejecutables	Autor de la SRS Tester Líder del proyecto
Plan de administración del proyecto	- El plan completo y especifica todos los componentes del plan. - Es implementable - Omisiones y ambigüedades	Líder del proyecto, Otro administrador del proyecto

pasar al siguiente nivel.

Para moverse de un nivel a otro, CMM especifica áreas en las cuales enfocarse. Se utiliza ampliamente en la industria del software.

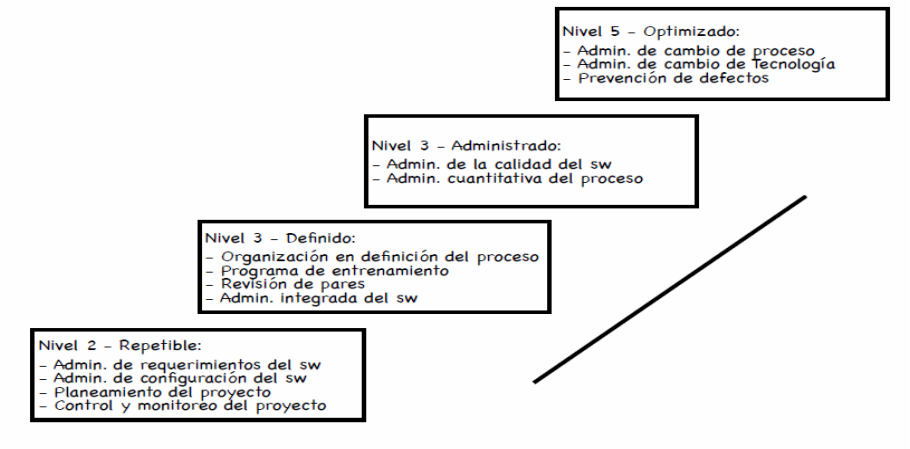

Figura 24: Niveles de CMM

2.5. Resumen

- La productividad y calidad alcanzada en un proyecto de software depende del proceso usado para ejecutar el proyecto, debido a esto, los procesos forman el corazón de la ingeniería del software.
- Un proceso es un conjunto de actividades realizadas en algún orden para alcanzar los objetivos deseados. Un modelo de proceso es una especificación general de un proceso que se sitúa en las mejores condiciones para algunas situaciones.
- El proceso del software consiste en muchos y diferentes componentes de procesos. Los más importantes son el **development process** y el **Managment process**.
- Development Process: enfatiza en cómo el software puede ser sistematizado. Hay muchos y diferentes modelos de procesos, cada cual se acomoda mejor a un tipo de problema:
  - Cascada es conceptualmente el más simple, donde los requerimientos, diseño, codificación y testing son fases realizadas en una progresión lineal. Es uno de los más usados y es el mejor cuando entendemos el problema.
  - Modelo prototipado: se construye un prototipo antes del sistema final, el prototipo es desarrollado enfocado en los requerimientos que no son totalmente comprendidos para llegar a un acuerdo. Generalmente es usado en los proyectos donde los requerimientos no son claros.
  - En el modelo iterativo: el software se desarrolla en iteraciones. cada iteración realiza un trabajo de sistema de software, el modelo no requiere que todos los requerimientos sean conocidos al comienzo, permite una respuesta temprana para las proximos iteraciones y reduce los riesgo **ya que aporta valor a medida que avanza el proyecto**.
  - En RUP, el proyecto es ejecutado en una secuencia de 4 fases:
    1. Incepción.
    2. Elaboración.
    3. Construcción.
    4. Transición.

    Cada una termina con una definida **milestone**.

    Una fase puede repetirse, es un modelo flexible que permite ir desde cascada a prototipo si se los desea.
  - Timeboxing: las diferentes iteraciones tienen la misma duración, y se divide en igual cantidad de etapas. Cada iteración de una etapa realiza un commit. Este modelo reduce en promedio el cierre de cada iteración, es muy usado en situaciones donde hay un corto ciclo de trabajo.

- Enfoques ágiles se basan fundamentalmente en desarrollar software en pequeñas iteraciones, el trabajo sobre el código es la única medida de progreso. En Programación extrema (XP) se realizan simples diseños para algún requerimiento inicial y se va mejorando según los nuevos requerimientos del cliente.

- El proyecto de Managment se enfoca en planear y controlar el proceso de desarrollo y consiste en 3 fases: planeamiento, monitoreo y control y un análisis de terminación. Muchos de los proyectos de Managment trabajan alrededor de un plan de proyecto que es producido en la planificación.

Ejercicios

1. ¿Cuál es la relación entre un modelo de proceso, la especificación de un proceso y un proceso de un proyecto?

2. ¿Cuáles son las salidas claves de una iteración en un proyecto que sigue el modelo iterativo de desarrollo?

3. ¿Qué modelo emplearía para los siguientes proyectos?

   a) Un proceso simple de procesamiento de datos.

   b) Un sistema para personas que nunca usaron una computadora antes, donde la interfaz y ser user-friendly es extremadamente importante.

   c) Un sistema de hojas de cálculos con requisitos básicos y otros deseables basados en ellos.

   d) Una web donde cambian rápido los requerimientos y el desarrollo del hogar está permitido en aspectos del proyecto.

   e) Una Página web con una larga lista de tareas para hacer y frecuentemente se irán agregando más tareas.

4. Un proyecto usa el modelo Timeboxing con tres etapas de timebox, pero de distinto tamaño. Suponga que:

   - La fase de requerimientos y especificación toma 2 semanas y esté a cargo de 2 personas.
   - La fase de construcción tome 3 semanas y esté a cargo de 4 personas.
   - La fase de entrega tome 1 semana y esté a cargo de 2 personas.

   Rediseñar el sistema para explotar al máximo la utilización de los recursos, suponiendo que todos son especialiastas en cualquier tarea:

   Hint: explote que la suma de la duración del primer tiempo y el tercero es igual a la del segundo

5. ¿Qué produce el monitoreo de las actividades en el proceso de desarrollo?

3. Análisis y especificación de los requisitos del Software

La IEEE, define requerimientos como:

1. Una condición o capacidad necesaria por un usuario para solucionar un problema o alcanzar un objetivo.

2. Una condición o capacidad necesaria que debe poseer o cumplir un sistema ... para satisfacer un contrato, un estándar u otro documento formal.

Note que en los requerimientos del software estamos tratando con los requerimientos del sistema propuesto, es decir, las capacidades que el sistema, si estuviera hecho debería tener.

Como hemos visto, todos los modelos de desarrollo requieren los requerimientos para ser especificados. Enfoques como XP requieren sólo un alto nivel de requerimientos para ser especificados en forma escrita, detallados requerimientos se sustituyen con la interacción con el cliente en la iteración de los requerimientos y se reflejan directamente en el software. Otros enfoques prefieren que los requerimientos sean especificados precisamente.

En tales situaciones, el objetivo de la actividad de requerimientos es producir la **espeficicación de los requisitos del software** que describen qué debe hacer el software sin decir cómo lo harán.

En este capítulo hablaremos acerca de:

- El rol de la SRS y los beneficios que una SRS brinda.
- Las diferentes actividades en el proceso para producir la deseada SRS.
- Las características deseadas de una SRS, la estructura de un documento SRS y sus componentes claves.
- Los enfoques para el análisis y especificación de los requisitos funcionales y cómo se pueden desarrollar los casos de uso.
- Otros enfoques para analizar los requerimientos como los diagramas de flujo.
- Cómo se validan los requerimientos.

3.1. El valor de una buena SRS

La mayoría de los software se originan en la necesidad de algún cliente.

- Entrada: las necesidades se encuentran en la cabeza de alguien (ideas abstractas).
- Salida: un detalle preciso de lo que será el sistema futuro.

El sistema de software propiamente dicho es creado por algunos desarrolladores. Finalmente, el sistema completo será usado por los usuarios. Entonces, hay 3 partes mayormente interesadas en el nuevo sistema: el cliente, el desarrollador y el usuario. De algún modo los requerimientos para el sistema que van a satisfacer las necesidades del cliente y las preocupaciones de los usuarios tienen que comunicarse al desarrollador. El problema es que ese cliente generalmente entienda el software o el desarrollo del proceso del software, y que el desarrollador a veces no entiende el problema del cliente y el área de aplicación. Esto provoca una brecha en la comunicación entre las partes involucradas y el desarrollo del proyecto. El propósito básico de la SRS es achicar esta brecha para tener una visión compartida de lo que va a realizarse. Por lo tanto una de las principales ventajas de una buena SRS es:

- Una SRS establece las bases del acuerdo entre el cliente y el proveedor de lo que el software hará.

Las bases del acuerdo frecuentemente se formalizan en un contrato legal entre el cliente y el desarrollador. Entonces, a través de la SRS, el cliente describe que es lo que espera del desarrollo, y el desarrollador comprende las capacidades que tiene que construir el software. Una ventaja ya mencionada pero importante es:

- Una SRS provee una referencia para la validación del producto final.

Es decir, la SRS ayuda al cliente a determinar si el software cumplió los requerimientos. Sin una apropiada SRS no hay manera de que el cliente pueda determinar si el software que se le entregó fue el que había ordenado, y no hay manera de que el desarrollador pueda convencer al cliente de que todos los requerimientos han sido cumplidos.

Proveer las bases del acuerdo y validación debería ser la razón fuerte para ambos, el cliente y el desarrollador, para hacer un minucioso y riguroso trabajo de entendimiento de los requerimientos y especificación, pero hay otras razones prácticas e importantes para tener una buena SRS.

Estudios muestran que muchos errores se realizan durante la fase de requerimientos y un error en la SRS se manifestará como un error en el sistema que implemente la SRS. Claramente si queremos un producto final de alta calidad que tenga pocos errores, debemos comenzar con una SRS de alta calidad. En otras palabras podemos decir que:

- Una SRS de alta calidad es un prerequisito para un software de alta calidad.

Finalmente la calidad de la SRS tiene un impacto en el costo y planificación del proyecto. Sabemos que pueden existir errores en la SRS, pero también sabemos que el trabajo para solucionar un error mientras avanza el proceso de desarrollo crece casi exponencialmente. Por lo tanto, mejorando la calidad de los requerimientos, podemos tener un gran ahorro en el futuro por tener que realizar menor cantidad de extracciones de errores costosos.

En otras palabras:

- Una SRS de alta calidad reduce el costo de desarrollo.

Ahorro sustancial: esfuerzo extra invertido en requisitos produce ahorros varias veces mayores a tal esfuerzo.

Fase	Costo (Persona-Horas)
Requisitos	2
Diseño	5
Codificación	15
Test	50
Operación y mantenimiento	150

3.2. El proceso de los requerimientos

El proceso de los requerimientos es la secuencia de actividades que necesitamos para realizar en la fase de requerimientos y que culmina con la producción de una documentación de alta calidad que contiene la SRS. El proceso de requerimientos típicamente consiste de tres tareas básicas: el problema o análisis de los requerimientos, la especificación de los requerimientos y la validación de los requerimientos.

El **análisis del problema** a menudo empieza con un alto nivel del "estado del problema". Durante el análisis el dominio del problema y el ambiente son modelados en un esfuerzo para entender el comportamiento del sistema, las restricciones del sistema, sus entradas y salidas, ...

El propósito básico de esta actividad es obtener un completo entendimiento de lo que necesita el software. Frecuentemente, durante el análisis, el analista tendrá una serie de reuniones con el cliente y usuarios finales. En las primeras reuniones, los usuarios y el cliente le explicarán al analista su trabajo y sus necesidades y percepciones, el analista fundamentalmente debe absorver los pedidos, una vez que el analista comprende el aspecto general del pedido focaliza en partes que no ha comprendido totalmente, puede ir administrando modelos o puede ir haciendo una tormenta de ideas o pensamientos acerca de lo que el sistema debe hacer. En los últimos encuentros el analista explica o expresa al cliente lo que entendió que el sistema debe hacer y se encarga de comprobar si en realidad es consistente con los objetivos del cliente.

El entendimiento obtenido en el análisis del problema forma las bases de la **especificación de los requerimientos**, en la cual se focaliza en clarificar las especificaciones de los requerimientos en un documento. Cuestiones como la representación, la especificación de los lenguajes, y las herramientas son guiadas durante esta actividad. Como el análisis produce una gran cantidad de información y conocimientos con posibles redundancias, organizarlos correctamente y describir los requerimientos es un objetivo importante de esta actividad.

La **validación de los requerimientos** se concentra en asegurar que lo especificado en la SRS son de hecho todos los requerimientos del software y asegurar que la SRS sea de alta calidad. El proceso de los requerimientos termina con la producción de una validada SRS. Hablaremos un poco más de esto en el capítulo.

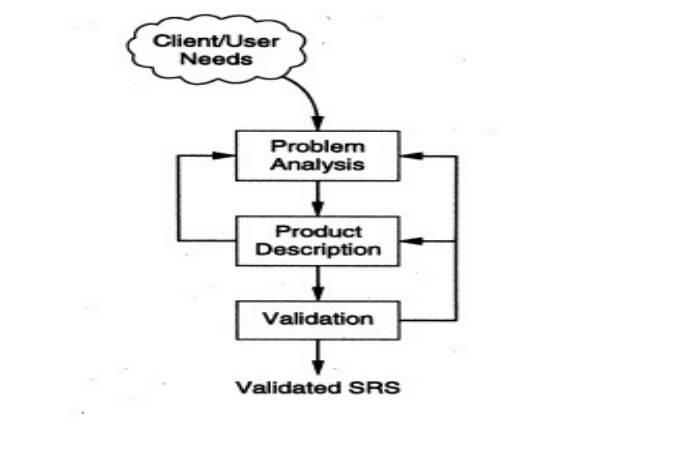

Figura 25: El proceso de los requerimientos no es necesariamente lineal

Cabe destacar que el proceso de los requerimientos no es una secuencia lineal de esas 3 actividades y hay una considerable superposición y respuesta entre estas actividades. La estructura general de los requerimientos está mostrada en la figura. Como se ve, desde la tarea de especificaciones es posible que volvamos a la tarea de análisis. Esto ocurre frecuentemente porque algunas partes del problema son analizadas y especificadas antes que otras partes son analizadas y especificadas. Además, el proceso de especificación frecuentemente muestra deficiencias en el entendimiento del problema, de este modo es necesario un análisis adicional. Esta actividad puede revelar problemas propios de la especificación, que requieran volver a la etapa de especificación, o puede mostrar defectos en el entendimiento del problema que requieran volver a la tarea de análisis.

3.3. Especificación de los requerimientos

El output final es el documento SRS. como el análisis precede a la especificación, la primer cuestión que surge es si se realiza un modelado formal durante el análisis, ¿Por qué los outputs del modelado no son tratados como la SRS? La principal razón es que el modelado generalmente se enfoca en la estructura del problema, no su comportamiento externo. En consecuencia, elementos como interfaces de usuario son modeladas con poca frecuencia mientras estos frecuentemente forman el mayor componente en una SRS. Similiarmente, para facilitar el modelado frecuentemente "temas erróneos" como situaciones de errorres son con poca frecuencia modelados correctamente, mientras que en una SRS, el comportamiento bajo esas situaciones también debe estar especificado. Similarmente, restricciones de performance, de diseño, cumplimiento de estándares, recuperación, ..., etc, no están incluidas en el modelo pero deben estar claramente especificadas en la SRS porque el diseñador debe diseñar adecuadamente el sistema para dichas situaciones. Por lo tanto queda claro que los outputs de

un modelo no pueden formar parte de una deseable SRS.

La transición del análisis a la especificación tampoco debe esperarse que sea sencilla, incluso si algún modelado formal es usado durante el análisis. Una buena SRS necesita especificar muchas cosas, algunas de las cuales no fueron satisfechas durante el análisis. Esencialmente, lo que pasa desde las actividades de requerimientos a la especificación es el conocimiento adquirido acerca del sistema. El modelado es esencialmente una herramienta para ayudar a obtener un minucioso y completo conocimiento acerca de los propósitos del sistema. La SRS se escribe basada en el conocimiento adquirido durante el análisis. Cómo convertir el conocimiento en un documento estructurado no es sencillo, la especificación propiamente dicha es la mayor tarea, que es relativamente independiente.

3.3.1. Características deseables de una SRS

Para satisfacer apropiadamente los objetivos básicos, una SRS debe tener ciertas propiedades y debe contener distintos tipos de requerimientos. Algunas de las características deseables de una SRS son:

1. Correcta.
2. Completa.
3. Inequívoca (sin ambigüedades).
4. Verificable.
5. Consistente.
6. Rankeada por importancia y/o estabilidad.
7. Rastreable.
8. Modificable.

Una SRS es **correcta** si cada requerimiento incluido en la SRS representa algo requerido en el sistema final. Es **completa** si todo lo que se supone que debe hacer el software y las respuestas del software a todas las clases de input de datos están en la SRS. Es **inequívoca** si todos y cada uno de los requerimientos tiene una única interpretación. Los requerimientos a menudo son escritos en lenguaje natural que es inherentemente ambiguo. Si los requerimientos están especificados en lenguaje natural el redactor de la SRS tiene que tener un especial cuidado para asegurar que no haya ambigüedades.

Una SRS es **verificable** si todos y cada uno de los requerimientos escritos son verificables. Un requisito es verificable si existe algún costo/efectivo proceso que pueda chequear si el software final cumple con los requerimientos. Es **consistente** si no hay requerimientos que entren en conflicto con otros.

Generalmente, todos los requerimientos del software no tienen igual importancia, algunos son críticos, otros son importantes pero no críticos y otros que

son deseables pero no importantes. Similarmente, algunos requerimientos forman un núcleo que no cambian su importancia con el tiempo, mientras que otros dependen del tiempo. Algunos ofrecen mayor valor a usuarios que otros. Una SRS está **rankeada** por importancia y/o estabilidad si se establece la importancia y la estabilidad para cada requerimiento. La estabilidad de los requerimientos reflejan las posibilidades de ser cambiadas en el futuro. Este entendimiento del valor de cada requerimiento es clave para los modelos de desarrollo iterativo. Los requerimientos pueden ser: críticos, importantes pero no críticos, deseables pero no importantes. Algunos requerimientos son esenciales y difícilmente cambien con el tiempo. Otros son propensos a cambios entonces se necesita definir un orden de prioridades en la construcción para reducir riesgos debido a cambios de requerimientos.

Rastreable: se debe poder determinar el origen de cada requerimiento y cómo éste se relaciona a los elementos del software.

- Hacia adelante, dado un requerimiento debe poder detectar en qué elementos de diseño o código tiene impacto.
- Hacia atrás: dado un elemento de diseño o código se debe poder rastrear qué requerimientos está atendiendo.

Modificable:

- Si la estructura y estilo de la SRS es tal que permite incorporar cambios fácilmente preservando completitud y consistencia.
- La redundancia es un gran estorbo para modificabilidad (puede resultar en inconsistencia).

De todas estas características, ser completo es quizás la más importante y la más dificil de establecer. Uno de los errores más comunes en la fase de requerimientos es que la lista de requisitos esté incompleta. Cuando olvidamos algún requerimiento necesitamos agregarlo luego durante el ciclo de desarrollo, donde en general es más caro incorporarlo. Lo completo también es la mayor fuente de desacuerdos entre el cliente y el desarrollador.

Sin embargo algunos piensan que ser completo no debe ser extensamente en todos los detalles, ya que sino esto se hace una larga tarea de requerimientos que son difíciles de verificar, lo que hace que los desarrolladores tengan distintos puntos de vista sobre los defectos en el software.

Para estar completo es necesario que esté "suficientemente detallado" y esto varía según el modelo, en un iterativo no se exigirá tanto como en el modelo cascada. Juntos la Performance y los requerimientos de Interfaz y restricciones de diseño pueden ser llamados "**requisitos no funcionales**".

3.3.2. Componentes de una SRS

La integridad o completitud de las especificaciones es difícil de alcanzar e incluso más difícil de verificar. Teniendo una guía acerca de los diferentes asuntos

que una SRS debe especificar facilitaremos llegar a una especificación completa de los requerimientos.

Los lineamientos básicos que una SRS debe seguir son:

- Funcionalidad.
- Performance.
- Restricciones de Diseño impuestas en la implementación.
- Interfaces Externas.

Los **requisitos funcionales** especifican que se espera del comportamiento del sistema, que outputs se producen dados ciertos inputs. Describen la relación entre las entradas y salidas del sistema. Para cada requisito funcional, una detallada descripción de todos los inputs y su fuente, las unidades de medida, y el rango de inputs válidos deben ser especificados. Todas las operaciones a realizar frente al input para obtener el output debe ser especificado. Esto incluye la especificación de la validación de chequeo entre lo que se ingresa y sale del sistema. Parámetros afectados por la operación y ecuaciones y otras operaciones deben ser usadas para transformar la entrada en su correspondiente output. Por ejemplo si hay alguna fórmula que deba realizar un output debe ser especificada.

Una importante parte de la especificación del comportamiento del sistema son las situaciones anormales, como un input inválido o un error durante la computación. Un requisito funcional debe establecer claramente la relación entre inputs y outputs.

Los **requisitos de performance** son partes de la SRS que especifican las restricciones de performance en el software. Hay dos tipos de requisitos de performance: estáticos y dinámicos.

- Requisitos Estáticos son aquellos que no imponen restricciones en las características de la ejecución del sistema. Estos incluyen requerimientos como el número de terminales a ser soportadas, el número de usuarios que puede soportar en simultaneo, y el número de archivos que el sistema tiene que procesar y sus tamaños. Estos también son llamados requisitos de **capacidad** del sistema.
- Requisitos Dinámicos especifican restricciones en la ejecución del comportamiento del sistema, estos típicamente incluyen el tiempo de respuesta y restricciones de rendimiento en el sistema. El tiempo de respuesta es el tiempo esperado para terminar una operación en circunstancias específicas. El rendimiento es el número esperado de operaciones que se pueden realizar por unidad de tiempo. Por ejemplo en la SRS se puede especificar el número de transacciones que se pueden procesar por unidad de tiempo, o el tiempo al que debiera responder a un comando. Rangos aceptables de diferentes acciones de parámetros deben ser especificados, así como un aceptable realización para ambos normal en condiciones de trabajo pico.

  Todos estos requisitos deben situarse en términos medibles. Requisitos como “el tiempo de respuesta debe ser bueno” o “el proceso de todas las

transacciones es rápido" no son deseables ya que son imprecisas y no se pueden verificar. En lugar, deben ser como "la respuesta en tiempo del comando x debe ser menor a un segundo el 90 % de las veces".

Hay un número de factores en el ambiente del cliente que pueden restringir las chances de buscar lo mejor en cuanto al diseño. Estos factores incluyen estándares que deben ser seguidos, límites de recursos, ambiente operativo, requisitos de confiabilidad y seguridad, y políticas que pueden tener un gran impacto en el diseño del sistema. Una SRS debe identificar y especificar todas estas restricciones, algunos ejemplos de estos son:

- Cumplir estándares.
- Limitaciones de Hardware.
- Confiabilidad y falta de tolerancia.
- Seguridad.

En la parte de requisitos de **interface externa**, todas las interacciones del software con la gente, hardware y otros software debe ser claramente especificado. Para la interfaz del usuario, las características de cada interfaz de usuario del producto del software deben ser especificados. La interfaz con el usuario está cobrando gran importancia y debe prestársele atención. Un preliminar manual de usuario debe ser creado con todos los comandos de usuario, formatos de pantalla, una explicación de cómo el sistema se va a mostrar al usuario, y la respuesta y mensajes de errrores. Como toda especificación, estos requisitos deben ser precisos y verificables. Entonces, frases como "el sistema debe ser user friendly" se deben evitar y deben ser usadas frases como "los comandos no deben ser mayores a 6 caracteres" o los "nombres de los comandos deben reflejar la función que realizan".

Para los requisitos de interfaz del hardware, la SRS debe especificar las características lógicas de cada interfaz entre el producto y las componentes de hardware. Si el software está hecho para ser ejecutado en un hardware, todas las características del hardware, incluyendo restricciones de memoria deben ser especificadas.

Los requisitos de interfaz deben especificar la interfaz con otro software del sistema que es usado o que usará. Esto incluye la interfaz con el sistema operativo.

3.3.3. Estructura de la documentación de los requerimientos

Los requisitos tienen que estar especificados en algún lenguaje de especificación. Aunque existen notaciones formales para especificar ciertas propiedades del sistema, los lenguajes naturales en la actualidad son en general los más usados para especificar los requisitos. Cuando se usan lenguajes formales, son usados en general para especificar propiedades o para especificar partes del sistema, como parte del conjunto de la SRS.

Todos los requisitos para el sistema, declaraciones usando una notación formal o lenguaje natural tienen que ser incluidos en un documento que sea claro y conciso. Por esto, es necesario organizar apropiadamente el documento de los requisitos. Aquí daremos la organización basada en la guía de la IEEE para los requisitos de las especificaciones del software.

Los estándares de la IEEE reconocen el hecho de que diferentes proyectos pueden requerir que los requerimientos sean organizados de otra manera, es decir, no hay un único método que sea adecuado para todos los proyectos. Provee diferentes maneras de estructurar la SRS. Las primeras dos secciones de la SRS son comunes a todas. La estructura general se da en la siguiente figura.

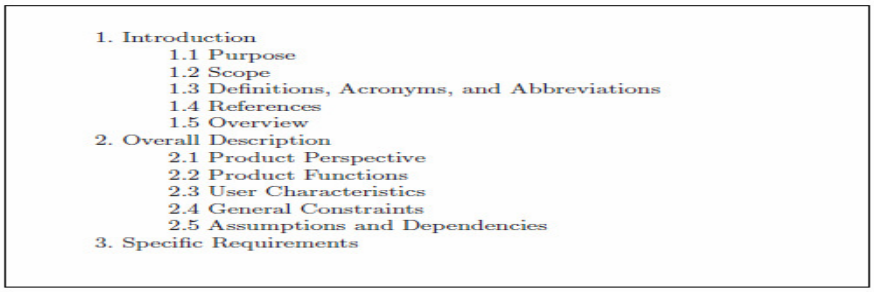

Figura 26: Estructura General de una SRS

La sección de introducción contiene el propósito, alcance, una visión general, ... etc, del documento de requisitos. El aspecto clave es clarificar la motivación y los objetivos de negociones que conducen el proyecto, y el alcance del proyecto. Los siguientes otorgan una perspectiva general del sistema como se comporta dentro de un gran sistema y una visión de todos los requisitos del sistema. No se mencionan requisitos detallados. La **perspectiva del producto** es esencialmente la relación de este producto con otros, definiendo si es un producto independiente o si forma parte de uno más grande, y cuál es la principal área del producto. Una descripción general y abstracta de las funciones a realizarse cuando se termine el producto. Diagramas esquemáticos mostrando una visión general de diferentes funciones y su relación con otras pueden ser usados. Similarmente, características generales de un eventual usuario final y restricciones generales también se especifican.

Si son usados modelos ágiles, esto es suficiente para una fase de requerimientos iniciales, estos enfoques prefieren detallar todos los requerimientos cuando lo requerido es implementado.

La sección de descripción detallada de requisitos describe los detalles de los requerimientos que el desarrollador necesita saber para diseñar y desarrollar el sistema. Esto es en general la parte más grande e importante del documento. Para esta sección diferentes organizaciones son sugeridas por el estándar. Estos requisitos pueden ser organizados por los modos de operación, clases de usuarios, objetos, características, estímulos o una jerarquía funcional.

Un método para oganizar la especificación de los requisitos es primero espe-

cificar las interfaces externas, seguido de los requisitos funcionales, y atributos del sistema, esta estructura se muestra en la siguiente figura.

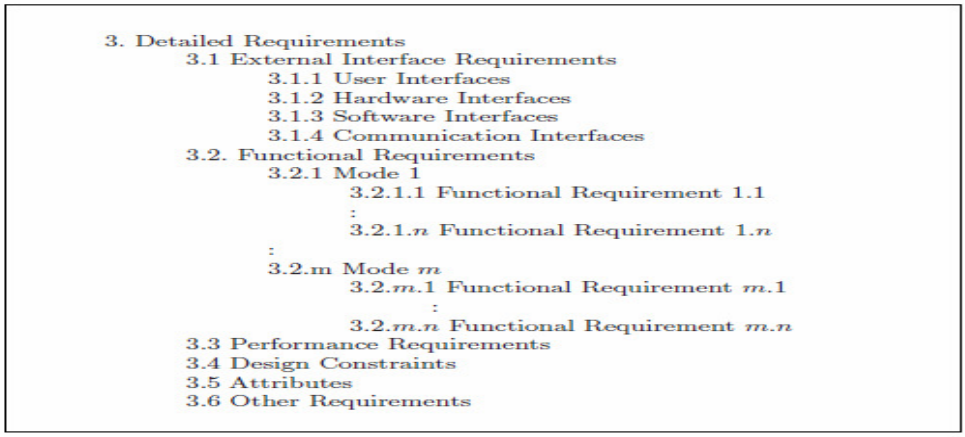

Figura 27: Una organización para la especificación de los requisitos

La sección de la interfaz externa de los requisitos especifica todas las interfaces del software: la gente, otros software, hardware y otros sistemas. La **interfaz de usuario** es claramente una componente muy importante, ella especifica cada interfaz humana que el sistema planea que tenga, incluyendo formatos de pantalla, tabla de menú, y una estructura de comandos. En las **interfaces de Hardware**, las características lógicas de cada interfaz entre el software y el hardware en las que el software puede correr se especifican. Esencialmente, cualquier acción del hardware se lista aquí. En la sección de requisitos funcionales, se describen las capacidades funcionales del sistema. En esta organización, las capacidades de todos los modos de operación del software se describen. Para cada requisito funcional, deben aclararse los inputs esperados outputs.

La sección de performance debe especificar tanto los requisitos estáticos como dinámicos. Todos los factores que restringen el diseño del sistema se describen en la sección de restricciones de performance.

La sección de atributos especifica alguno de los atributos generales que el sistema debería tener. Cualquier requisito no cubierto aquí es listado en la sección de otros requisitos.

La sección de restricciones de diseño especifica todos los requisitos impuestos en el diseño (seguridad, falta de tolerancia y cumplimiento de estándares).

Cuando se usan casos, los requisitos funcionales de la SRS se reemplaza por descripción de uso de casos. Y la parte de perspectiva del producto en la SRS debe proveer un resumen de los casos de uso.

3.4. Especificación funcional con casos de uso

Los requisitos funcionales a veces forman el núcleo del documento de requisitos (SRS). El enfoque tradicional para especificar las funcionalidades es especificar cada función que el sistema debe proveer. Los casos de uso especifican la funcionalidad del sistema especificando el comportamiento del sistema, capturado como las interacciones de los usuarios con el sistema. Los casos de uso pueden ser usados para describir el proceso de negocios de un proceso de negocios más grande o una organización que entregue el software, o puede sólo describir el comportamiento de un sistema de software. Nos enfocaremos en describir el comportamiento del software que vamos a construir.

Aunque los casos de uso son principalmente para especificar el comportamiento, ellos pueden también ser usados efectivamente para el análisis. Luego cuando hablemos de cómo desarrollar casos de uso discutiremos como ellos pueden ayudar también en la obtención de recursos.

Los casos de uso llamaron la atención luego de que fueron usados como parte del enfoque orientado a objetos propuesto por Jacobson. Debido a esta conexión con el enfoque orientado a objetos, los casos de uso son a veces vistos como parte del enfoque de orientación a objetos para desarrollar software. Sin embargo, hay un método general para describir la interacción del sistema. La discusión de los casos de uso aquí están basados en los conceptos y la discusión del proceso.

3.4.1. Fundamentos o conceptos Básicos

Un sistema de software (en nuestro caso cuyos requisitos están empezando a ser descubiertos) puede ser usado por muchos usuarios, o por otros sistemas. En un caso de uso, un **actor** es una persona o un sistema que usa el sistema para alcanzar algún objetivo. Note que un actor representa un grupo de usuarios (gente o sistema) que se comportan de manera similar. Diferentes actores representan grupos con diferentes objetivos. Entonces, es mejor tener un actor "receptor" y un actor "emisor" que tiene un genérico actor "usuario" para un sistema en el cual algunos mensajes se envíen por usuarios y sean recibidos por otros usuarios.

Un **actor primario** es el principal actor que inicia los casos de uso (UC) para alcanzar el objetivo y cuya satisfacción de la meta es el principal objetivo del caso de uso. El actor primario es un concepto lógico y aunque asumimos que es el que ejecuta el caso de uso, algún agente puede actualmente ejecutarlo en lugar del actor primario. Por ejemplo, un VP puede ser el actor primario para obtener un informe del crecimiento de ventas por región, aunque puede esto actualmente ser ejecutado por un asistente. Consideramos que el actor primario como la persona que actualmente utiliza el resultado del caso de uso y quien sea el principal consumidor del objetivo. El manejo del tiempo es otro ejemplo de como un caso de uso puede ser ejecutado en favor del actor primario, (en esta situación el reporte es generado por algún tiempo).

Note, sin embargo, que aunque la meta del actor primario es manejar las fuerzas dentro de un caso de uso, el caso de uso debe también, cumplir con

las metas de todos las otras partes interesadas, aunque los casos de uso pueden ser manejados por los objetivos del actor primario. Por ejemplo un caso de uso "retirar dinero de un cajero" tiene una persona como el actor primario y será normalmente descripto como toda interacción entre la persona y el cajero. Sin embargo, el banco también es una persona interesada del cajero, y los objetivos pueden incluir tanto los registros de los usuarios, y el dinero sólo será dado si hay suficiente dinero en el estado de cuenta, y que se entregue algún momento en un tiempo, ... El cumplimiento de estos objetivos también deberían ser mostrados por un caso de uno "sacar dinero de un cajero".

Para describir la interacción, un caso de uso usa **escenarios**, un escenario describe un conjunto de acciones que se realizan para lograr algo o ciertas condiciones. El conjunto de acciones se especifica generalmente como una secuencia (por ser la forma más comoda de escribir), las tareas especificadas pueden realizarse en paralelo o en un orden distinto. Cada paso en un escenario es lógicamente una acción realizada por un actor o un sistema. Generalmente, un paso es una acción del actor, algún paso lógico que el sistema debe pasar para cumplir el objetivo, o un estado interno cambiado por el sistema para alcanzar alguna meta.

Un caso de uso siempre tiene un **escenario principal satisfactorio**, que describe la interacción si nada falla y todos los pasos que suceden en el escenario. Puede haber muchos escenarios exitosos. Aunque los casos de uso tienen como objetivo cumplirlos, distintas situaciones pueden surgir mientras el sistema y el actor están interactuando que posiblemente no permitan al sistema lograr completamente su objetivo. Para estas situaciones, los casos de uso también tienen **escenarios excepcionales o de extensión** que describen el comportamiento del sistema si alguno de los pasos principales en el escenario exitoso no se completan satisfactoriamente. A veces son llamados **escenarios excepcionales**. Un **caso de uso** es una colección de todos los buenos y malos escenarios relacionados con el objetivo. La terminología de los casos de uso está resumida en la siguiente tabla.

Término	Definición
Actor	Una persona o un sistema que usa el sistema construido para alcanzar algún objetivo
Actor Primario	El principal actor para el que se crea el caso de uso y cuya satisfacción es el objetivo principal del caso de uso
Escenario	Un conjunto de acciones realizadas para alcanzar el objetivo bajo condiciones específicas.
Escenario Ppal.	Describe la interacción si nada falla.
Escenario de Extensión	Describen el comportamiento del sistema si alguno de los pasos en el escenario principal no se completa satisfactoriamente.

Para alcanzar el objetivo deseado, un sistema puede dividirse en subsistemas. Algunos de estos pueden ser alcanzados por el propio sistema, pero pueden

ser tratados como casos de uso ejecutados por separado con otros actores, que pueden ser de otros sistemas. Por ejemplo el usuario "extraer dinero de un cajero" se usa un servicio de autentificación. La interacción con este servicio puede ser tratada como un caso de uso separado. El escenario en un caso de uso puede por esto emplear otro caso de uso realizando alguna tarea. En otras palabras, los casos de uso permiten una organización jeráquica.

Debiera ser evidente que los fundamentos básicos del modelo que usan casos asumen que el sistema principalmente responde los pedidos por actores que usan el sistema. Por la descripción la interacción entre actores y el sistema, el comportamiento del sistema puede ser especificado, y aunque el comportamiento funcional esté especificado. Una ventaja clave de este enfoque es que los casos de uso se enfocan en el comportamiento externo, así se evitan hacer limpiamente el diseño interno durante el requerimiento, algunas cosas que deseamos pero no es fácil de hacer siguiendo otros modelos.

Los casos de uso son descripciones textuales en lenguaje natural, y representan el comportamiento de los requisitos en el sistema. Esta especificación del comportamiento puede capturar muchos de los requisitos funcionales del sistema. Por esto, los casos de uso no forman una completa SRS, pero pueden formar una gran parte de este. Una completa SRS, como vimos, puede necesitar capturar otros requisitos como performance y restricciones de diseño.

Aunque los casos de uso son detallados textualmente, se pueden usar diagramas para mejorar la descripción textual. Por ejemplo el digrama del caso de uso provee una visión general del caso de uso y los actores en el sistema y sus dependencias. Un diagrama de un caso de uso en UML generalmente muestra cada caso de uso en el sistema como una elipse, muestra los actores primarios para los casos de uso como una figura conectada para cada caso de uso con una línea y muestra la relación entre los casos de uso con arcos entre los casos de uso. Sin embargo, como los casos de uso son básicamente textos en lenguaje natural, los diagramas juegan un rol limitado para desarrollar un específico caso de uso.

3.4.2. Ejemplos

Consideremos la construcción de un pequeño sistema de subastas online para una comunidad universitaria, llamado el sistema universitario de subastas (UAS), a través del cual diferentes miembros de la univerdad pueden vender y comprar bienes. Vamos a asumir que hay un subsistema separado a través del cual se realizan los pagos y el comprador y vendedor tienen una cuenta en él.

En este sistema, aunque tenemos la misma gente que puede comprar y vender tenemos "compradores" y "vendedores" como actores lógicamente separados, donde ambos tienen distintos objetivos a alcanzar. Además de esto, el sistema propiamente dicho es una parte interesada y actor. El sistema financiero es otro.

Primero consideremos los principales casos de uso del sistema:

- "Poner un bien en subasta".
- "Hacer una propuesta".

- "Completar una subasta".

Se muestran a continuación:

- "Poner un bien en subasta".
  - **Actor Primario**: Vendedor
  - **Precondición**: El vendedor se ha logueado.
  - **Escenario Principal exitoso**:
    1. El vendedor postea un Item para la subasta.
    2. El sistema le muestra precios de items similares.
    3. El vendedor especifica el precio inicial y fecha de cierre de la subasta.
    4. El sistema acepta la oferta y lo postea
  - **Escenarios de Excepción**:
    1. No hay elementos similares ofrecidos.
       - El sistema le informa al vendedor la situación.
- "Hacer una propuesta"
  - **Actor Primario**: Comprador
  - **Precondición**: El comprador está logueado.
  - **Escenario Satisfactorio Principal**:
    1. El comprador busca y elige algún item.
    2. El sistema muestra al comprador, el precio inicial, las propuestas, el precio actual y pregunta al cliente por su propuesta.
    3. El comprador especifica su propuesta.
    4. El sistema acepta la propuesta y lo incapacita para subastar en el mismo item.
    5. El sistema actualiza la última propuesta, informa a los otros usuarios y actualiza los registros para ese item.
  - **Escenarios de Excepción**:
    1. El usuario ofrece un precio menor al más alto
       - El sistema informa al usuario esta condición y le sugiere una propuesta más alta.
    2. El comprador no tiene fondos para la oferta que realizó
       - El sistema informa al usuario esta condición y le pregunta si quiere obtener más dinero.

- "Completar una subasta".
  - **Actor Primario**: Sistema de Remates.
  - **Precondición**: Se alcanzó la fecha límite de una subasta.
  - **Escenario Satisfactorio Principal**:
    1. Se elije la mejor oferta, se manda un mail al ofertor, se le vende el producto a ese precio y también se le manda un mail a los otros postores.
    2. Se debita de la cuenta del ofertor el crédito.
    3. Se desbloquean el resto de las cuentas que habían ofrecido dinero.
    4. Se transfiere dinero al vendedor y al sistema.
    5. Se elimina el item del sitio y se actualizan los registros.
  - **Escenarios de Excepción**: None.

Los casos de uso son propiamente explicativos. Este es el principal valor de los casos de uso, que son naturales y son historias que se pueden entender fácilmente por el analista y el lector. Esto ayuda considerablemente a minimizar la brecha de comunicación entre los desarrolladores y las partes interesadas.

Algunos puntos que vale la pena discutir de los casos de uso. Los casos de uso están nombrados para referirse a propósitos. El nombre de un caso de uso especifica el objetivo del actor principal (por lo tanto no hay una línea separada para especificar el objetivo). El actor primario puede ser una persona o un sistema. El actor primario puede ser otro software que requiera un servicio. La precondición de un caso de uso especifica lo que el sistema debe asegurar antes de comenzar para lograr el objetivo. Las precondiciones suelen ser del tipo: "el usuario está logueado" "los datos están en un archivo o estructura de datos" , etc ...

Vale la pena señalar que las descripciones de los casos de uso son listas que contienen tareas que no necesariamente están atadas a los objetivos del actor primario.

Es fácil de entender la idea de situaciones de excepción. Sólo listamos los casos más evidentes. Esta lista puede ser muy larga dependiendo de los objetivos de la organización. Por ejemplo, uno podría ser "el usuario no completó la transacción" y debemos especificar claramente lo que haremos en ese caso.

Un caso de uso puede emplear otro caso de uso para realizar alguna tarea. Los casos de uso permiten una organización basada en el **refinamiento**, es posible ir definiendo funciones más específicas. Por ejemplo, el actor primario para el caso "buscar un item" es el comprador. A su vez mientras listamos los escenarios, nuevos casos de uso y nuevos actores pueden surgir. En el documento de requisitos, todos los casos de uso que se mencionen van a ser necesarios especificar si son o no parte de un sistema ya construido.

3.4.3. Extensiones

Además de especificar el actor primario, el objetivo y el escenario satisfactorio y los escenarios excepcionales, un caso de uso también puede especificar el alcance. Si el sistema que estamos construyendo tiene muchos subsistemas, como suele pasar, a veces algunos casos de usos están explicando el comportamiento de un subsistema. En estas situaciones es mejor especificar el alcance del caso de uso como el del subsistema. Por ejemplo, un caso de uso para un sistema puede ser "estar logueado". Incluso aunque esta parte del sistema, la interacción del usuario con el sistema puede ser a veces necesario aclarar el alcance donde estoy trabajando.

3.4.4. Desarrollando casos de uso

Los casos de uso no son la SRS. La estructura general comprende a 4 etiquetas en lenguaje natural:

- Actores y Objetivos.
- El escenario satisfactorio principal.
- Condiciones de Falla.
- Manejo de errores.

Estos 4 niveles pueden ser una guía para el empleo de los casos de uso.

Paso 1: Identificar los actores y sus objetivos y obtener un acuerdo con las partes interesadas en lo referido a objetos. La lista de actores y objetivos claramente definirá el alcance del sistema y proveerá una visión general de las capacidades del sistema.

Paso 2: Entender y especificar el escenario principal para cada caso de uso, dando más detalles acerca de las principales funciones del sistema. En este paso el analista debe ver si todos los casos cubren las funcionalidades o ver si se necesita algún otro caso de uso y en ese caso se expande la lista de casos de uso.

Paso 3: Cuando se llegó a un acuerdo de los principales pasos en la ejecución del escenario principal, entonces se pueden concretar/analizar las condiciones de falla. Enunciar una lista de condiciones de fallas es una buena tarea para cubrir todas las situaciones erróneas que el sistema debe manejar.

Paso 4: Finalmente, especificaremos que haremos para cada condición de falla.

Para escribir casos de uso, la técnica general para escribir es seguir ciertas normas:

- Gramática simple.
- Debe estar claramente especificado quién realiza cada paso.
- Mantener el conjunto de escenarios lo máximo posible.
- Combinar casos redundantes.

3.5. Otros enfoques para el análisis

El objetivo básico del problema de análisis es obtener un claro entendimiento de las necesidades de los clientes y los usuarios, qué se desea exactamente del software, y qué restricciones tenemos en su solución. Frecuentemente el cliente y el usuario no entienden o no conocen lo que necesitan, porque el potencial del nuevo sistema a veces no se aprecia totalmente. El analista tiene que asegurar que las necesidades reales del usuario y del cliente sean descubiertas, incluso si ellos no las tienen en claro. Es decir, el analista no sólo detecta y organiza la información acerca de la organización del proceso del cliente, sino también puede actuar como un **consultor** jugando un rol activo ayudando al cliente y al usuario a identificar sus necesidades.

El principio básico usado en el análisis es la misma que cualquier tarea compleja: "divide y vencerás". Es decir, partiendo el problema en subproblemas y tratando de entender cada subproblema y su relación con otros subproblemas es un esfuerzo para tratar de entender el problema.

Los conceptos de **estado** y **proyección** pueden a veces ser usados efectivamente en la división del problema. El estado del sistema representa algunas condiciones acerca del sistema. Frecuentemente, cuando se usa estados, el sistema primero es visto como operando en uno de sus posibles estados, y un detallado análisis se realiza para cada estado. Este enfoque es a veces usado en el software de tiempo real o en el control del proceso del software.

En la **proyección**, un sistema está definido desde múltiples puntos de vista. Mientras se usa la proyección, distintos puntos de vista del sistema se definen y el sistema luego es analizado de sus distintas perspectivas. Las diferentes "proyecciones" obtenidas se combinarán para formar el análisis del sistema completo. Analizar el sistema desde distintas perspectivas es a veces, más fácil ya que se enfoca el alcance del estudio.

En el resto de esta sección discutiremos otros dos métodos para el análisis del problema. Como el objetivo del problema es entender el dominio del problema, un analista debe estar familiarizado con distintos métodos y elegir el enfoque que sienta más adecuado para el problema.

3.5.1. Diagramas de flujo

Los diagramas de flujo son comúnmente usados durante el análisis. Los diagramas de datos de flujos (DFDs) son bastante generales y no limitan el análisis del problema para la SRS. Un DFD muestra el comportaminto del sistema a través del flujo de datos. Muestra el sistema como una función que transforma entradas en salidas. Cualquier sistema complejo realizará esta conversión de IN a OUT en varios pasos, y el flujo de datos sufrirá una serie de transformaciones antes de convertirse en el output del programa.

El objetivo del DFD es capturar las transformaciones que toman lugar en el sistema hasta que enventualmente se transforman en la salida del sistema. El agente que realiza la transformación de un estado a otro es llamado proceso o (burbuja). Así un DFD muestra el movimiento de datos a través de distintas

transformaciones o procesos en el sistema. Los procesos se muestran con círculos y los flujos de datos se representan con flechas etiqueteadas. Un rectángulo representa una fuente o sumidero y es un originador o consumidor de datos. El resumidero está fuera del estudio del sistema.

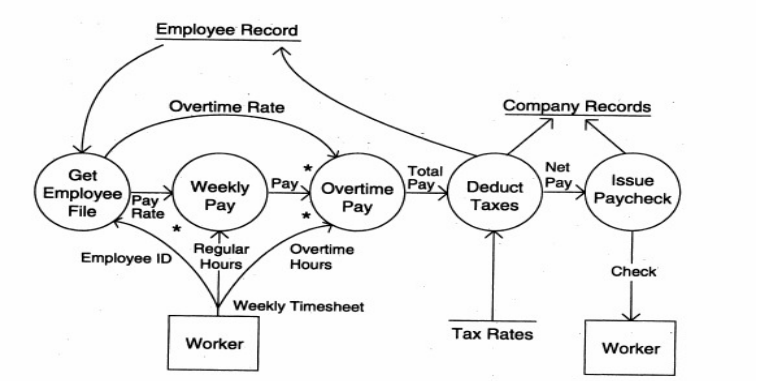

Figura 28: Diagrama de Flujo de Datos de un Sistema de Pago a Trabajadores

En este DFD hay un input básico, el reporte semanal, que se origina del trabajador. El output básico es el cheque, que se entrega al trabajador. En este sistema, primero el empleador registra y lo obtiene usando el ID del trabajador, que está contenida en el reporte. Desde el registro del empleado, se obtiene la tasa y las horas extras trabajadas. Estas tasas regulares y las horas extras (del reporte) son usadas para computar el pago. Luego de que el pago es determinado, se deducen la tasas. Para computar la deducción por tasa, se usa la información del archivo Pago de tasas. El monto de deducción por tasa se registra en los registros del empleado y las compañías. Finalmente, el cheque se usa para efectuar el pago. El monto pagado también se registra en los registros de la compañía.

Explicaremos algunas convenciones dadas para graficar este diagrama. Todos los archivos externos como el registro del empleado, el registro de la compañía, y las tasas se muestran como una línea etiqueteada. La necesidad para múltiples flujos de datos por un proceso se representa con $\star$ entre los flujos de datos. Este símbolo representa la relación AND. Por ejemplo, si hay $\star$, entre dos inputs de flujos de datos A y B, significa: A AND B son necesarios para el proceso. En el DFD, para el proceso **pago mensual** los flujos de datos son "horas" y "el pago de tasas" (ambos) se necesitan, como se ve en el DFD. Similarmente, la relación OR se representa con "+" entre los flujos de datos.

Este DFD es una descripción abstracta de un sistema para manejar pagos. No interesa si es manual o sistematizada. Este diagrama podría tranquilamente ser un sistema donde la computación sólo se hace con calculadoras, y los registros ser carpetas físicas y legajos. Los detalles y los datos menores no se especifican

en este DFD. Por ejemplo, qué sucede si hay un error, no está en el diagrma. Si más detalles son necesarios se puede ir redefiniendo el DFD.

Debiera ser claro que un diagrama no es un flujo de caracteres, el diagrama representa un flujo de datos, mientras que el flowchart muestra el control de flujo. Un DFD no es una representación de los procedimientos en diagramas. Entonces mientras se construye un DFD, uno no debe estar involucrado en los detalles de los procedimientos, no se debe pensar en funciones o procedimientos porque limitaría el diseño. Por ejemplo cuestiones como condiciones o loops deben ser ignoradas. En la construcción o dibujo del DFD el diseñador tiene que especificar las principales transformaciones en el camino de flujo de datos desde la entrada a la salida. Cómo esas transformaciones se realizan no está permitido mientras se dibuja el DFD.

Muchos sistemas son demasiado grandes para un sólo DFD, para describir el procesamiento de datos claramente. Si es necesario puede ser usada alguna descomposición o algún mecanismo de abstracción para esos sistemas. Es decir DFDs pueden ser organizados jeráquicamente, esto ayuda en la partición progresiva y en el análisis de un gran sistema. Estos DFDs a veces son llamados conjunto de diagramas de flujos etiqueteados.

Un conjunto de diagrama de flujo etiqueteado empieza con un DFD, que es una representación bastante abstracta del sistema. A veces, antes del diagrama de flujo inicial, un diagrama contextual puede hacerse en el que sólo se explicita Entrada, salida-Out, fuente, destino. Entonces cada proceso se **refina** y un DFD se realiza para ese proceso. En otras palabras, una burbuja en un DFD se expande en otro DFD durante el refinamiento. Para que la jerarquía sea consistente, es importante que las entradas y salidas para un proceso sean las mismas que las de un proceso a alto nivel. Este refinamiento se detiene, si cada burbuja se considera "atómica". y cada burbuja puede ser entendida fácilmente. Hay que señalar que se mantienen los IN-OUT pero puedo ir cambiando los datos. Es decir, una unidad de datos puede dividirse en partes para ser procesadas, cuando un DFD detallado comienza. Entonces a medida que se descomponene los procesos, la descomposición de datos ocurre.

En un diagrama de flujo de datos, los flujos de datos se identifican con nombre único. Estos nombres se eligen según el funcionamiento de los datos. Sin embargo, para especificar la estructura precisa de los flujos de datos, un **diccionario de datos** suele usarse. El diccionario de asociación precisa la estructura de cada flujo de datos en un DFD. Para definir la estructura de datos, una expresión regular se usa. Mientras se especifican los items se usa terminología como "/", "$\star$", "+".

3.5.2. Diagramas de ER

Los diagramas de entidad relación (ER) se usaron por muchos años para modelar los datos de los aspectos de un sistema. Un diagrama de entidad relación ERD puede ser usado para modelar los datos del sistema y cómo los datos del sistema se relacionan entre ellos, pero no cubre como se procesan los datos o cómo está actualmente el sistema o cómo lo manipula. Es usado a menudo por

diseñadores de bases de datos para representar estructuras de una base de datos, y es una herramienta para analizar el software que emplean bases de datos. El modelo ER forma la lógica del diseño de las bases de datos y puede ser usado fácilmente para construir las tablas iniciales para una base de datos relacional.

Los diagramas de ER tienen 2 conceptos principales y notaciones que los representan. Estos son Entidades y Relaciones. Entidades son las principales fuentes de información o los conceptos en un sistema. Las entidades pueden ser vista como tipos que describen todos los elementos que tienen propiedades en común. Las entidades se representan con Cajas en los diagramas de ER. Con una caja se representan todas las instancias de un concepto o el tipo que la entidad representa. Una entidad es esencialmente equivalente a una tabla de una base de datos o una **hoja** en un **hoja de cálculos**, donde cada línea representa una instancia de una entidad. Las entidades pueden tener atributos, que son propiedades del concepto representado. Los atributos pueden ser visto como las columnas de una tabla y se representan con elipses unidas a la entidad. Para mayor claridad, los atributos a veces no se muestran en una diagrama de ER:

Si todas las entidades están identificadas y representadas, tendremos un conjunto de cajas etiqueteadas en el diagrama. Teniendo una línea entre dos cajas significa que cada elemento de la entidad está relacionado con los elementos de la otra entidad, y viceversa. Esta relación puede tener un nombre etiqueteando la línea. En algunos gráficos esta relación puede ir dentro de un rombo.

Un diagrama de ER especifica algunas propiedades de la relación. En particular puede especificar si la relación es opcional (o necesaria), y cuántos elementos de la entidad están relacionados.

Las relaciones reflejan algunas propiedades del dominio del problema. A continuación mostraremos un digrama de ER del cual podemos distinguir usuarios, categorías, items y ofertas. También están claras las relaciones entre ellas. Un vendedor puede vender muchos items pero cada item sólo tiene una subventa.

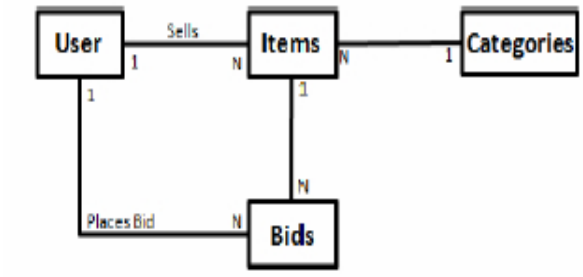

Figura 29: Diagrama de entidad relación para un sistema de subastas

Desde un diagrama de ER, es fácil determinar la estructura inicial en tablas. Cada entidad representa una tabla, las relaciones determinan qué campos en una tabla deben satisfacer esa relación, sumados a los campos por cada atributo. Por ejemplo, para el diagrama de la imagen podemos construir 4 tablas, usuarios,

categorias, items y ofertas. Un usuario se relaciona con items, uno-varios, la tabla de item debe tener ID-USER que lo identifica unívocamente.

Como podemos ver un diagrama ER es complementario a métodos como el uso de casos. Donde los casos de uso se enfocan en la natural interacción y funcionalidades, los diagramas de ER se enfocan en la estructura de las entidades usadas en los casos de uso. Debido a que son naturalmente complementarios, ambos (casos de uso y ERD) pueden ser usados en el análisis de requerimientos de un sistema y ambos pueden estar en una SRS.

3.6. Validación

El desarrollado del software empieza con un documento de requisitos, que también suele ser usado para determinar eventualmente cuando el sistema entregado es aceptable. Por esto es importante que la SRS no contenga errores y especifique los requisitos del cliente correctamente. Además, mientras más largo es el tiempo en que se detecta el error, más grande es el costo en corregirlos, por esto es extremadamente deseable detectar los errores en la fase de requisitos antes de que comience el diseño y el desarrollo del sistema.

Debido a la naturalidad de la fase de especificación de los requerimientos, esto da lugar a muchos malentendidos y a cometer errores, y es posible que la especificación de los requerimientos no aclare exactamente las necesidades del cliente. El objetivo básico de la validación es asegurar que la SRS refleje los actuales requerimientos exactamente y claramente. Un objetivo es chequear que la SRS sea de buena calidad.

Antes de hablar de la validación, vamos a considerar el tipo de errores que típicamente ocurren en una SRS. Muchos errores son posibles. pero los más comunes pueden ser clasificados en 4 tipos:

- Omisión.
- Inconsistencia.
- Hechos incorrectos.
- Ambigüedad.

Omisión: es el error más común en los requerimientos. En este tipo de error, simplemente se omitió algún requisito del cliente, este puede haber estado relacionado al comportamiento del sistema, performance, restricciones de otro factor. La omisión afecta la completitud de la SRS.

La **inconsistencia** se puede provocar debido a contradicciones entre los requerimientos del cliente o con el ambiente en que el sistema va a operar.

Los **hechos incorrectos** son especificaciones de elementos que no pidió el cliente.

La **ambigüedad** sucede cuando algunos requisitos pueden tener distintos significados entonces la interpretación no es única.

Algunas estadísticas sobre errores encontrados en distintos trabajos sugieren:

Omisión	Inconsistencia	Hechos incorrectores	Ambigüedad
26 %	10 %	38 %	26 %

Otra fuente revela que:

Omisión	Inconsistencia	Hechos incorrectores	Ambigüedad
32 %	49 %	13 %	5 %

Estos datos sugieren que los problemas más grandes están en omisiones y ambigüedades.

Como los requisitos son documentos textuales que no pueden ser ejecutados, inspecciones y revisiones son opciones para validar los requerimientos. Debido a que la SRS especifica "formalmente" algo que "informalmente" está en la cabeza del cliente, la validación debe incluir a los clientes y usuarios. Debido a esto, los grupos de validación generalmente están formados por el cliente y usuarios representativos.

La revisión de los requisitos son generalmente revisiones de un grupo para encontrar errores y puntos que no queden claros en el sistema. El grupo de revisión debe incluir el autor de la SRS, alguien que entienda las necesidades del cliente, una persona del equipo de diseño y la persona responsable del mantenimiento de la SRS. Es además una buena práctica incluir un ingeniero que opine sobre la calidad del documento.

Aunque el principal objetivo de la revisión es revelar cualquier tipo de error de la SRS, como discutimos anteriormente, también es considerado como un factor que afecta a la calidad como ser testeable y reducible. Durante la revisión, uno de los trabajos de los **revisores** es encontrar los requerimientos que son subjetivos y difíciles de establecer criterios para testearlos. Durante la revisión, el grupo se va poniendo de acuerdo en los errores encontrados.

La revisión de los requisitos es probablemente la tarea más efectiva para la detección de errores en los requerimientos. Si los requerimientos están escritos en un lenguaje formal es posible el uso de herramientas la validación de los requisitos.

3.6.1. Métricas de Validación (Puntos de función)

Es una métrica como las Líneas de código (LOC). Se determina sólo con la SRS. Define el tamaño en términos de la funcionalidad.

Tipos de funciones	Complejos	Promedio	simple
Entradas externas	3	4	6
Salidas externas	4	5	7
Arhivos Lógicos Internos	7	10	15
Archivos de Interface Externa	5	7	10
Transacciones Externa	3	4	6

- Entradas externas: tipo de entradas (dato/control) externa a la aplicación.
- Salidas Externas: tipo de salida que deja el sistema.

- Archivos lógicos internos: grupo lógico de datos/control de información generada/usada/manipulada.
- Archivos de Interfaz externa: Archivos Pasados/compartidos entre aplicaciones.
- Transacciones externas: Input/Output inmediatos (queries).

La tarea consiste en contar cada tipo de función diferenciando según sea compleja, promedio o simple.

Definimos $C_{i,j}$ como la cantidad de funciones de tipo i con la complejidad j. Luego calculamos el punto de función no ajustado (UFP).

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} W_{i,j} * C_{i,j}$$

Otra idea sugiere ajustar el UFP de acuerdo a la complejidad del entorno. Se evalúa según las siguientes características:

1. Comunicación de datos.
2. Procesamiento Distribuido.
3. Objetivos de Desempeño.
4. Carga en la configuración de Operación.
5. Tasa de transacción.
6. Ingreso de datos on-line.
7. Eficiencia del usuario final.
8. Actualización on-line.
9. Complejidad del procesamiento lógico.
10. Reusabilidad.
11. Facilidad para la instalación.
12. Facilidad para la operación.
13. Múltiples Sitios
14. Intenciones de facilitar cambios.

Cada uno de estos items debe evaluarse como:

No presente	0
Influencia Insignificante	1
Influencia Moderada	2
Influencia Promedio	3
Influencia Significativa	4
Influencia Fuerte	5

Finalmente se define el factor de ajuste de Complejidad (CAF).

$$CAF = 0{,}65 + 0{,}01 * \sum_{i=1}^{14} P_i$$

$$\text{Puntos de Función} = \text{CAF} * \text{UFP}$$

Generalmente 1 punto de función equivale a 125 LOC en C, 50 en Java o C++.

3.7. Resumen

- El principal objetivo del proceso de los requerimientos es producir la especificación de los requerimientos del software (SRS) que adecuadamente captura los requisitos del cliente y que forma las bases del desarrollo del software y la validación.
- Existen 3 tareas básicas en el proceso de los requerimientos
  - Análisis del problema.
  - Especificación.
  - Validación.

  El objetivo del análisis es entender los distintos aspectos del problema, su contexto, y cómo se acomoda con la organización del cliente. En la especificación de los requerimientos, lo entendido del problema es especificado o escrito produciendo la SRS. La validación de los requisitos para asegurar que los requisitos especificados en la SRS son los necesarios y deseados.
- Los aspectos claves y deseables de una SRS son:
  - Correctitud
  - Completa
  - Consistente
  - Inambigua
  - Verificable
  - Rankeada por importancia
  - Rastreable
  - Modificable

- Una buena SRS debe especificar todas las funciones que el software debe soportar, los requisitos de performance del sistema, las restricciones que existen en el diseño y las interfaces externas.
- Los casos de uso son enfoques más populares para especificar los requisitos funcionales.
    - Cada caso de uso especifica la interacción del sistema con el actor primario que comienza el caso de uso para cumplir algún objetivo.
    - Un caso de uso tiene: precondición, escenario principal, algunos escenarios excepcionales, por esta razón preveen el total comportamiento del sistema.
    - Para desarrollar los casos de uso, primero deben identificarse los objetivos y actores, luego los escenarios exitosos, luegos las condiciones de fallas y luego finalmente el manejo de las fallas.
- Con diagramas de flujo de datos, se analiza un sistema desde el punto de vista de qué flujos de datos hay en el sistema. Un DFD consiste de procesos y flujos de datos de los procesos.
- Omisiones, hechos errónes, inconsistencia y ambigüedad son los errores más comunes en una SRS. Para la validación, el método más usado es hacer un grupo estructurado que revise los requisitos

Ejercicios

1. El objetivo básico de la actividad de los requisitos es obtener una SRS que tenga algunas propiedades deseables ¿Cuál es el rol del modelo en el desarrollo de una SRS?
2. ¿Cuáles son los principales componentes de una SRS? ¿Cuál es el principal criterio para evaluar la calidad de una SRS?
3. ¿Cuáles son los principales tipos de errores en los requisitos?
4. Tomar una red social a gusto y listar los principales casos de uso del sistema, con objetivos, precondiciones y escenarios excepcionales.
5. Hacer lo mismo(4) pero con una web de conferencia donde especialistas pueden suber papers que van a ser revisados hasta su última etapa donde son aceptados.
6. ¿Por qué crees que está incluido en la revisión de requisitos un equipo?

4. Arquitectura del Software

Cualquier sistema complejo está compuesto por subsistemas que interactúan bajo el control del diseño del sistema que es el comportamiento que se espera del sistema. Cuando diseñamos un sistema, por lo tanto, lo lógico es identificar los subsistemas y las reglas de interacción entre los subsistemas. Esto es lo que la arquitectura del software ayuda/intenta a hacer.

El área de la arquitectura del software es reciente. Como los sistemas de software distribuidos vienen incrementando y son cada vez más complejos, la arquitectura se volvió un paso importante en la construcción del sistema. Debido al amplio rango de opciones disponibles sobre cómo debe configurarse el sistema, diseñar con sumo cuidado la arquitectura se volvió muy importante. Es decir durante el diseño de la arquitectura donde se elige algún tipo de middleware, o algún tipo de base de datos, o algún tipo de servidor, o se establece alguna componente de seguridad. La arquitectura es también el lugar más temprano donde las propiedades como confianza y performance pueden ser evaluadas para el sistema, esta capacidad se está volviendo cada vez más importante.

En este capítulo discutiremos acerca de:

- Los roles claves que juega la descripción de la arquitectura en un proyecto de software.
- Las múltiples vistas de la arquitectura que pueden ser usadas para especificar distintos aspectos estructurales del sistema que comenzamos a construir.
- Los componentes y conectores arquitectónicos del sistema y cómo pueden expresarse.
- Diferentes estilos propuestos para la visión de Componentes y Conectores que pueden ser usados para diseñar la arquitectura del sistema propuesto.
- Cómo puede ser evaluada la arquitectura de un sistema.

4.1. El rol de la arquitectura del software

¿Qué es la arquitectura? generalmente hablando, la arquitectura de un sistema provee una vista a muy alto nivel de las partes de un sistema, cómo se relacionan para formar la totalidad del sistema. Es decir, la arquitectura divide el sistema en partes lógicas donde cada parte puede ser comprendida independientemente, y luego describe el sistema en términos de esas partes y sus relaciones. Cualquier sistema complejo puede dividirse de distintas maneras, cada una provee una visión y cada una tiene diferentes partes lógicas.

Debido a esta posibilidad de tener múltiples estructuras, una de las definiciones ampliamente más aceptada de arquitectura es **la arquitectura de un sistema es la estructura o estructuras del sistema, que comprenden los elementos del software, las propiedades externamente visibles de esos elementos y las relaciones entre ellos**. Esta definición implica que para

cada elemento en la arquitectura, sólo estamos interesados en sus abstracciones que especifican las propiedades que otros elementos pueden asumir que existen y que son necesarias para especificar las relaciones. Detalles de cómo se soportan esas propiedades no son necesarias para la arquitectura. Esta es una importante capacidad que permite a la descripción de la arquitectura representar un sistema complejo en una forma concisa que puede comprenderse fácilmente.

La descripción de la arquitectura de un sistema será por lo tanto describir las diferentes estructuras del sistema. La siguiente pregunta natural es ¿Por qué debiera un equipo que construye un sistema para algún cliente estar interesado en crear y documentar las estructuras para el sistema propuesto? Alguno de los roles importantes que la descripción de la arquitectura realiza es:

1. **Entendimiento y comunicación**: una descripción de arquitectura se realiza primariamente para comunicar la arquitectura a sus partes interesadas, que incluyen a los usuarios que usarán el sistema, los clientes que encargaron el sistema, los constructores que desarrollarán el sistema, y por supuesto, los arquitectos. Además de esta descripción, las partes interesadas ganan en el entendimiento de algunas propiedades del sistema y cómo intentan cumplir los requisitos funcionales y de calidad. Como la descripción provee un lenguaje común, entre las partes interesadas, es también un vehículo para la negociación y acuerdo entre las partes interesadas, que pueden tener objetivos en conflicto.

2. **Reusar**: la ingeniería del software mundialmente, ha trabajado por largo tiempo para lograr una disciplina donde el software pueda ensamblarse de partes que se desarrollan por distintas personas y estén disponibles para otros usos. Si uno busca construir un software en donde se reusarán algunas componentes, entonces un punto clave de la arquitectura es decidir qué se reusará a alto nivel. La arquitectura debe elegir la manera en que vamos a usar los componentes para que se acomoden apropiadamente con otros. Componentes que podríamos desarrollar. La arquitectura también facilita el reuso de productos de la misma familia. La arquitectura ayuda a especificar que es lo que se acomodará y qué puede variar entre los productos y puede por esto ayudar a minimizar el conjunto de elementos que varíen. Nuevamente, es muy difícil alcanzar este tipo de reuso como un nivel detallado.

3. **Contrucción y evolución**: como la arquitectura divide el sistema en partes, esta división puede naturalmente asignarse a diferentes equipos o personas. Una adecuada partición del sistema permite establecer las partes que se realicen en distintos grupos. Como, al menos por definición, las partes especificadas en la arquitectura son relativamente independientes (la dependencia sólo está en sus relaciones) es posible esto.

4. **Análisis**: es altamente deseable si algunas propiedades del comportamiento del sistema pueden determinarse antes de que empiece la construcción. Esto va a permitir a los diseñadores considerar alternativas, y elegir la que

mejor se acomode a las necesidades. Muchas disciplinas de ingeniería usan modelos para analizar el diseño de un producto por su costo, confiabilidad, performance, etc. La arquitectura abre esta posibilidad también para el software. Es posible, aunque actualmente los métodos no estén totalmente desarrollados o estandarizados, analizar o predecir las propiedades de un sistema desde su arquitectura. Por ejemplo la confianza o Performance de un sistema puede ser analizada. Como el análisis puede ayudar a determinar mejor el sistema que cumplirá con los requisitos de calidad y performance, y sino, qué necesita para hacer cumplir los requisitos. Por ejemplo, mientras construyamos una website para un Shopping, es posible analizar el tiempo de respuesta o rendimiento para alguna arquitectura propuesta, dando algunas asunciones acerca de los pedidos de carga y de hardware. Esto puede ayudar a decidir mejor si la performance es o no satisfactoria, y sino qué nuevas capacidades pueden sumarse para mejorar el nivel de satisfacción.

No todos los usos pueden ser significativos en un proyecto y cuáles de estos usos es pertinente a un proyecto depende de la naturaleza del proyecto. En algunos proyectos la comunicación puede ser muy importante, pero quizás un análisis de performance detallado puede ser innecesario (Porque el sistema es pequeño o será usado sólo por un pequeño grupo de usuarios). En algunos otros sistemas, el análisis de performance puede ser el principal uso de la arquitectura.

4.2. Vistas de la Arquitectura

Hay una visión general emergente de que no hay una única arquitectura de un sistema. La definición que nosotros adoptamos también expresa ese sentimiento. Consecuentemente, no hay un único dibujo del sistema. La situación es similar a la que ocurre en una construcción civil. Para una construcción si querés ver el plan de las plantas, se muestra un conjunto de dibujos. Si sos un ingeniero eléctrico y querés ver cómo está planeada la distribución de la electricidad, vas a ver otro conjunto de dibujos u otro plano. Estos planos no son independientes unos de otros, todos tratan de la misma construcción. Sin embargo, cada plano otorga una visión de la construcción, una visión que se enfoca en explicar un aspecto de la construcción y se enfocará en que ese sea un buen trabajo, y no en los otros aspectos.

Una situación similar existe con la arquitectura del software. En el software los distintos planos son llamados **vistas** (views). Una vista representa al sistema compuesto de algunos tipos de elementos y las relaciones entre ellos. Qué elementos son usados en una vista depende de qué quiere mostrar esa visión. Diferentes vistas exponen diferentes propiedades y atributos, por esto permite a las partes interesadas y al analista evaluar apropiadamente aquellos atributos del sistema. Por enfocarnos sólo en algunos aspectos del sistema, una visión reduce la complejidad al lector, por esto ayuda al entendimiento del sistema y al análisis.

Una vista describe una estructura del sistema. Usaremos estos dos conceptos: vistas y estructuras, intercambiables. También usaremos el término **vista**

arquitectónica para referirnos a una vista. Se han propuesto distintos tipos de vistas arquitectónicas. Muchas de las cuales pertenecen a uno de estos tres tipos:

- Módulo.
- Componentes y Conectores.
- Asignación de Recursos.

En la vista de módulos, el sistema es visto como una colección de unidades de código, cada una implementando alguna parte de la funcionalidad del sistema (**No representa entidades en ejecución)** . Es decir, los principales elementos en esta vista son los módulos. Basada en el código y no explicita ni representa ninguna rutina estructural del sistema. Ejemplos de módulos son: paquetes, una clase, un procedimiento, un método, una colección de funciones o una colección de clases. Las relaciones entre estos módulos también están basadas en el código y depende de cómo el código de un módulo interactúa con otros módulos. Ejemplos de relaciones en esta vista son: "es parte de", "usadas o depende de", "generalización o especificación".

En la vista de componentes y conectores, el sistema es visto como una colección de entidades llamadas componentes. Un componente es una unidad que tiene una identidad en la ejecución del sistema. Objetos (no clases), una colección de objetos, y procesos son ejemplos de componentes. Ejemplos de conectores son pipes y sockets. Compartir datos puede también actuar como conector si los componentes usan algún middleware para comunicarse y coordinar, entonces el middleware es un conector. Por lo tanto, los elementos primarios de esta vista son componentes y conectores.

La vista de asignación de recursos se enfoca en cómo las distintas unidades de software se acomodan a recursos como el hardware, archivos y la gente. Es decir esta vista especifica la relación entre los elementos del software y los elementos del ambiente en que el sistema se ejecutará. Expone propiedades estructurales como qué procesos corren en qué procedimiento, y cómo el sistema de archivos está organizado.

La descripción de la arquitectura consiste en vistas de diferentes tipos, donde cada vista expone alguna estructura del sistema. La vista de módulos muestra cómo el software está estructurado como un conjunto de implementación de unidades, la vista de C&C muestra cómo el software está estructurado, cómo interactúan los elementos en la rutina y la vista de asignación de recursos muestra como el software se relaciona con las estructuras del nonsoftware. Estos tres tipos de vistas forman la arquitectura del sistema.

Note que las distintas vistas **no son ajenas**. Todas ellas representan el mismo sistema. Por lo tanto, hay relaciones entre elementos de distintas visiones. Estas relaciones pueden ser simples o muy complejas. Por ejemplo, la relación entre módulos y componentes puede ser uno a uno en el módulo que implemente un componente. Por otro lado, puede ser que un módulo esté usado por muchos componentes y un componente use muchos módulos. Mientras creamos las distintas vistas los diseñadores tienen que tener cuidado de estas relaciones.

La próxima cuestión es ¿Cuáles son las vistas estándar que debieran expresarse para la descripción de la arquitectura del sistema? Para responder esta pregunta, puede ayudarnos la analogía con la construcción. Si uno está construyendo una casa pequeña, entonces quizás no sea necesario tener una vista separada de salidas de emergencia y de incendios. Similarmente si no hay aire acondicionado en un edificio no hay necesidad de agregarlo a los planos. Por otra parte, una construcción de una oficina puede requerir ambas.

La situación con el software suele ser similar. Como dijimos, depende de qué propiedades nos interesen, distintas vistas de la arquitectura podemos necesitar. Sin embargo, de estas vistas, la de C&C es por defecto la principal, la que siempre se realiza cuando se hace la arquitectura incluso algunos definen la arquitectura como la vista de C&C de un sistema.

Una pregunta natural que surge es ¿En qué se diferencia la arquitectura del diseño si ambas buscan los mismos objetivos y se basan fundamentalmente en la idea "divide y vencerás"? Primero, debiera ser claro que la arquitectura es un diseño que se encuentra en el dominio de la solución y trata acerca de la estructura del sistema propuesto. Además, la arquitectura otorga una visión a alto nivel del sistema basándose en la abstacción para transmitir lo principal, algo que el diseño también hace. Entonces, la arquitectura es diseño.

Podemos ver la arquitectura como un diseño a muy alto nivel, enfocado como la primera etapa del diseño. Lo que denominaremos diseño es realmente aquello acerca de los módulos que eventualmente existen cómo código. Es decir son una representación más concreta de la implementación, aunque no lo sean. Consecuentemente durante el diseño a bajo nivel temas como la estructura de datos, archivos, tienen que abordarse mientras que esos temas no son relevantes a nivel de la arquitectura. También podemos ver el diseño como el que provee la vista de los módulos de la arquitectura del sistema.

Los límites entre la arquitectura y el diseño de alto nivel no están totalmente claros. La manera en que este campo ha evolucionado tiende a separar cada vez más diseño de arquitectura. A nivel de la arquitectura hay una necesidad de mostrar sólo lo que necesitemos, para la evaluación. La estructura interna de esas partes no es importante. Por otro lado, durante el diseño, el diseño de las estructuras de las partes puede llevarnos a construir alguna de las tareas claves. Sin embargo, qué partes de la estructura deben ser examinadas y realizadas durante la arquitectura y qué partes durante el diseño es cuestión de elección. Generalmente hablando, detalles que no son necesarios para realizar los tipos de análisis que requieran en el tiempo de arquitectura son innecesarios y deben ser dejados para la fase de diseño.

4.3. Vista de Componentes y Conectores

La vista de C&C tiene dos elementos principales componentes y conectores. Componentes son usualmente elementos computables o almacenamiento de datos que tienen alguna presencia durante la ejecución del sistema. Conectores definen las principales maneras de comunicación entre los componentes y qué componente está conectado con cual a través de un conector.

Una vista de C&C describe la estructura en tiempo de ejecución del sistema, qué componentes existen cuando el sistema está en ejecución y cómo estos interactúan durante la ejecución. La estructura de C&C es esencialmente un grafo, donde los nodos son las componentes y los conectores los lados.

Muchas descripciones de arquitecturas sólo se contienen la vista de C&C, de allí su importancia.

4.3.1. Componentes

Componentes son generalmente unidades de computación o almacenamiento de datos en el sistema. Un componente tiene un nombre, que generalmente se elige para representar el rol del componente o la función que realiza. El nombre también provee una identidad única al componente, que es necesaria para referenciar detalles acerca del componente en los documentos de soporte como el dibujo de C&C muestra sólo los nombres de los componentes.

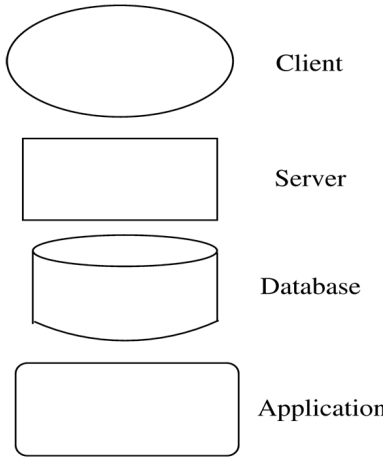

Figura 30: Ejemplos de Componentes

Un componente es un tipo de componente, donde el tipo representa un componente genérico, definiendo la computación general y las interfaces que un tipo de componente debe tener. Note que aunque un componente tiene un tipo, en la vista arquitectónica de C&C, tenemos componentes (es decir instancias

actuales) y no tipos. Ejemplos de estos tipos son: clientes, servers, filters, etc. Diferentes dominios pueden tener otro tipo genérico como controladores, actores y emisores (para el control del sistema de dominio).

En un diagrama que representa la vista arquitectónica de C&C de un sistema, es altamente deseable tener una representación diferente para distintos tipos de componentes, entonces podemos identificar visualmente los diferentes tipos. En un diagrama de cajas y líneas, a veces todos los componentes se representan con cajas rectangulares. Como el enfoque requerirá que los tipos de componentes se describan separadamente y el lector tiene que leer la descripción de las figuras con los tipos de componentes distintos. Algunos de los símbolos usados para representar comúnmente se encuentran en la figura.

Para asegurarse de que los principales símbolos de los distintos tipos de componentes, es deseable tener un conjunto clave de símbolos para describir qué tipo de componente representa el símbolo.

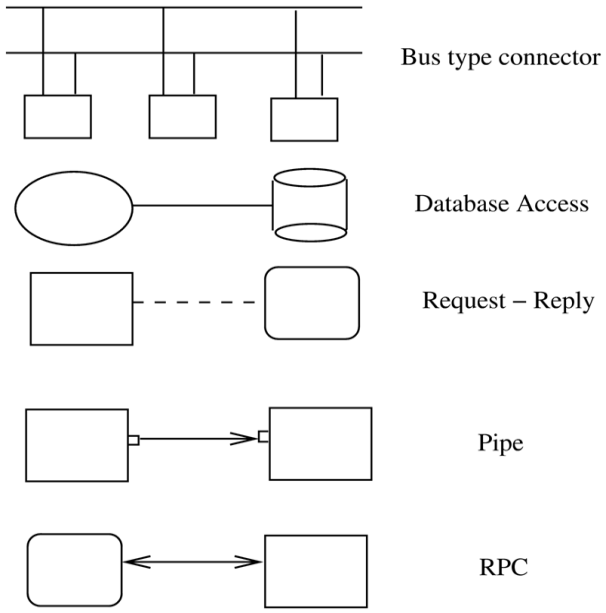

Figura 31: Ejemplos de Conectores

4.3.2. Conectores

Después de todo, los componentes existen para proveer partes del servicio y características del sistema, y estos deben combinarse para desarrollar el conjunto de funcionalidades del sistema. Para componer un sistema desde sus componentes la información acerca de la interacción entre los componentes es necesaria.

La interacción entre componentes puede ser cubierta a través de una descripción simple de los procesos subyacentes en la ejecución o el funcionamiento del sistema. Por ejemplo, un componente puede interactuar usando una llamada a procedimiento (un conector), que es provisto por el funcionamiento de un lenguaje de programación. Sin embargo, la interacción puede ser más compleja que sólo mecanismos. Ejemplos de estos mecanismos remotos pueden ser puertos TCP/IP, o protocolos como HTTP. Estos mecanismos requieren una buena cantidad de rutinas subyacentes a la infraestructura, así como una especial programación con los componentes para usar la infraestructura. Consecuentemente, es extremadamente importante identificar y explicitar la representación de los conectores. La especificación de los conectores nos ayudará a identificar una adecuada infraestructura necesaria para implementar la arquitectura, así como para clarificar las necesidades de programación para los componentes que la usen.

Note que los conectores no necesitan ser binarios y un conector puede proveer una comunicación n-aria entre múltiples componentes. Por ejemplo, un bus de broadcast puede ser usado como conector, que le permite a los componentes enviar un mensaje a todos los otros componentes. Algunos de los símbolos comunes usados para representar comúnmente los conectores se muestran en la figura.

Un conector también tiene un nombre que debiera describir la naturaleza de la interacción que el conector soporta. Un conector también tiene un tipo, que es una descripción genérica de la interacción, especificando propiedades como si es binario o n-arios, tipos de interfaces que soporta, etc. Si un protocolo es usado por un tipo de conector, debiera explicitarse.

Es una pena señalar que la implementación de un conector puede ser muy compleja. Si el conector es provisto por el sistema subyacente, entonces los componentes sólo tienen que asegurar que ellos usen los conectores a la par de sus especificaciones. Si, no obstante, el sistema subyacente no provee un conector usado en la arquitectura entonces, como mencionamos antes, el conector tendrá que implementarse como parte del proyecto de construcción del sistema. Es decir durante el desarrollo, no sólo las componentes necesitan ser desarrollados, sino también tendremos que asignar recursos al desarrollo de los conectores. Generalmente, mientras creamos la arquitectura, deseamos que nuestra arquitectura use conectores disponibles en el sistema que luego se entregará.

4.3.3. Un Ejemplo de C&C

Suponga que tenemos que diseñar y construir un sistema simple para realizar encuestas de manera online a estudiantes de un campus de una universidad. La encuesta consiste de un conjunto de multiple-choice, y el sistema deberá permitirle al estudiante que pueda completarlo y mandarlo de manera online.

También queremos que cuando los usuarios suban el formulario, el sistema le muestre el actual resultado de la investigación, es decir qué porcentaje de estudiantes llenaron qué respuesta con el porcentaje asociado por pregunta.

La mejor manera de construir el sistema es a través de la Web, así es como lo eligiría cualquier desarrollador.

Para este ejemplo simple se propone una arquitectura ternaria.

Este consiste de un cliente al que se le mostrará un formulario que un estudiante puede completar y enviar, y también les mostrará los resultados actuales. El segundo componente es el servidor, que procesa la información enviada por el estudiante y guarda esto en la base de datos que es el tercer componente. El servidor también consulta a la base de datos para obtener el resultado de las consultas y envía el resultado en un formato adecuado (HTML).

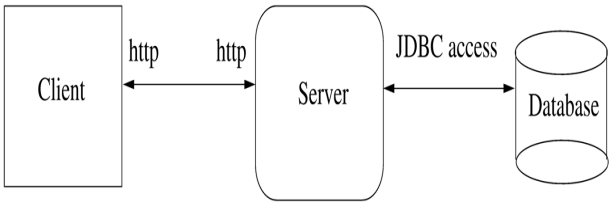

Figura 32: Vista de Componentes y Conectores del sistema de encuestas.

Note también que el cliente, servidor y la base de datos son todos distintos tipos de componentes, y por esta razón se muestran usando diferentes símbolos. A su vez la figura muestra que los conectores entre componentes son también de diferentes tipos. El diagrama muestra claramente las diferencias, esquema que puede comprenderse fácilmente.

Notar que a nivel de arquitectura no están incluidos los detalles del Host. ¿Cuál es la URL? ¿En qué lenguaje se escribirá? Esas cuestiones no pertenecen a la arquitectura.

Note también que el conector entre el cliente y el server explicita el uso de HTTP. El diagrama también dice que es un cliente Web. Esto implica y asume que estará corriendo en un server HTTP y que el cliente que realice la encuesta lo hará por medio de un browser.

Hay algunas implicaciones de elegir este conector entre los componentes. El cliente deberá escribir la respuesta de manera que pueda mandarse un Request usando HTTP (esto implicará el uso de algún formulario HTML). Similarmente, también implica que el server tiene que tomar el pedido del HTTP server en el formato especificado por ese protocolo. Además el cliente debe responder al cliente mediante otro HTML. Por lo tanto cuando se acepta finalmente la discusión, las implicaciones para la estructura como para la implementación debe ser totalmente entendidas y las acciones deben ser tomadas para asegurar que esas asunciones son válidas.

La arquitectura presentada arriba no tiene seguridad y un estudiante puede hacer muchas veces la encuesta si lo desea, incluso alguien que no sea estudiante puede hacer la encuesta. Ahora suponga que se busca que el sistema pueda

ser abierto sólo por estudiantes registrados, y cada estudiante pueda hacer la encuesta a lo sumo una vez. Para identificar a los estudiantes se explicó que cada estudiante tiene una cuenta, y su información de cuenta está disponible en el server o el instituto.

Ahora debemos cambiar la arquitectura. La arquitectura ahora debe separar el formulario de login del usuario y una componente separada del server que realiza la autenticación. Para la validación, nos dirigimos al proxy server para chequear si el login y el password provisto es válido. Entonces se devuelve una cookie al cliente que se guarda con el protocolo de cookies. Cuando el cliente complete la encuesta, la cookie valida al usuario, y el server chequea si el cliente ha completado la encuesta. La arquitectura para este sistema se muestra en la siguiente figura:

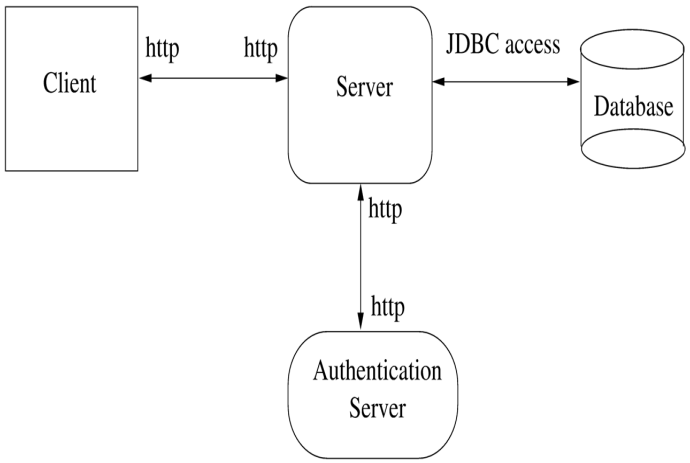

Figura 33: Arquitectura para el sistema de encuestas con autenticación

Note que incluso diciendo que la conexión entre el cliente y el server es en HTTP, hay alguna distinción con la arquitectura anterior. En la primera arquitectura, HTTP plano es suficiente. En esta también se necesitan cookies, el conector ahora es realmente: HTTP + Cookies. Entonces, si el usuario deshabilita las cookies, el requisito del conector no está disponible y el sistema no funcionará.

Ahora suponga que el sistema sea extendido de distintas maneras. Se encontró que el servidor es desconfiable y que frecuentemente se cae. Se estimó también que cuando el sistema muestre los resultados de las encuestas luego de enviar el formulario, alguna respuesta no puede estar totalmente actualizada, y esto puede aceptarse hasta cierto límite. Ahora asumimos que el resultado puede estar desactualizado sólo 5 encuestas. Incluso si se cayó el sistema, entonces debe proveerse una facilidad para esto.

Para hacer el sistema más confiable, se propone la siguiente estrategia. Cuando los estudiantes mandan la encuesta, el server interactúa con la base de datos como antes, si se cayó la base de datos o no está disponible, los datos de la encuesta se guardan localmente en una cache (Componente), y el resultado obtenido en la cache se usa para mostrar al cliente. Esto sólo se hace para 5 encuestas luego de esto el sistema no va a aceptar más.

Entonces, tenemos, otro componente en el sistema llamado Cache Manager, y la conexión entre el server y este nuevo componente es del tipo call/return. Esta arquitectura se muestra en la siguiente figura.

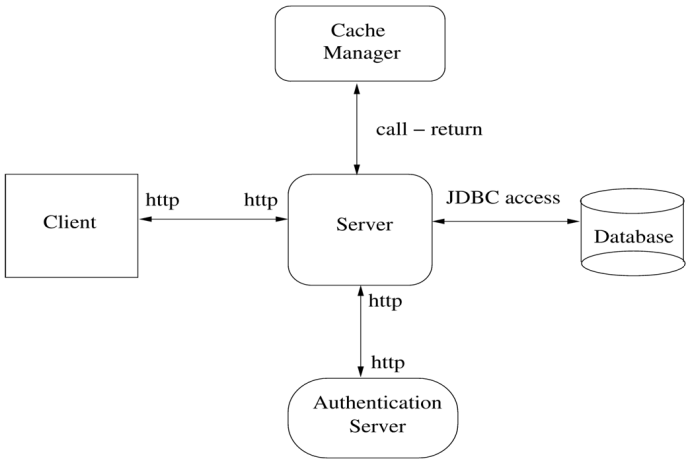

Figura 34: Arquitectura para el sistema de encuestas con caché

Debiera ser claro que mediante el uso de cache, la disponibilidad del sistema aumenta. La caché también tendrá un impacto en la performance. Esta extensión de la arquitectura muestra como la arquitectura también afecta a cuestiones como disponibilidad y performance, y cómo la elección de una apropiada arquitectura puede ayudar a cumplir los objetivos de calidad (o sólo a aumentar la calidad al sistema). Luego discutiremos una manera formal de la realización de una evaluación en estas arquitecturas para ver cómo impacta en algunos atributos de la calidad.

4.4. Estilos de Arquitectura para la vista de C&C

Debiera ser claro que diferentes sistemas tendrán diferentes arquitecturas. Hay algunas arquitecturas generales que se observan en muchos sistemas y que representan una estructura general para una clase de problemas. Estos son llamados **estilos arquitectónicos**. Un estilo define una familia de arquitecturas que satisfacen una determinada restricción. En esta sección discutiremos acerca de algunos estilos comunes para la vista de C&C que pueden ser usados para una gran cantidad de problemas. Los estilos pueden proveer ideas para crear una vista de arquitectura para el problema que tengamos. Los estilos también pueden combinarse para formar vistas más ricas o mejores.

4.4.1. Pipe and Filter

El estilo arquitectónico de pipe and Filter se acomoda muy bien para sistemas que principalmente hacen una transformación de datos, transformación por la cual se recibe algún dato de input y el sistema debe transformarlo adecuadamente para producir alguna salida. Un sistema con esta vista alcanza su objetivo mediante una red de transformaciones pequeñas y componiéndolas de manera que juntas alcancen la transformación deseada.

Este estilo sólo tiene un tipo de componente llamado **Filtro** (Filter). También tiene un solo tipo de conector llamado **Conducto** (Pipe). El filtro realiza una transformación de datos, y envía la información transformada a otro filtro para un futuro procesamiento usando un conector Pipe. En otras palabras, un filtro recibe datos que necesita de algún Conducto definido, realiza la transformación y envía los datos por un Conducto de salida definido. Un Filtro puede tener más de un input y más de un Output. Los filtros pueden ser asíncronos y ser entidades independientes y como sólo están preocupadas por lo que llega en el Pipe, el filter no necesita saber la identidad del filter que manda los datos ni la identidad del que luego va a consumir la información producida por él.

El conector Pipe es un canal unidireccional que transmite un stream de datos recibidos de una punta de la conexión a otra. Un pipe no cambia los datos de ninguna manera sólo transmite los datos en el orden que los recibe. Como los filtros pueden ser asíncronos y debieran trabajar sin el conocimiento de la identidad del productor y el consumidor, un buffering y sincronización son necesarias para asegurar un funcionamiento silencioso en la relación consumidor provista. Los filtros simplemente consumen y producen datos.

Hay algunas restricciones que impone el estilo:

- Primero, como se mencionó arriba los filtros debieran trabajar sin conocer la identidad del consumidor o productor, sólo debieran necesitar los datos.
- Segundo, un pipe, que es un conector de 2 caminos, debe conectar un puerto de salida de un filter con el de entrada de otro filter.

Una estructura pura de Pipe and Filter tendrá también restricciones que un filter sea de un hilo independiente que procese los datos a medida que lleguen. Implementar esto requerirá esencialmente una infraestructura para soportar el

mecanismo de Pipe con buffers que realicen tareas de sincronización necesarias. Para el uso de pipe, el constructor de un filtro debe tener precaución de todas las propiedades del Pipe, particularmente con miras al buffering y a la sincronización, mecanismos de Input/Output y las banderas para terminar un dato.

Sin embargo, puede haber situaciones en que un filtro no necesita procesar los datos a medida que lleguen. Sin este tipo de restricciones, el estilo de Pipe and Filter puede tener filtros que produzcan los datos completamente antes de enviarlos, o que recién se empiece a procesar los datos cuando estén "completos". En algunos sistemas los filtros no pueden operar concurrentemente, como en los sistemas tipo Batch. Sin embargo eso puede simplificar los Pipes y los mecanismos que estos soporten.

Consideremos un ejemplo, para un sistema que necesita contar la frecuencia de diferentes palabras en un archivo. Una arquitectura que use el estilo de Pipe and Filter este sistema se muestra en la figura siguiente:

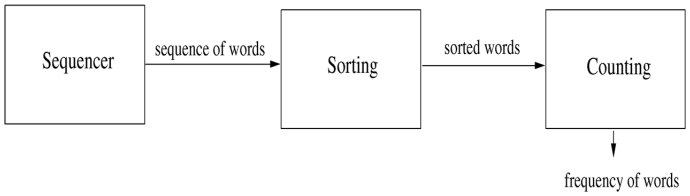

Figura 35: Ejemplo de Pipe and Filter

Esta arquitectura propone que el input data debe ser divido en una secuencia de palabras para el componente Sequencer. Esta secuencia de palabras debe ser ordenada por el componente Sort que manda estas palabras ordenadas a otro filtro Counter un contador del número de ocurrencias de difentes palabras. Esta estructura de ordenar las palabras antes de contar se determinó para mejorar la eficiencia.

Como puede verse en el ejemplo, el estilo de arquitectura Pipe and Filter se acomoda bien para el procesamiento y transmisión de datos. En consecuencia, es usado en aplicación de procesamiento de texto, de señales, codificación, corrección de errores y otras transformaciónes de datos.

El estilo Pipe and Filter permite, debido a sus restricciones, componer la estructura general del sistema en pequeñas transformaciones. O viéndolo de otra manera, permite factorizar la transformación deseada en transformaciones más pequeñas y para esto se construyen los filtros. Es decir, permite utilizar técnicas de composición y descomposición funcional algo que es matemáticamente conmovedor.

4.4.2. Estilo de datos Compartidos

En este estilo hay dos tipos de componentes: los repositorios y los usuarios de datos . Los componentes del tipo repositorio de datos son aquellos donde el sistema almacena los datos compartidos, estos pueden ser archivos o bases de datos. Estos componentes proveen un confiable y permanente depósito, tiene cuidado de cualquier sicronización necesaria para el acceso concurrente, y proveen el acceso a los datos. Componentes de tipo usuarios de datos, acceden a los datos desde los repositorios, realizan la computación de los datos obtenidos, y si quieren compartir los datos con otros componentes ponen los datos de nuevo en el repositorio. En otras palabras, los usuarios de datos son elementos computacionales que reciben datos del repositorio y guardan luego los datos en el repositorio. Estos componentes no se comunican directamente unos con otros, sólo lo pueden hacer mediante la transferencia de datos en el repositorio.

Hay dos variaciones posibles a este estilo: En el estilo **Pizarra**, si algún dato se postea en el repositorio, todos los que acceden a los datos necesitan saberlo. En otras palabras, este tipo de posteo de datos se encarga de avisar a todos los componentes de esto y cuando empiece la ejecución de estos componentes necesitan actualizar sus datos. En las bases de datos esto suele implementarse con disparadores.

El otro es **el estilo repositorio**, en el cual el repositorio es pasivo, que permanentemente provee la capacidad de almacenamiento y acceso a los datos. En este estilo los componentes acceden al repositorio cuando quieran.

Como se puede imaginar, muchas aplicaciones de bases de datos usan este estilo de arquitecturas. Las bases de datos, pensadas originalmente como repositorios, ahora actúan como ambas o bien como repositorios o **Pizarra** y proveen disparadores que permiten que esto se implemente eficientemente. Muchos sistemas web siguen este estilo al final, para responder a los pedidos de los usuarios, diferentes scripts (usuario de datos) acceden y actualizan algún dato compartido.

Como un ejemplo de un sistema que use esta arquitectura, consideremos un sistema de registros de estudiantes de una universidad. El sistema claramente tiene un repositorio central que contiene la información de los cursos, estudiantes, correlativas, etc. Tiene un **administrador** que setea el repositorio, los derechos de distintas personas, el componente **registration** permite a los estudiantes registrarse y actualizar la información para los estudiantes y los cursos. La **aprobación** de los cursos es un componente que garantiza la aprobación de los cursos que requiere el consentimiento del profesor. El componente **Reports** produce una vista a los estudiantes para ver los distintos cursos al final de la registración. El componente **Course Feedback** se usa para tomar una respuesta de los estudiantes al fin del curso. Esta arquitectura se muestra en la siguiente figura.

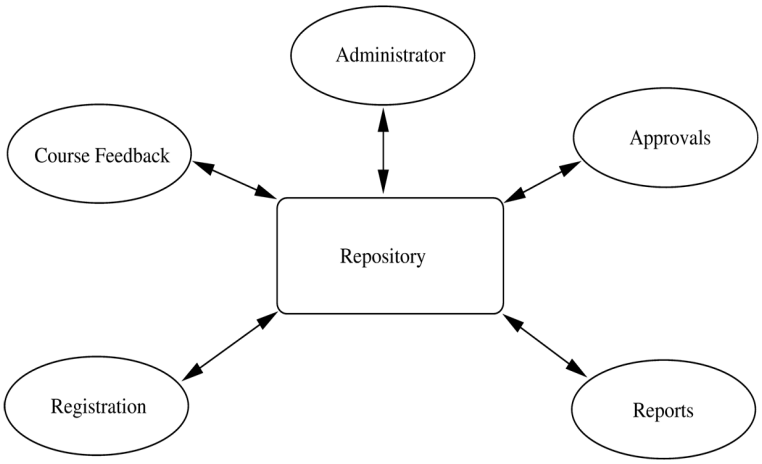

Figura 36: Ejemplo de Shared Data

Note que todos los diferentes componentes no necesitan comunicarse uno con otro, y ni siquiera necesitan saber de la presencia de los otros. Por ejemplo si luego se desea agregar otro componente que calendarice los distintos cursos, se podría agregar fácilmente. No es necesario cambiar ningún componente.

Hay un solo tipo de conector en este estilo **Read/Write**. Note, sin embargo, que es general, podría especificarse puntualmente el tipo de lectura o de escritura. Por ejemplo que cada escritura sea atómica o que se leyera de algún archivo, directorio, línea de código especial.

4.4.3. Estilo Cliente-Servidor

Otro estilo comúnmente usado en la actualidad para construir sistemas es el de cliente Servidor. En este estilo, hay dos tipos de componentes: clientes y servidores. Una restricción de este estilo es que los clientes sólo pueden comunicarse con el servidor y no pueden comunicarse con otros clientes. La comunicación entre el cliente y el servidor se inicia con un pedido de algún servicio por parte del cliente al servidor que lo soporta. El servidor recibe el pedido por un puerto definido, realiza el servicio y devuelve el resultado de la computación al cliente que solicitó el servicio.

Hay un solo tipo de conector en este estilo: el tipo **Request/Reply**. Un conector une el cliente al servidor. Este tipo de conector es asimétrico, el cliente sólo puede realizar pedidos (y recibir respuestas), mientras que el servidor sólo puede responder a los pedidos que recibe por el conector. La comunicación es frecuentemente síncrona, el cliente espera la respuesta del servidor. Es decir, el cliente se bloquea ante un pedido hasta que obtiene una respuesta.

Una forma general de esta estructura es una de n-niveles. En este estilo, el cliente envía el producto al servidor, pero el servidor en orden en que recibe manda algún pedido a otro servidor. Es decir, el servidor también actúa de

cliente para el nivel siguiente. Esta jerarquía puede contar por algunos niveles, llegando a un sistema de n-niveles. Un ejemplo común de esta arquitectura es uno de 3 niveles. En este estilo, el cliente que hace el pedido recibe respuesta del cliente del tercer nivel. Los niveles/nivel, intermedios llamados Bussines level, contienen los componentes que procesan los datos por el cliente y aplican necesarias regalas de bussines. El tercer nivel es la base de datos, en donde residen los datos, el bussines interactúa con el tercer nivel para todos los datos que necesita.

La mayoría de las veces, en una arquitectura de cliente-Servidor, el componente server y el componente cliente residen en distintas máquinas. Incluso si están en la misma máquina, está diseñado para que pueda existir en máquinas distintas. Por lo tanto, el conector entre el cliente y el servidor se espera que soporte el tipo **Request/Reply** en diferentes máquinas. Entonces los conectores son internamente muy complejos y soportan un montón de características para trabajar a través de la red. Muchos clientes-servidores usan puertos RCP para sus conectores. La web usa HTTP para soportar los conectores.

Note que hay una distinción entre **Layered Architecture** y una arquitectura por Niveles. La Arquitectura Layered es una visión de módulos que proveen como los módulos sean organizados y usados, es decir sólo puedo comunicar módulos si pertenecen al mismo Layered. Por lo tanto pueden combinarse. Por ejemplo, en una arquitectura cliente-servidor. El server puede tener una distribución de los componentes organizadas con una arquitectura tipo Layered y el sistema general puede tener múltiples niveles, donde uno de estos sea ese servidor.

4.4.4. Algunos otros estilos

- **Estilo Publicar-Suscribirse**: en este estilo hay dos componentes, un tipo de componente que se suscribe a un conjunto definido de eventos. Otro tipo de componentes que generan o publican eventos.
- **Peer To Peer**, oriented Object Style: donde cada cliente es además de cliente un server.
- **Estilo de comunicación entre procesos**: consiste de múltiples hilos de comunicación usados para grandes sistemas.

4.5. Diseño de la documentación de la arquitectura

Hasta aquí hemos enfocado en representar las vistas a través de diagramas. Mientras diseñamos, los diagramas son ciertamente un buen camino para explorar opciones y fomentar discusiones y realizar una lluvia de ideas entre los arquitectos. Pero cuando terminó el diseño, la arquitectura debe comunicarse apropiadamente a todas las partes interesadas en el acuerdo. Esto requiere que la arquitectura sea un documento preciso con suficiente información para realizar los tipos de análisis de diferentes partes interesadas, es decir para abarcar las preocupaciones de los clientes totalmente.

Sin un apropiado documento de descripción de la arquitectura, no es posible llegar a un completo entendimiento. Por lo tanto es importante crear una apropiada documentación de la arquitectura. En esta sección discutiremos lo que un documento de arquitectura debe tener.

Como diferentes proyectos requerirán distintas vistas, diferentes proyectos necesitan un nivel distinto de detalles en la documentación de la arquitectura. En general, sin embargo, el documento que describe la arquitectura debe contar con lo siguiente:

- Contexto e la arquitectura y del sistema.
- Descripción de la vista de la Arquitectura.
- Documentación a través de las vistas.

Sabemos que una arquitectura para un sistema es manejado por los objetivos y necesidades de las partes interesadas. Por lo tanto, el primer aspecto que un documento de arquitectura debe contener es la identificación de las partes interesadas y sus preocupaciones. Esta porción debe dar una visión general del sistema, las distintas partes interesadas y las propiedades para las cuales evaluaremos la arquitectura. Un **diagrama contextual** que establece el alcance del sistema, sus límites, y los principales actores que interactúan con el sistema, y fuentes y destinos posibles de los datos. Un diagrama contextual frecuentemente se representa con el sistema en el centro y mostrando las relaciones con los usuarios y los datos.

Con el diagrama contextual definido el documento puede proceder con la descripción de las diferentes estructuras o vistas arquitectónicas. Como establecimos antes, puede ser que necesitemos de distintas vistas para el mismo proyecto y las necesidades dependen del proyecto y las partes interesadas. La descripción de la arquitectura siempre debe contar con un gráfico que es la principal descripción de la vista.

Sin embargo una representación gráfica no es una descripción completa de la visión. Esto da una idea intuitiva del diseño, pero no es suficiente detallada. Por ejemplo, el propósito y funcionalidad de un módulo o un componente es indicado sólo por el nombre en el gráfico lo cual no es suficientemente detallado. Por lo tanto el documento puede tener algunas de las siguientes aclaraciones:

- Catálogo de elementos: con más información acerca de los elementos mostrados en la figura. Además de describir el propósito de los elementos también puede incluir sus interfaces, sintáctica (nombrarlos) y semántica (describir que hace en la interfaz).
- Fundamento de la arquitectura: explica las razones fundamentales de la elección de los distintos puntos elegidos y componentes elegidos. Cuáles se descartaron.
- Comportamiento: alguna representación del comportamiento del sistema. A su vez puede incluir el diagrama de secuencia que se explicará en la siguiente sección.

- Otra información: esto puede incluir una descripción de todas las decisiones que no fueron tomadas durante la creación de la arquitectura pero que deben ser tomadas más adelante, por ejemplo, la elección del servidor o de un protocolo, dando alguna razón para la elección de alguna de ellas.

Sabemos que las distintas arquitecturas están relacionadas. En cuanto a la discusión hasta aquí describimos las vistas de arquitecturas independientemente. El documento de arquitectura podría además de la descripción separada de las vistas describir las relaciones entre ellos. Esencialmente, esta documentación describe las relaciones entre las distintas vistas. A su vez puede incluir un justificativo de la elección de las distintas vistas.

Sin embargo, la relación entre las distintas vistas puede ser demasiado simple o muy completa. En el primer caso, las vistas pueden verse casi iguales y la descripción separada de las vistas llevará a la repetición. En estas situaciones, por razones prácticas es mejor **combinar diferentes vistas en una**. Además de eliminar la duplicación este enfoque puede ayudar a clarificar las relaciones entre las vistas.

La estructura general propuesta aquí es una guía para organizar el documento de la arquitectura. Sin embargo, el principal propósito de este documento es claramente comunicar la arquitectura a las partes interesadas y si alguna de estas sugerencias puede ser redundante es recomendable no agregarla.

Pasando en limpio nos quedaría:

Organización del documento de Arquitectura

1. Contexto del sistema y la Arquitectura.
2. Descripción de ls vistas de la arquitectura
   a) Presentación principal de la vista.
   b) Catálogo de elementos.
   c) Fundamento de la Arquitectura.
   d) Comportamiento.
   e) Otra información.
3. Documento Transversal a las vistas.

4.6. Evaluación de la Arquitectura

La arquitectura de un sistema impacta en algunos atributos no funcionales de la calidad del software, claves, como ser modificable, su performance, la confiabilidad, portabilidad, etc. La arquitectura en este punto tiene mucha más importancia que el diseño o la codificación.

¿Cómo puede evaluarse la arquitectura propuesta para estos atributos? pueden usarse métodos formales para los procesos relacionados a las redes o el análisis de distintos compontes del sistema, también aquí la experiencia juega un rol clave.

4.6.1. El método ATAM

El método de análisis ATAM (Architecture Tradeoff Analysis Method), analiza las propiedas y las concesiones entre ellas.

Pasos principales:

1. Recolectar Escenarios
   - Los escenarios describen las interacciones del sistema.
   - Elegir los escenarios de interés para el análisis.
   - Incluir escenarios excepcionales sólo si son importantes.
2. Recolectar Requerimientos y/o restricciones
   - Definir lo que se espera del sistema en tales escenarios.
   - Deben especificar los niveles deseados para los atributos de interés (preferiblemente cuantificados).
3. Describir las vistas arquitectónicas
   - Las vistas del sistema que serán evaluadas son recolectadas.
   - Distintas vistas pueden ser necesarias para distintos análisis.
4. Análisis específico de cada atributo
   - Se analizan las vistas bajo distintos escenarios separadamente para cada atributo de interés distinto.
   - Esto determina los niveles que la arquitectura puede proveer a cada atributo.
   - Se comparan con los requeridos.
   - Esto forma la base para la elección entre una arquitectura u otra de modificación de la arquitectura propuesta.
   - Puede utilizar cualquier técnica o modelo.
5. Identificar puntos sensibles y de compromisos
   - Análisis de sensibilidad: ¿Cuál es el impacto que tiene un elemento sobre un atributo de calidad?
     - Los elementos de mayor impacto son los puntos de sensibilidad.
   - Análisis de compromiso
     - Los puntos de compromiso son los elementos que son puntos de sensibilidad para varios atributos.

4.7. Resumen

- La arquitectura de un sistema provee una visión a muy alto nivel del sistema en término de las partes del sistema y cómo están relacionadas para formar en conjunto el sistema.
- Dependiendo de cómo se divide el sistema, obtenemos distinas vistas de la arquitectura del sistema. En consecuencia, la arquitectura de un sistema de software está definida como estructura del sistema que abarcan elementos del software, sus propiedades externas visibles, las relaciones entre ellos.
- La arquitectura facilita el desarrollo de un sistema de alta calidad. También permite el análisis de muchas propiedades del sistema como la performance que depende en su mayoría de la arquitectura elegida tempranamente en el ciclo de la construcción del sistema.
- Hay principalmente 3 vistas de la arquitectura: módulos, componentes y conectores y la asignación de recursos.
  - En la vista de módulos el sistema es visto como una estructura de módulos como paquetes, clases, funciones, etc.
  - En la vista de componentes y conectores, el sistema es una colección de entidades llamadas componentes y estas entidades interactúan mediante conectores.
  - Una vista de Asignación de recursos describe cómo las distintas unidades del Software se sitúan a recuros de hardware en el sistema.
- La vista de C&C es la más común y es generalmente el núcleo de la descripción de la arquitectura. Esta vista es frecuentemente descripta por diagramas de bloques especificando los diferentes componentes y conectores entre los componentes.
- Hay algunos estilos comunes de la vista de C&C:
  - Conductos y Filtros
  - Cliente-Servidor
  - Compartir Datos
- La arquitectura forma las bases para el sistema y el resto de las actividades de allí la necesidad de generar un apropiado documento. Un apropiado documento debiera describir el contexto en el que la arquitectura se diseña, las distintas vistas creadas de arquitecturas y cómo estas datos se relacionan con otros. La descripción de la arquitectura debe especificar los distintos tipos de elementos, el comportamiento externo y los fundamentos de la arquitectura.
- La arquitectura debiera ser evaluada con miras a satisfacer los requisitos. Un enfoque común es hacer una evaluación subjetiva con respecto a propiedades deseadas.

Ejercicios

1. ¿Por qué la arquitectura no es sólo una estrucura que consiste de partes y sus relaciones?

2. ¿Cuáles son los diferentes estilos arquitectónicos para el estilo de C&C?

3. Considere un Web interactiva que provee features para la realización de diferentes tareas. Muestre como la arquitectura puede ser presentada con estilos de Shared Data o de Cliente Servidor. ¿Cuál Preferiría y Por qué?

4. ¿Qué debería contener un documento de la arquitectura?

5. Sugerir cómo evaluaría la arquitectura propuesta con miras a la modificabilidad.

5. El planeamiento del proyecto de software

El planeamiento es la actividad más importante del proyecto de Managment. Tiene dos objetivos básicos: establecer un costo razonable, un calendario y objetivos de calidad para el proyecto, y la realización de un plan para alcanzar los objetivos del proyecto. Un proyecto se cumple si se alcanzan los objetivos de costo, planificación y calidad. Si no tenemos definido los objetivos del proyecto no va a ser posible decidir si los hemos cumplido satisfactoriamente, y sin un plan detallado no es posible un real monitoreo o control del proyecto. A veces los proyectos se precipitan hacia la implementación sin gastar ningún esfuerzo en el planeamiento. No hacer un apropiado planeamiento para un proyecto de software grande es sacar un ticket al fracaso. Por esta razón, tratamos el planeamiento del proyecto como un capítulo independiente. Note que también veremos la fase de monitoreo como parte del planeamiento, y como el proyecto debe ser monitoreado también forma parte de la fase de planificación.

Las entradas de la actividad de planeamiento son la SRS y quizás la descripción de la arquitectura. Una SRS totalmente detallada no es esencial para la planificación, pero para un buen plan se deben conocer todos los requisitos importantes, y a su vez sería óptimo que las decisiones claves de la arquitectura hayan sido tomadas.

En general hay 2 outputs definidos de la actividad de planeamiento "**the overall project Managment Plan document**" que establece los objetivos del proyecto referidos al costo, calendario, fronteras de calidad y define el plan para "Managing Risk", el monitoreo del proyecto, etc; y el plan detallado, a veces denominado "**detailed project schedule**" que especifica las tareas necesarias a realizar para cumplir los objetivos, los recursos que lo realizarán y su planificación. El plan estructural guía el desarrollo del plan detallado, que es la principal guía durante la ejecución del proyecto para el monitoreo del proyecto.

En este capítulo discutiremos:

- Cómo estimar el esfuerzo y calendario para establecer los objetivos del proyecto y milestone y determinar el tamaño del equipo necesario para ejecutar el proyecto.
- Cómo establecer objetivos de calidad para el proyecto y preparar un plan de calidad.
- Cómo identificar los riesgos de alta prioridad que tratan con el cumplimiento de proyecto y un plan para mitigarlos.
- Cómo planear el monitoreo de un proyecto usando medidas para chequear si el proyecto está progresando de acuerdo a lo planeado.
- Cómo desarrollar detalladas tareas de manejo de las estimaciones generales y otras tareas de planeamiento hechas para eso, que si las seguimos, llegaremos a cumplir los objetivos generales de un proyecto.

5.1. Estimación del esfuerzo

Para el desarrollo del proyecto de un software, estimaciones sobre el esfuerzo general y la agenda son prerequisitos esenciales para el planeamiento del proyecto. Estas estimaciones son necesarias antes de que el desarrollo comience ya que establecen los objetivos de costo y calendario de un proyecto. Sin esas cuestiones simples como ¿Está el proyecto atrasado? ¿Excedimos los costos? y ¿Cuándo podría terminarse el proyecto? no pueden ser respondidas. Otras cuestiones más prácticas del uso de estas estimaciones están en la construcción del software, donde debemos entregale al cliente posibles costos y calendarios del proyecto para realizar un contrato. (El problema fundamentalmente está en que lo que intentamos establecer es una medida de esfuerzo humano para la realización de un producto). Estimaciones de esfuerzo y calendario también son usadas en determinadas fases para establecer el nivel del staff, para un plan detallado y para el proyecto de monitoreo.

La habilidad para estimar fácilmente el esfuerzo requerido depende del nivel de información disponible acerca del proyecto. Mientras más detallada sea la información más exacta puede ser la estimación. Por supuesto, incluso si tenemos toda la información disponible la efectividad de la estimación dependerá de los modelos o procedimientos usados y del proceso. Vamos a discutir aquí dos enfoques comúnmente usados.

5.1.1. Enfoques de estimación Top-Down

Aunque el esfuerzo para un proyecto está en función de muchos parámetros, hay un acuerdo general en que el actor primario que controla el esfuerzo es el **tamaño del proyecto**. Es decir, mientras más grande sea el proyecto, requerirá más esfuerzo. El enfoque Top-Down tiene en cuenta esto y considerará al esfuerzo como una función del tamaño del proyecto. Note que para este enfoque, primero debemos determinar la naturaleza de la función y luego para aplicarlos necesitaremos estimar el tamaño del proyecto para el cual estimaremos el esfuerzo.

Si sabemos la productividad para proyectos similares, entonces puede ser usada esa función para determinar el esfuerzo del tamaño. Si la productividad es P ($\frac{KLOC}{PM}$), entonces la estimación del esfuerzo para el proyecto será $\frac{size}{P}$ persona-mes. note que la productividad propiamente dicha depende del tamaño del proyecto (proyectos grandes a veces tienen una menor productividad), este enfoque puede funcionar sólo si el tamaño y el tipo de proyecto es similar al del conjunto del cual obtuvimos esa productividad **P**.

Una función más general comúnmente usada para determinar el esfuerzo desde el tamaño es:

$$\text{Esfuerzo} = a * \text{Tamaño}^b$$

Donde a y b son constantes, y el tamaño del proyecto está en KLOC (el tamaño también puede estar medido en los llamados "puntos de función" que pueden determinarse de los requisitos). El valor para estas constantes para una organización, se determina en base a un análisis regresivo, basado en trabajos

realizados en el pasado. En el modelo constructivo (CoCoMo) la ecuación es:

$$E = 3{,}9 * (\text{Tamaño})^{0{,}91}$$

Aunque el tamaño es el principal factor que afecta el costo, otros factores pueden tener efectos. En el CoCoMo se estima inicialmente con esa ecuación y luego se incluyen otros factores para obtener una estimación final. Para esto CoCoMo usa 15 atributos de un proyecto llamados **cost drivers**. Cada atributo tiene su escala y un ranking, según esto nos da un factor que luego multiplicamos por la estimación inicial.

Cost Drivers	Rating				
	Very Low	Low	Nominal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

Figura 37: Multiplicadores de esfuerzo para distintos Cost Drivers

Los 15 factores se multiplican para obtener un **ajuste de factores de esfuerzo** (EAF). El esfuerzo final estimado, es E multiplicado por EAF.

Ejemplo: un sistema de subastas ya mencionados en unidades anteriores y una estimación de tamaño de las distintas partes es:

Login	200 LOC
Payment	200 LOC
Administration Interface	600 LOC
Seller Functions	200 LOC
Buyer Funtions	500 LOC
View and Bookkeeping	300 LOC
Total	2000 LOC

La estimación total de este software es alrededor de 2KLOC. Si quisiéramos usar CoCoMo para una estimación de esfuerzo, deberíamos estimar el valor de diferentes “cost drivers”. Supongamos que esperamos que la complejidad

del sistema sea alta, que la capacidad de los programadores sea baja y que la aplicación de experiencia del equipo sea baja. El resto de los factores tienen un ranking nominal.

Para esto nos da:

$$EAF = 1,15 * 1,17 * 1,13 = 1,52$$

El esfuerzo inicial estimado para el proyecto de la ecuación nominal nos da:

$$E_i = 3,9 * 2^{0,91} = 7,3 \text{ PM}$$

Usamos el EAF para obtener la estimación del esfuerzo:

$$E = 1,52 * 7,3 = 11,1 \text{ PM}$$

En lugar de una estimación de esfuerzos del producto total puede usarse una estimación por fases. La distribución de esfuerzos sugerida por CoComo es:

Phase	Size			
	Small	Intermediate	Medium	Large
	2 KLOC	8 KLOC	32 KLOC	128 KLOC
Product design	16	16	16	16
Detailed design	26	25	24	23
Code and unit test	42	40	38	36
Integration and test	16	19	22	25

Figura 38: Distribución de esfuerzos sugerida por CoCoMo

Debería notarse que aunque tengamos una función acorde para medir la productividad, el enfoque Top-Down requiere la estimación del tamaño del software. Uno podría preguntarse por qué no se hace una estimación directa del esfuerzo en lugar de una estimación del tamaño? La respuesta es que la estimación del tamaño en general es más fácil que la del esfuerzo. Esto se debe principalmente al hecho de que el tamaño del sistema se puede estimar del tamaño de todos sus componentes. En cambio una estimación del esfuerzo no puede calcularse como la suma de esfuerzo para realizar cada componente ya que a esto se le suma el esfuerzo requerido para integrarlos en un sistema.

Claramente para que la estimación Top-Down funcione, es importante tener una buena estimación del tamaño. No se conoce un método “simple” para estimar el tamaño con precisión. A la hora de estimar el tamaño del software, la mejor manera puede ser obtener muchos detalles acerca de lo que vamos a desarrollar para estar seguros de que no varíe mucho el tamaño de los componentes. Mediante la obtención de detalles y usándolos para estimar los tamaños es más propenso a que el actual tamaño que estimamos esté cerca del tamaño final del software.

5.1.2. El Enfoque de estimación Bottom-Up

En este enfoque, primero se divide el proyecto en tareas y luego se obtiene una estimación para cada una de las tareas del proyecto. De la estimación de las distintas tareas se obtiene la estimación general. Es decir, la estimación general del proyecto deriva de la estimación de las partes. Este tipo de enfoque también es llamado **activity-based estimation**. Esencialmente, en este enfoque el tamaño y la complejidad del proyecto se captura en un conjunto de tareas que el proyecto tiene que realizar.

El enfoque Bottom-Up se presta propiamente directo a la estimación del esfuerzo, una vez que el proyecto está dividido en tareas más pequeñas, es posible estimar directamente el esfuerzo requerido por ellas, especialmente si las tareas son relativamente pequeñas. Una dificultad en este enfoque es que para obtener una estimación general, se deben enumerar todas las tareas. El riesgo de este enfoque entonces es omitir alguna actividad. También, estimando directamente el esfuerzo de algunas tareas generales, como el proyecto de Management, puede dificultarse la embergadura/lapso del proyecto.

Si se desarrolló la arquitectura del sistema a construir y si por información pasada conocemos la distribución de los esfuerzos en distintas fases, entonces el enfoque Bottom-Up no necesita listar completamente todas las tareas, y es posible un enfoque menos tedioso.

Aquí describiremos un enfoque usado en la organización comercial.

En este enfoque, los programas más grandes (o unidades o módulos) en el software a construir son los primeros a determinar. Cada unidad/programa se clasifica como: simple, medio o complejo basado en ciertos criterios. Para cada unidad de clasificación se decide un promedio de esfuerzo para la codificación y test. Este promedio puede estar basado en experiencia de proyectos similares, algunos lineamientos o experiencia de alguna persona.

Una vez que se clasificaron todas las unidades, y se determinó este promedio sabremos el esfuerzo total. Para el esfuerzo total en codificación, el esfuerzo requerido para otras fases y actividades en el proyecto es detrminado. Como un porcentaje del esfuerzo de codificación. Para esto, la información acerca de performance pasada en el proceso, como se distribuyó el esfuerzo en diferentes fases del proyecto se determina. Esta determinación luego se usa para determinar el esfuerzo de otras fases y actividades de esfuerzo para codificación. Desde estas estimaciones se obtiene la estimación del esfuerzo total para un proyecto.

El enfoque se presta a una justa mezcla de experiencia e información. Si no hay disponible información pasada (por ejemplo en un nuevo tipo de proyecto), uno puede estimar el esfuerzo de codificación usando la experiencia en codificación de cada una de las unidades especificadas. Con esta estimación, podemos obtener una estimación razonable para otras tareas trabajando con algún criterio o estándar. Esta estrategia puede fácilmente contar con actividades que son difíciles de enumerar tempranamente pero consumen esfuerzo para otras actividades o categorías.

El procedimiento para la estimación puede ser resumida en los siguientes pasos:

1. Identificar los módulos en el sistema y clasificarlos como: simple, medio o complejo.

2. Determinar un esfuerzo promedio para codificar lo simple, medio o módulos complejos.

3. Obtener el efuerzo total de codificación para los distintos tipos de módulos.

4. Usando el esfuerzo de distribución para proyectos similares, estimar el esfuerzo para otras tareas y el esfuerzo total.

5. Refinar la estimación basado en factores específicos del proyecto.

Este procedimiento usa una judiciosa mezcla de datos del pasado (en la forma de distribuir el esfuerzo) y experiencia de los programadores. Este enfoques es también simple y similar para cualquier tamaño de proyecto. Por esta razón, para pequeños proyectos, mucha gente encuentra a este enfoque natural y confortable.

Note que este método de clasificar programas en un grupo pequeño de categorías y el uso de un promedio para cada categoría sólo es para estimar el esfuerzo. En la descripción del calendario, cuando un "project Management" asigna cada unidad a un miembro del equipo para codificar y el presupuesto de tiempo para la actividad, características de la unidad se toma para asignar más o menos tiempo que el promedio.

5.2. Planificación y Recursos Humanos

Luego de establecer un objetivo en primera línea de esfuerzo, necesitamos establecer un objetivo para el horario de entrega. Con la estimación del esfuerzo (en persona-mes), puede ser tentador elegir cualquier duración del proyecto basado en nuestra conveniencia y acomodar el tamaño de equipo que deseemos para asegurar que se cumpla el esfuerzo estimado. Sin embargo, como se sabe ahora, las personas y los meses no son totalmente cambiables en un proyecto de software. Personas y meses se pueden intercambiar arbitrariamente sólo si todas las tareas del proyecto pueden hacerse en paralelo, y no es necesaria la comunicación entre las personas que realizan las tareas. Esto no se cumple en los proyectos de software, hay dependencia entre las tareas (testing sólo puede hacerse luego de la codificación), y una persona que realiza alguna tarea en un proyecto necesita comunicarse con otras que realizan otras tareas.

Sin embargo, para un proyecto con alguna estimación de esfuerzo, múltiples planificaciones pueden ser posibles. Por ejemplo, para un proyecto cuyo esfuerzo es estimado en 56 persona-mes, una entrega en 8 meses es posible con 7 personas o lo mismo una entrega en 7 meses es posible con 8 personas. En otras palabras, una vez que se fija el esfuerzo, hay cierta flexibilidad en establecer una fecha de entrega por un apropiado staff, pero esta flexibilidad no es ilimitada. Demostraciones empíricas muestran que no hay una ecuación que relaciones correctamente esfuerzo y entrega.

El objetivo es fijar una fecha razonable que se pueda cumplir (si se asignan los recursos deseados). Un método para determinar el calendario general es determinarlo en función del esfuerzo. Esta función puede determinarse de datos de proyectos completados usando técnicas estáticas como un ajuste regresivo de una curva a través de un gráfico de dispersión obtenido del gráfico entre los esfuerzos y entregas de trabajos anteriores. Esta curva generalmente no es lineal porque el horario de entrega no crece linealmente con el esfuerzo. Existen diversos modelos que siguen este enfoque. Una fórmula de la IBM establece que M meses toman:

$$M = 4,1 * E^{0,36}$$

En CoCoMo, la ecuación es:

$$M = 2,5 * E^{0,38}$$

Como la fijación del horario no es una función que dependa solamente del esfuerzo, la determinación de la fecha de esta manera esencialmente es una guía.

Otro método para establecerlo es el método llamado la **raíz cuadrada**, que sugiere que la fecha o tiempo puede rondar la raíz cuadrada del esfuerzo en persona mes. Por ejemplo, si el esfuerzo de un proyecto es 50 Persona-Mes, la fecha de entrega puede acomodarse entre 7 a 8 meses.

Con este macro podemos determinar fácilmente el tiempo de los principales **milestone** del proyecto. Suele ocurrir en los procesos que al inicio y al final pocas personas son necesarias.

Como no es tan común "mudar" gente entre etapas en pequeños proyectos se asigna un grupo fijo de personas, que en las primeras y últimas fases estarán "desocupadas" por lo que se encargan de profundizar en algún entrenamiento o documentación. Generalmente, el diseño requiere el 25 % del tiempo, la construcción el 50 %, la integración del sistema y testing el otro 25 %. CoCoMo dice: 19 % diseño, 62 % programación y 18 % para la integración.

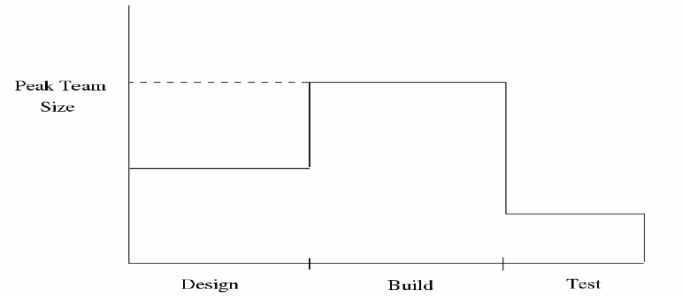

Figura 39: Distribución de mano de obra en un proyecto típico

5.3. Planeamiento de Control de calidad

Teniendo un conjunto de objetivos para el esfuerzo y entrega, es necesario definir el objetivo para la tercer dimensión clave de un proyecto: **la calidad**. Sin embargo, no hay una forma de establecer fácilmente objetivos relacionados a la calidad. Para la calidad, incluso si establecemos el objetivo en términos del producto esperado en términos de la densidad de errores, no es fácil hacer un plan para ese objetivo ni tampoco establecer si se cumplió el objetivo. Por lo tanto, a veces, los objetivos de calidad se especifican en términos de criterios aceptables, el producto final debiera trabajar en todos los casos establecidos y los tests de casos son aceptables. Además, puede establecerse un número de errores aceptables en la fase de test. Por ejemplo, "si no hay más de n errores descubiertos en el testing se aceptará"

El plan de calidad es un conjunto de calidades-Relacionadas a actividades que el proyecto planea hacer para alcanzar el objetivo de calidad. Para planificar la calidad, primero entenderemos la "**Defect injection**" y "**removal cycle**", como aquellos defectos que determinan la calidad del producto final.

El desarrollo del software está altamente orientado a personas y por esto es propenso a errores. El desarrollo comienza sin errores, los defectos se inyectan en el software cuando comienzan las fases de construcción. Es decir, durante la transformación de las necesidades del usuario a la satisfacción de esas necesidades se inyectan errores en las tareas realizadas. Estas inyecciones en etapas están primariamente, SRS, diseño de alto nivel, diseño detallado y codificación. Para asegurar la entrega de un software de alta calidad, estos errores se remueven durante actividades de control de calidad (QC). Las actividades de QC para la remoción de defectos incluyen, revisión de los requerimientos, revisiones de diseño y de código, testing.

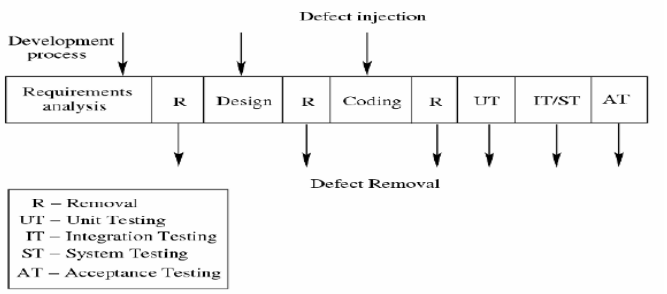

Figura 40: Detección de errores y ciclo de remoción

Como el objetivo final es entregar un software con baja densidad de errores, asegurar la calidad se resuelve alrededor de dos temas principales: reducir la inyección de errores, incrementar la remoción de errores. La primera se realiza

en general siguiendo estándares, metodologías, que nos ayudan a prevenir las chances de inyección de errores. El plan de calidad por esta razón se enfoca mayormente en un adecuado planeamiento de control de calidad con tareas para la remoción de defectos.

Revisiones y testing son dos de las actividades más comunes del QC en un proyecto. Mientras que las revisiones son estructuradas, orientadas a los procesos humanos, testing es el proceso de ejecutar software con el objetivo de identificar defectos.

El enfoque más común para el plan de calidad de un proyecto es especificar las actividades de QC a ser realizadas en el proyecto, y tener lineamientos adecuados para realizar cada QC actividad, con los cuales la probabilidad de cumplir los objetivos de calidad sean altos. Durante la ejecución del proyecto, estas actividades se llevan a cabo de acuerdo con los procesos definidos.

Cuando este enfoque se usa para asegurar la calidad, cuantificar reclamos puede ser muy difícil de hacer. Para la valoración cuantitativa de los procesos de calidad son necesarias métricas basadas en el análisis. Por lo tanto para asegurar calidad lo mejor es aplicar técnicas adecuadas de QC en los lugares correctos del proceso y usar la experiencia para asegurar que suficientes tareas del QC se hicieron en el proyecto.

Por lo tanto el plan de calidad es una gran especificación de qué tareas de QC se realizarán y cuándo, y qué procesos y lineamientos se usarán para la realización de tareas de QC. La elección depende de la naturaleza y los objetivos y restricciones del proyecto. Típicamente, las tareas de QC serán tareas planificadas en una detallada planificación de un proyecto (fecha). Por ejemplo, se especificará qué documentos serán revizados, qué partes de código se inspeccionarán y qué niveles de testing se realizarán. El plan puede mejorar considerablemente si algunas especificaciones de algunos niveles de defectos se esperan encontrar para diferentes controles de calidad, esto puede ayudar al monitoreo de la calidad como proceda el proyecto.

5.4. Planificacion del Manejo de Riesgos

Un proyecto de software es un compromiso complejo. Eventos imprevistos pueden tener un impacto adverso en la capacidad del proyecto de cumplir con los objetivos de costo, agenda o de calidad. El manejo de riesgos es un intento de minimizar las posibilidades de fracaso causadas por eventos imprevistos. El objetivo del manejo de riesgos no es abandonar proyectos que tengan algún tipo de riesgos sino minimizar el impacto frente a sucesos imprevistos en los compromisos que vamos a tomar.

Un riesgo es un evento probabilístico, puede o no ocurrir. Por esta razón, tenemos una visión simplista de no querer ver los riesgos que pueden ocurrir.

5.4.1. Conceptos de la administración de riesgos

Un **riesgo** se define como la exposición a la posibilidad de daño o pérdida. Es decir, el riesgo implica que hay una posibilidad de que algo malo pueda pasar.

En el contexto de proyectos de software, algo negativo implica un efecto adverso en cuanto a los costos o agenda. El Managment de riesgos es el área que intenta asegurar que el impacto frente a un riesgo en el costo, calidad y agenda sea mínimo.

Riesgos son eventos no regulares, es decir la salida de algunos contratados de la empresa no se toma como un riesgo por lo que se trata por el proceso de Managment.

Debería quedar claro que la administración de riesgos tiene que tratar con la identificación de eventos indeseables que pueden ocurrir, la probabilidad de que ocurran y las pérdidas provocadas frente a esos eventos. Una vez que son conocidos, podemos formular estrategias para que además de reducir la probabilidad de que los riesgos se materialicen intentan reducir el efecto si se producen. Por lo tanto el managment de riesgos resuelve temas alrededor de la **evaluación** y **control** de riesgos.

5.4.2. Evaluación de riesgos

1. Identificar los riesgos: para un proyecto particular, que generalmente consiste de la revisión de listas prefijadas de riesgos que incluyen:
   - Personal Incapacitado.
   - Plan Irreal.
   - Desarrollo erróneo de funciones.
   - Desarrollar una errónea interfaz de usuario.
   - Gold Plating: agregar items al proyecto que consuman recursos sin un feedback.
2. Análisis y Establecer Prioridades: en esta etapa se estiman las probabilidades de que ocurran los riesgos y las pérdidas asociadas al mismo.

   Exposición a Riesgo: RE.

   Luego tomamos: $RE = Probabilidad(Riesgo) * \text{Pérdida}(Riesgo)$

   Es decir RE, está asocidada a un sólo riesgo, luego se calcula el RE_i para todos los riesgos identificados y se establecen prioridades según el orden por su RE_i asociado.

5.4.3. Control de riesgos

El principal objetivo del managment de riesgos es identificar los principales riesgos y enfocarse en ellos. Una vez que el proyecto de managment identificó y priorizó los riesgos, es fácil identificar los principales. La pregunta que sigue es que hago con estos. Es decir conocer los riesgos sólo tiene valor si es posible prepararte a ellos.

Para la mayoría de los riesgos la estrategia es realizar acciones para intentar reducir la probabilidad de que se materialicen. Es decir el control no es sólo una tarea intelectual.

El monitoreo de los riesgos es la actividad de monitorear el estado de varios riesgos y sus actividades de control. Un enfoque simple para el monitoreo de riesgos es analizar los riesgos en cada principal milestones y cambiar el plan si es necesario.

5.5. Planificación del seguimiento del Proyecto

El plan de proyecto de managment es simplemente un documento que puede ser usado para guiar la ejecución de un proyecto. Incluso un buen plan es inútil a menos que se ejecute apropiadamente. Y la ejecución no puede ser conducida apropiadamente a menos que se monitoree adecuadamente y se realicen tareas alrededos del plan.

El monitoreo requiere medidas para validar la situación del proyecto. Si las medidas se toman durante la ejecución del proyecto debemos tener en cuenta qué estamos midiendo, cómo lo estamos midiendo y cuándo.

5.5.1. Métricas

El propósito básico de las métricas en un proyecto es proveer datos al proyecto de Managment acerca del estado actual del proyecto. Cómo los objetivos del proyecto se establecen en medida de software a entregar, el costo, la agenda y calidad para monitorear el estado de un proyecto, el tamaño, esfuerzo, agenda y defectos son las principales medidas que necesitamos.

Para un efectivo monitoreo, un proyecto debe planearse para colectar estas medidas. La mayoría de las empresas proveen herramientas y políticas para registrar los datos básicos que están disponibles por los managers para el rastreo de errores.

5.5.2. Seguimiento del Proyecto

De las actividades discutidas hasta ahora resultará el documento del proyecto de managment que establece los objetivos del proyecto en cuanto al esfuerzo, planificación y calidad y el proyecto de monitoreo. Ahora este plan debe traducirse un plan detallado a ser seguido en el proyecto.

5.6. Resumen

- El proyecto de planeamiento tiene dos objetivos: establecer los objetivos generales o expectativas para el proyecto y la estructura general para el managment del proyecto y las tareas de planificación a realizarse para que se cumplan los objetivos. El proyecto de planificación de calidad y el plan managment de riesgos se establecen para intentar alcanzar la calidad y no caer frente a posibles riesgos.

- En el enfoque Top-Down para estimación de esfuerzo del proyecto, el esfuerzo se estima desde el tamaño estimado que puede ser refinado usando

otras características del proyecto. El esfuerzo para diferentes fases se determina desde el esfuerzo general usando la distribución de datos y esfuerzos. COCOMO es un modelo que usa este enfoque.

El enfoque Bottom-Up para la estimación, primero se identifican los módulos a construirse, y luego se usa un esfuerzo promedio para codificar estos módulos para determinar el esfuerzo general de codificación. Desde la estimación del esfuerzo, el esfuerzo para otras fases y en general del proyecto se estima.

- Un proyecto puede tener riesgos, y cumplir los objetivos bajo la presencia de riesgos requiere un adecuado Managment de riesgos. Los riesgos son aquellos eventos que pueden o no ocurrir. Estos pueden priorizarse basados en las expectativas de pérdidas de ese riesgo que es una combinación de la probabilidad de que ocurra por las pérdidas causadas si se producen.
- El progreso del proyecto necesita ser monitoreado usando adecuadas métricas entonces se pueden aplicar acciones correctivas cuando son necesarias. Estas métricas comúnmente se usan para la agenda actual, esfuerzo consumido, defectos encontrados y el tamaño del producto.
- Para la ejecución, la estructura general se divide en una detallada planificación de tareas a realizarse para cumplir los objetivos y las restricciones. Estas tareas se asignan a un grupo específico de trabajo.

Ejercicios

1. ¿Cuál es el rol de la estimación del esfuerzo de un proyecto, y por qué es importante hacer esta estimación tempranamente?
2. Si la arquitectura propuesta en el sistema ha establecido los principales componentes en el sistema, y disponés de código similar que realiza las mismas funciones, ¿qué método usarías para la estimación?
3. Suponga que una organización planea usar COCOMO para la estimación del esfuerzo, pero sólo busca utilizar 3 costs Drivers, la complejidad del producto, la capacidad de los programadores y el desarrollo de la agenda. En esta situación, para una estimación inicial, en ¿cuánto puede variar la estimación final?
4. ¿Por qué no es factible realizar todas las combinaciones posibles para lograr la estimación de esfuerzo variando la gente y los meses?
5. Un cliente te consulta para completar un proyecto, cuyo esfuerzo estimado es E, en un tiempo T. ¿Cómo decidirías mejor para aceptar si esta agenda es aceptable o no?
6. Para un proyecto de grupo de estudiantes durante un semestre, liste cuáles son los principales riesgos que el proyecto puede tener, y las estrategias posibles para mitigar estos problemas.

7. Suponga que realiza una planificación detallada para tu proyecto cuyo esfuerzo y planificación estimada para varias milestone se realizaron en un plan de proyecto. ¿Cómo revisaría si la agenda es consistente con el plan general? ¿Qué haría si no lo es?

6. Diseño

La actividad de diseño comienza cuando el documento de requisitos para el software que vamos a desarrollar está disponible y ya se haya diseñado la arquitectura. Durante el diseño a su vez vamos a refinar la arquitectura. Generalmente, el diseño se enfoca en lo que hemos llamado durante la arquitectura: la vista de módulos. Es decir, durante el diseño vamos a determinar qué módulos el sistema debe tener y tienen que desarrollarse. A veces, la vista de módulos puede ser posiblemente la estructura de los módulos de cada componente en la arquitectura. En ese caso es simple mapear los componentes y módulos pero no es siempre así. En ese caso debemos asegurar que la vista de módulos creada en el diseño sea consistente con la arquitectura.

El diseño del sistema es esencialmente el proyecto original o un plan para la solución del sistema. Es decir, **el diseño es esencialmente una actividad creativa** Aquí vamos a considerar que un sistema es un conjunto de módulos que claramente definen el comportamiento del mismo.

El proceso de diseño de software suele tener dos niveles: en el primer nivel nos enfocamos en decidir qué módulos son necesarios para el sistema, la especificación de esos módulos y cómo los módulos deberían comunicarse, esto es lo que llamamos **diseño de alto nivel**. En el segundo nivel, el diseño interno de los módulos se decide cómo la especificación de los módulos puede cumplirse. Este nivel a veces suele llamarse **diseño detallado** o diseño lógico. El diseño detallado esencialmente expande el diseño del sistema hasta contener los detalles necesarios para realizar la codificación.

Una metodología de diseño es un enfoque sistematizado para crear un diseño mediante el uso de un conjunto de técnicas.

6.1. Conceptos de Diseño

El diseño del sistema es **correcto** si el sistema construido de acuerdo al diseño satisface todos los requisitos del sistema. Claramente, el objetivo durante la fase de diseño es producir un diseño correcto. Sin embargo, correctitud no es el único criterio durante la fase de diseño.

El objetivo es encontrar el mejor diseño posible. Se deberán explorar diversos diseños alternartivos. Los criterios de evaluación son usualmente subjetivos y no cuantificables.

Principales criterios para evaluar un diseño:

- Correción: es fundamental pero no es el único criterio.
  - ¿El diseño implementa requerimientos?
  - ¿Es factible el diseño dada las restricciones?
- Eficiencia:
  - Le compete al uso apropiado de los recursos del sistema (principalmente CPU y memoria).

    - Debido al abaratamiento del hardware toma un segundo plano.
    - Sólo cobra importancia en sistemas integrados o de tiempo real.
- Simplicidad:
    - Tiene impacto directo en el mantenimiento.
    - El mantenimiento es caro.
    - Un diseño simple facilita la comprensión del sistema ⇒ hace el software mantenible.
    - Facilita el testing.
    - Facilita el descubrimiento y corrección de bugs.
    - Facilita la modificación del código.

Eficiencia y simplicidad no son independientes por lo que el diseñador debe lograr un equilibrio.

El diseño es un proceso creativo. **No** existe una serie de pasos que permitan derivar el diseño de los requerimientos. Sólo hay principios que guían el proceso de diseño.

6.1.1. Principios Fundamentales

Los siguientes conceptos forman la base de la mayoría de las metodologías de diseño.

- Partición y Jerarquía.
- Abstracción
- Modularidad

Partición y Jerarquía

- Principio básico: "divide y conquistarás"
- Dividir el problema en pequeñas partes que sean manejables
    - Cada parte debe poder solucionarse separadamente.
    - Cada parte debe poder modificarse separadamente.

Las partes no son totalmente independientes entre sí: deben comunicarse/-cooperar para solucionar el problema mayor. La comunicación agrega complejidad. A medida que la cantidad de componentes aumenta, el costo del particionado (incluyendo la complejidad dela comunicación) también aumenta. Detener el particionado cuando el costo supera al beneficio.

Tratar de mantener la mayor independencia posible entre las distintas partes ⇒ simplifica el diseño y facilita mantenimiento. El particionado del problema determina una jerarquía de componentes en el diseño. Usualmente la jerarquía se forma a partir de la relación "es parte de".

Abstracción

Esencial en el particionado del problema. La abstracción de una componente describe el comportamiento externo sin dar detalles internos de cómo se produce este comportamiento.

- Abstracción de componentes existentes:
  - Representa a las componentes como cajas negras.
  - Oculta detalle, provee comportamiento externo.
  - Útil para comprender sistemas existentes $\Rightarrow$ tiene un rol importante en mantenimiento.
  - Útil para determinar el diseño del sistema existente.
- Abstracción durante el proceso de diseño:
  - Las componentes no existen.
  - Para decidir como interactúan las componentes sólo el comportamiento externo es relevante.
  - Permite concentrarse en una componente a la vez.
  - Permite considerar una componente sin preocuparse por las otras.
  - Permite que el diseñador controle la complejidad.
  - Permite una transición gradual de lo más abstracto a lo más concreto.

Dos mecanismos comunes de abstracción:

- Abstracción funcional:

  Especifica el comportamiento funcional de un módulo. Los módulos se tratan como funciones de entrada/salida. La mayoría de los lenguajes proveen características para soportarla. Ej: procedimientos, funciones. Un módulo funcional puede especificarse usando precondiciones y postcondiciones. Forma la base de las metodologás orientadas a funciones.

- Abstracción de datos.

  Una entidad del mundo real provee servicios al entorno. Es el mismo caso para las entidades de datos: se esperan ciertas operaciones de un objeto de datos. Los detalles internos no son relevantes. La abstracción de datos provee esta visión: Los datos se tratan como objetos junto a sus operaciones.

  Las operaciones definidas para un objeto sólo pueden realizarse sobre este objeto. Desde fuera, los detalles internos del objetos permanecen ocultos, sus operaciones son visibles. Los lenguajes modernos soportan abstracción de datos. Forma la base de las metodologías orientadas a objetos.

Modularidad

Un sistema se dice modular si consiste de componentes discretas tal que puedan implementarse separadamente y un cambio a una de ellas tenga mínimo impacto sobre las otras.

Provee la abstracción en el software. Es el soporte de la estructura jerárquica de los programas. Mejora la claridad del diseño y facilita la implementación. Reduce los costos de testing, debugging y mantenimiento.

- No se obtiene simplemente recortando el programa en módulos.
- Necesita criterios de descomposición: resulta de la conjunción de la abstracción y el particionado.

Un sistema es una jerarquía de componentes. Dos enfoques para diseñar tal jerarquía: Estrategias top-down y bottom-up

- Top-down: comienza en la componente de más alto nivel (la más abstracta); prosigue construyendo las componentes de niveles más bajos descendiendo en la jerarquía.

  El diseño comienza con la especificación del sistema.

  - Refinamiento Paso a Paso: Define el módulo que implementará la especificación. Especifica los módulos subordinados. Luego, iterativamente, trata cada uno de estos módulos especificados como el nuevo problema. El refinamiento procede hasta alcanzar un nivel donde el diseño pueda ser implementado directamente.
  - Ventajas: En cada paso existe una clara imagen del diseño. Enfoque más natural para manipular problemas complejos.
  - Contras: La mayoría de las metodologías de diseño se basan en este enfoque.

- Bottom-up: comienza por las componentes de más bajo nivel en la jerarquía (las más simples); prosigue hacia los niveles más altos hasta construir la componente más alta.

  El diseño comienza desde los módulos más básicos o primitivos.

  - Proceso por capas de abstracción: Utilizando estos módulos se construyen operaciones que proveen una capa de abstracción. Y luego estas operaciones se usan para construir operaciones más poderosas en una capa de abstracción superior (y así sucesivamente).
  - Ventaja: Necesario si existen módulos disponibles para reusar.
  - Contra: Es necesario tener una idea de las abstracciones superiores.

  Top-down o bottom-up puros no son prácticos. En general se utiliza una combinación de ambos.

Un sistema es considerado **modular** si consiste de módulos discretos donde cada módulo puede implementarse por separado y un cambio en un módulo tiene un mínimo impacto en otros módulos.

La modularidad es claramente una propiedad deseable, ayuda al debugging del sistema.

Acoplamiento y cohesión son dos criterios de modularización, a su vez hablaremos del principio abierto-cerrado que es otro criterio de modularidad.

6.1.2. Acoplamiento

Dos módulos son considerados independientes si uno puede funcionar completamente sin la presencia del otro. Obviamente si dos módulos son independientes, pueden solucionarse y modificarse por separado. Sin embargo, todos los módulos del sistema no pueden ser independientes unos de otros, estos deben interactuar para lograr el comportamiento requerido del sistema. La noción de acoplamiento intenta capturar el concepto de "cuán fuertemente" diferentes módulos están interconectados.

Alto acoplamiento entre módulos va de la mano con fuertes conexiones entre módulos mientras que un bajo acoplamiento entre módulos tiene conexiones débiles.

La elección de módulos decide el acoplamiento entre módulos.

El acoplamiento incrementa con la complejidad y "oscuridad" y la interfaz entre módulos. Para mantener un acoplamiento bajo vamos a querer minimizar el número de interfaces por módulos y la complejidad de cada interfaz. Una interfaz de un módulo se utiliza para pasar información a otros módulos.

El acoplamiento incrementa también por interfaces "oscuras" tales como compartir variables entre módulos.

A su vez la complejidad de la interfaz de un módulo incrementa el acoplamiento, por ejemplo la cantidad de parámetros pasados en una interfaz debiera intentar minimizarse.

El tipo de flujo de datos es el tercer factor que afecta el acoplamiento ya que no quedan en claro las abstracciones que se están manejando.

El costo de estos tres factores está resumido en la siguiente tabla

	Complejidad de la interfaz	Tipo de conexión	Tipo de Comunicación
Bajo	Simple Obvio	Con un Módulo por el nombre	Dato
Alto	Complicado oscura	Con un Módulo con elementos internos	Control Híbrido

En diseños orientados a objetos existen 3 tipos de acoplamiento:

- Acoplamiento de Interacción.
- Acoplamiento de Componentes.
- Acoplamiento de Herencia.

Acoplamiento de Interacción

Ocurre debido a que métodos de una clase invocan métodos de otra clase, para mantener bajo el acoplamiento sólo debemos pasar datos y no objetos cuando invocamos métodos de otra clase.

Acoplamiento de Componentes

Un objeto A de la clase C está acoplado con una clase C_1, si C tiene una variable de tipo C_1 o tiene un método que toma un parámetro de C_1.

Acoplamiento de Herencia

Se genera debido a las herencias entre clases. El problema se da cuando usamos herencia múltiple y cuando queremos pisar métodos heredados. El escenario más débil se da cuando se hereda de una clase sólo para extenderla. Entonces a la hora de Heredar debemos preguntarnos si estamos agregando funcionamiento.

6.1.3. Cohesión

Hemos visto que el acoplamiento es reducido cuando se minimizan las relaciones entre elementos de distintos módulos. Es decir, el acoplamiento es bajo cuando elementos de distintos módulos tienen una pequeña o ninguna relación entre ellos. Por otra parte, otra manera de cumplir esto es incrementar las relaciones entre elementos del mismo módulo. La cohesión es el concepto que intenta capturar estas relaciones intramodular. Con la cohesión, estamos interesadas en determinar que tan **closely** los elementos de un módulo están relacionados con otros.

La cohesión de un módulo representa cuán ajustadas son las relaciones entre elementos del módulo. La cohesión de un módulo le da al diseñador una idea acerca de cómo los elementos de un módulo forman juntos el mismo. Cohesión y acoplamiento están claramente relacionados. Usualmente, mientras más cohesión haya entre módulos del sistema, menor será el acoplamiento entre módulos. Esta relación no es perfecta, pero se observa en la práctiva. A su vez hay distintos niveles de cohesión:

1. Casual
2. Lógico
3. Temporal
4. Procedimental
5. Comunicacional
6. Secuencial
7. Funcional

Casual: es el nivel más bajo y funcional el más alto. Cohesión casual ocurre cuando no hay ninguna relación entre elementos del módulo. Cohesión casual puede ocurrir si a un programa existente se lo divide en piezas donde cada pieza sea un módulo.

Un módulo tiene **cohesión lógica** si hay alguna relación lógica entre elementos del módulo, y los elementos realizan funciones que caen en la misma clase lógica. Un típico ejemplo de este tipo de cohesión es un módulo que se encarga de todas las salidas de cada entrada. En general, cohesión lógica debería evitarse en lo posible.

La **cohesión temporal** es similar a la lógica, excepto que los elementos también están relacionados en el tiempo de ejecución. Módulos que realizan actividades como: inicialización, limpieza y terminación son usualmente módulos relacionados temporalmente. Esto en general abandona el problema de pasar flags y usualmente hace el código más simple.

Un módulo con **cohesión procedimental** contiene elementos que pertenecen a una unidad común de procesamiento. Por ejemplo, un ciclo o una secuencia de decisiones de estados puede combinarse en distintos módulos.

Un módulo con **cohesión comunicacional** tiene elementos que están relacionados por una referencia al mismo tipo de entrada y salida de datos. Es decir, en una cohesión comunicacional, los elementos están relacionados porque operan sobre el mismo tipo de datos. Ejemplos pueden ser prints o Records. La cohesión comunicacional puede realizar mucho más de una función. Sin embargo, la cohesión comunicacional es suficientemente alta, es generalmente aceptada si estructuras alternativas con cohesión más alta no pueden identificarse fácilmente.

Cuando los elementos de un módulo están juntos porque la salida de una forma de entrada es salida de otra, tenemos **cohesión secuencial**, si tenemos una secuencia de elementos de entrada que son salida de otros, la cohesión secuencial no provee lineamientos de cómo combinarlos en módulos.

Cohesión funcional es la cohesión más fuerte. En un límite funcional entre módulos, todos los elementos de módulos están para realizar una simple función. Por función, no nos referimos sólo a funciones matemáticas también a las del tipo: ordenar un arreglo o computar la raíz cuadrada son del tipo de cohesión funcional.

¿Cómo podemos determinar el nivel de cohesión de un Módulo? No hay una fórmula matemática que pueda usarse. Sólo podemos usar nuestro propio juicio. Una técnica muy usada para determinar si un módulo tiene cohesión funcional es intentar describir el propósito del módulo en una oración. Si no podemos describirlos usando una oración simple, el módulo no tiene cohesión funcional.

Cohesión en sistemas orientados a objetos abarca tres aspectos

- Cohesión de Métodos.
- Cohesión de Clases.
- Cohesión de Herencias.

Cohesión de métodos es la misma que la cohesión en módulos funcionales se enfoca en por qué los diferentes elementos de código de un método están juntos en un método. La forma más alta de cohesión se da cuando cada método implemente una función claramente definida, y todos los estados en el método contribuyen a implementar esta función.

Cohesión de clases se enfoca en por qué diferentes atributos y métodos están en la misma clase. El objetivo es tener una clase que implemente un simple concepto de abstracción con todos sus elementos contribuyendo a implementar este concepto. En general, cuando hay múltiples conceptos encapsulados en una clase, la cohesión de clase no es tan alta como podría ser y el diseñador debe intentar cambiar el diseño para tener cada clase encapsulando un simple concepto. Un ejemplo de esta situación es cuando tenemos una clase donde el conjunto de métodos puede dividirse en más grupos, cada uno accediendo a un distinto subconjunto de atributos. Es decir, el conjunto de métodos y atributos puede dividirse en grupos separados, cada uno encapsulando distintos conceptos. Claramente en esta situación, teniendo clases separadas que encapsulan los conceptos separados, podemos tener módulos con mayor cohesión.

Cohesión en Herencia se enfoca en la razón de porque las clases están juntas en una jerarquía. Las dos principales razones para el uso de herencia es: generalizar-especializar y para reusar código. La cohesión es considerada alta si la jerarquía soporta generalización-especificación de algún concepto. Es considerada baja si la jerarquía es principalmente para compartir código con un débil concepto de relación entre la superclase y subclase. En otras palabras, es deseabe en un sistema orientado a objetos que la jerarquía de clase se use si es posible indentificar una especialización o generalización en la relación.

6.1.4. El principio Abierto-Cerrado

Este es un concepto de diseño que existe más en el contexto O.O. como la cohesión y el acoplamiento, el objetivo aquí también es promover la construcción de sistemas que sean fáciles de modificar, cómo la modificación y los cambios ocurren frecuentemente y como el diseño no puede acomodarse fácilmente al cambio resultará en la muerte del sistema y no será capaz de adaptarse al mundo.

El principio básico, establecido por Bertrand Meyer, es "las entidades del software deben ser abiertas para la extensión pero cerradas para la modificación". Un módulo "abierto para la extensión" significa que el comportamiento puede ser extendido para acomodarse a nuevas demandas propuestas en el módulo debido a cambios en los requerimientos y funcionalidad del sistema. Que el módulo sea "cerrado para modificación" significa que el código existente no se cambia para realizar mejoras.

Entonces surge la pregunta ¿Cómo es posible hacer mejorar a un módulo sin cambiar código existente? Esto principalmente restringe los cambios a módulos únicamente a extensiones, es decir, permite agregar código, pero deshabilita cambios de código existente. Si esto puede hacerse, claramente, el valor es tremendo. Cambios en el código involucra grandes riesgos y asegurar que un cambio

no rrompa elementos que nos llevará a rehacer grandes casos de testing. Este riesgo puede minimizarse si no se hacen cambios en el código existente. Pero sino se realizan cambios, ¿Cómo es posible realizar mejoras? Este principio dice que las mejoras deben realizarse agregando nuevo código sobre la alteración de código viejo.

Hay otros tipos de beneficios de esta que es que los programadores prefieren escribir código nuevo antes que leer código viejo inclusive aquel escritos por ellos mismos.

Este principio puede satisfacerse en diseños orientados a objetos apropiadamente usando herencia y polimorfismo. Consideremos el siguiente ejemplo de un programa para un cliente que realiza impresiones.

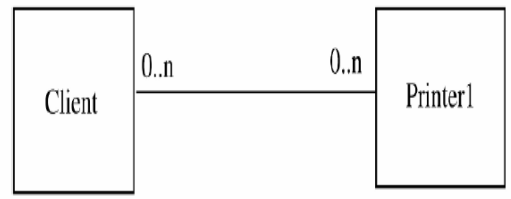

Figura 41: Ejemplo Inicial sin usar Subtipos ni Herencia

La solución será crear una interfaz de impresora con un método "imprimir" general y según cada impresora cada uno puede extender o implementar esta interfaz por lo que agregar y sacar distintas impresora sólo se limita a agregar módulos que respeten esta interfaz.

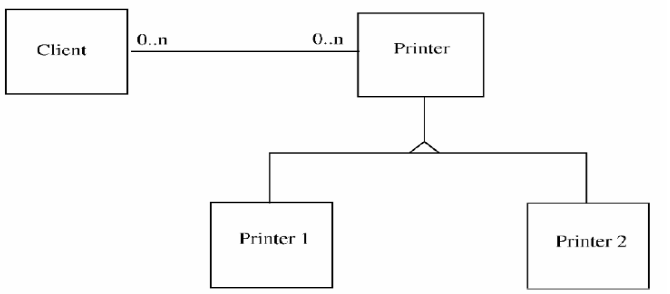

Figura 42: Ejemplo usando Herencia o Interfaz

6.2. Diseño orientado a Funciones

Crear el diseño del sistema de software es la mayor preocupación de la fase de diseño. Se han propuesto muchas técnicas de diseño durante años para proveer alguna disciplina para manejar la complejidad del diseño de grandes sistemas. El objetivo de las metodologías de diseño no es reducir el proceso de diseño a una secuencia mecánica de pasos pero sí proveer lineamientos para ayudar al diseñador durante la fase de diseño. Discutiremos una metodología para el desarollo del diseño de sistema orientado a funciones. La metodología emplea diagramas para la creación de diseño por lo tanto describiremos esta notación.

6.2.1. Diagramas estructurales

La estructura del programa está compuesta de módulos junto a sus interconexiones. Cada programa tiene una estructura y dado el programa esta estructura puede determinarse. En estos gráficos, un módulo se representa con una caja con el nombre del módulo inscripto. Una flecha desde el módulo A al B significa que el módulo A llama al módulo B. A es superordinado de B y B se llama subordinado de A. La flecha tiene una etiqueta con los parámetros.

El parámetro puede ser:

- Un dato: una flecha con una circunferencia en la cóla.
- Información de control: una flecha con un círculo en la cola.

```
1 Main()
2 {
3       int sum, n , N, a[MAX];
4       Readnums(a,&N);
5       Sort(a,N);
6       SConf(&N);
7       sum = add_n(a,n);
8       Printf("%d",sum);
9 }
```

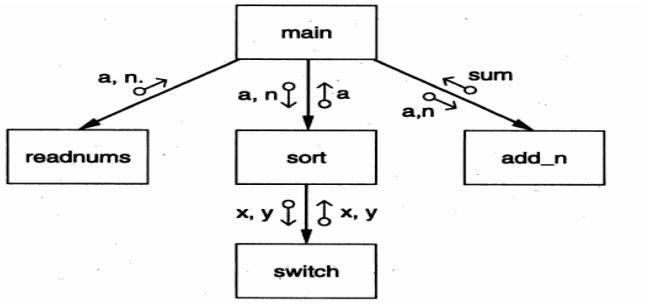

Figura 43: Diagrama de la estructura del Sorter Program

En general, la información de los procedimientos no se representa en estos esquemas y el foco está en representar la jerarquía de los módulos. Sin embargo, hay situaciones donde el diseñador quiere expresar esta información. A su vez, pueden expresarse ciclos, por ejemplo: el módulo A llama al módulo B y luego en un ciclo llama a B y C.

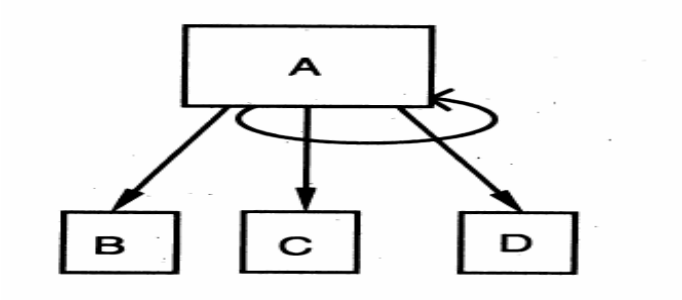

Figura 44: Representación de un ciclo

Si la decisión de llamar a un módulo depende de otro esto puede representarse con un diagrama, ejemplo: el módulo A llama a B y C y según la respuesta de este último llama a D.

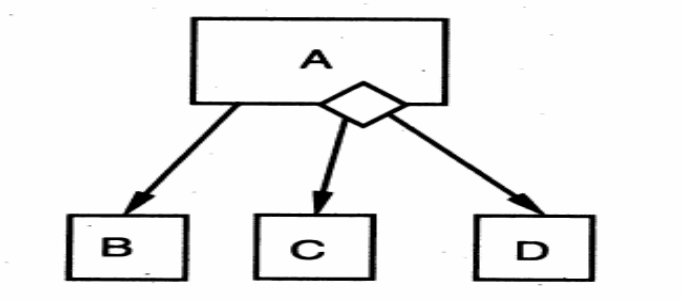

Figura 45: Representación de una decisión según un resultado previo

Los módulos pueden ser del tipo Input o Output, transformadores, a su vez hay módulos que administrasn la información y flujo de datos llamados coordinadores.

Luego el objetivo del diseño es que los programas implementen el mismo tengan una estructura jerárquica, con cohesión funcional entre módulos y con menor cantidad de interconexiones entre módulos tanto como sea posible.

6.2.2. Metodología estructurada de diseño

El principio básico detrás de la metodología del diseño de la estructura es dividir el problema. La primer partición busca distinguir los 3 módulos principales:

1. El administrador de Entradas.
2. El transformador principal.
3. El administrador de las Salidas.

La división es el corazón de esta metodología, en la que hay 4 pasos principales:

1. Reestablecer el problema con un diagrama de flujo de datos (DFDs).
2. Identificar los elementos de Entrada y Salida.
3. Primer nivel de Factorización.
4. Factorizar la entrada, salida y las branches del transformador.

A continuación describiremos cada paso detalladamente:

1. Reestablecer el problema con un diagrama de flujo de datos (DFDs).

   Para usar esta metodología es clave el DFDs, sin embargo hay una diferencia fundamental entre el dibujo del DFD proviniendo de la fase de análisis de requerimientos que se realizó para identificar el dominio del problema. El análista tiene poca información sobre el problema y por lo tanto su tarea es recolectar toda la información posible sobre el mismo y representarlo luego en un DFDs.

   Durante la actividad de diseño estamos tratando con la **solución** del problema y desarrollando un modelo para un eventual sistema. Es decir, el DFD realizado durante el diseño describe cómo fluirán los datos en el sistema cuando esté construido. En este modelado, los principales transformadores o funciones se deciden, y el DFDs muestra las principales transformaciones que va tener y cómo fluirán los datos a través de diferentes transformadores.

   A continuación se muestra un ejemplo de DFDs de un Cajero Automático.

   Hay dos tipos principales de datos de entrada en el diagrama:

   a) El número de cuenta y el código.

   b) El monto a sustraerse.

   Observar el uso de $*$ en diferentes lugares del DFDs.

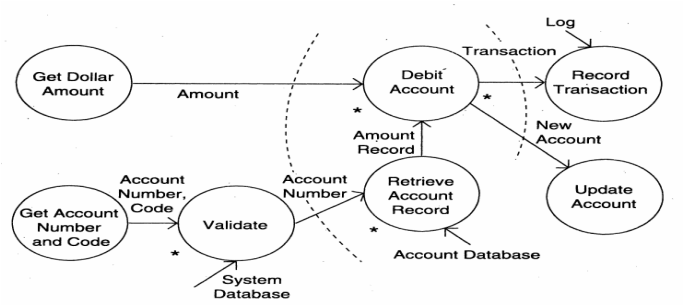

Figura 46: DFDs del Cajero Automático

2. Identificar los elementos de Entrada y Salida.

   En la mayoría de los casos la transformación que realiza el sistema no puede aplicarse directamente a los datos que manejamos de entrada o de salida. El objetivo de este segundo paso es separar en el DFDs, la conversión de Entrada o salida al formato deseado desde el que uno realiza la transformación actual.

   Para este nivel lo más importante es identificar el más alto nivel de abstracción para el Input y el Output. Generalmente son los datos obtenidos desde la entrada física que pasó por una validación, formateo o conversión total. En el DFDs del ejemplo se muestra la separación de estos niveles.

3. Primer nivel de factorización.

   Una vez que tenemos identificado Input/Output y el principal transformador del sistema, estamos listos para identificar algunos módulos para el sistema. Primero especificando un módulo principal cuyo propósito es invocar a los subordinados. El módulo principal es por lo tanto un módulo de coordinación, para cada Input abstracto se crea un módulo subordinado del principal cada uno de estos es un módulo de entrada cuyo propósito es entregar al módulo principal el nivel más abstracto de entrada para el cual fue creado.

   Similarmente se crean módulos subordinados para manejar las salidas.

   Luego cada flecha que une el principal con sus subordinados es enviado con su apropiada dirección.

   Finalmente par cada transformador principal se crea un módulo y se define la subordinación al principal y su relación de Input/Output.

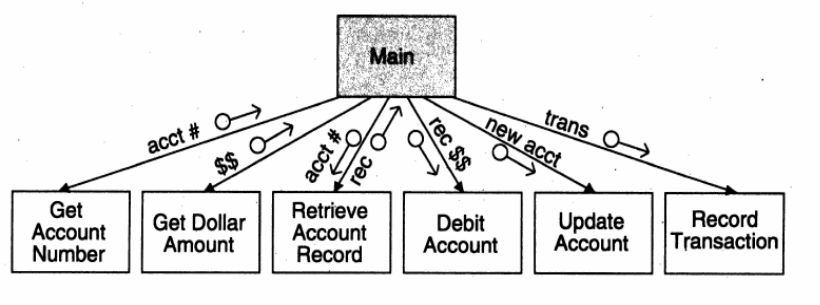

Figura 47: Gráfico del Primer nivel de factorización

4. Factorizar la entrada, salida y las branches del transformador: la idea es reconsiderar cada entrada/salida y transformador como un módulo principal y ver si es posible seguir detallando similarmente como lo hicimos en los pasos $1 \ldots 3$ cada módulo ahora tomado como central.

   No hay un límite claro de que tan preciso o detallado debe ser esta factorización, el principal lineamiento es reconsiderar cada módulo como un nuevo problema y realizar el diagrama correspondiente a ese módulo.

   Esta factorización continuará hasta que los módulos creados sean atómicos.

6.2.3. Ejemplo

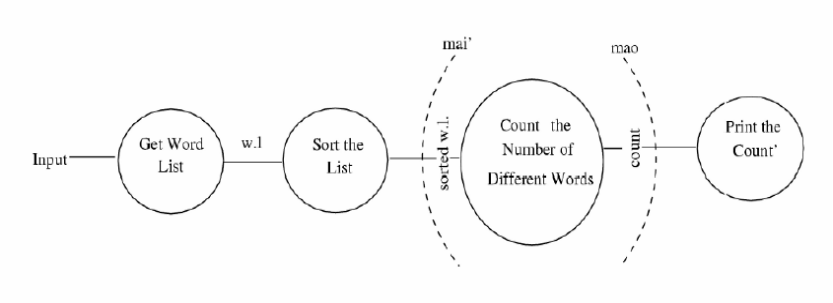

Figura 48: Determinar el número de palabras distintas en un archivo

Este problema tiene un único stream de entrada, el archivo, mientra que la salida deseada es el contador con la cantidad de palabras distintas del archivo.

Antes de comenzar el conteo ordenamos la lista de palabras del archivo, por eficiencia.

Los arcos punteados muestran el nivel más abstracto de Entrada y salida.

El gráfico luego del primer nivel de factorización es el siguiente.

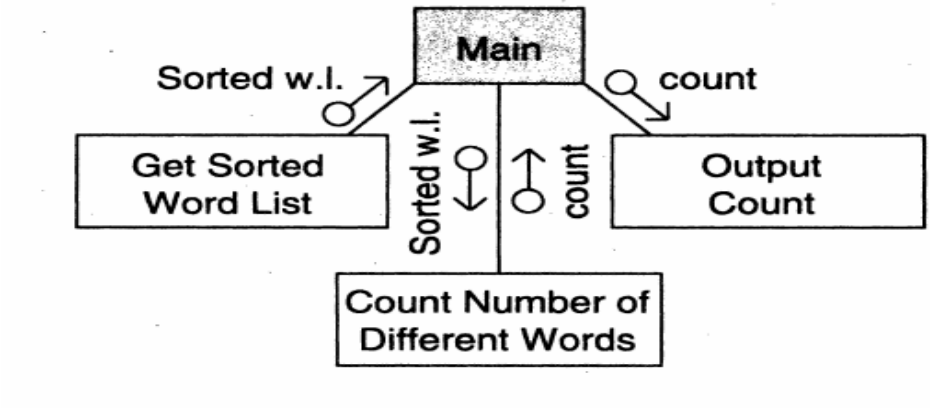

Figura 49: Primer Nivel de Factoreo

Luego la factorización del módulo de entrada puede verse en la siguiente figura donde tomamos el módulo: ordenar la lista, como un problema central.

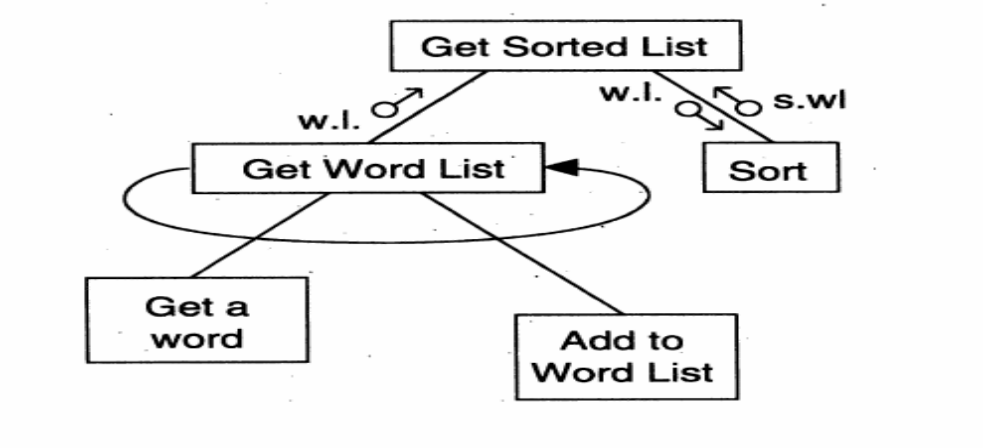

Figura 50: Factorizando el módulo de Input

Luego el módulo central puede factorizarse en el siguiente gráfico:

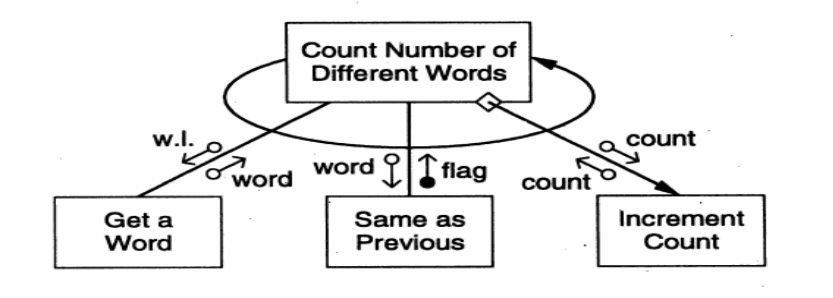

Figura 51: Factorización del módulo central de Transformación

6.3. Diseño orientado a Objetos

Es importante el diseño orientado a objetos ya que los objetos expresan cerradamente un dominio de problema y por esta razón generalmente permanecen más inmunes frente a cambios en los requerimientos.

La herencia y las relaciones cerradas entre objetos son características claves para reusar código efectivamente.

El diseño orientado a objetos suele ser más natural y provee por esta razón estructuras más prácticas para pensar y abstraerse.

6.3.1. Conceptos Orientados a Objetos

- Clases y objetos.
- Encapsulación de datos.
- Un objeto tiene un estado.
- Herencia y polimorfismo (habilidad de un objeto de poder ser de diferentes tipos).

6.3.2. Lenguaje de Modelamiento Unificado (UML)

UML es una notación gráfica para expresar diseños orientados a objetos. Es llamado un lenguaje de modelación y no sólo una notación de diseño ya que permite representar varios aspectos del sistema no sólo el diseño que debe implementarse.

Para un diseño orientado a objetos una especificación de las clases que formarán el sistema puede ser suficiente. Sin embargo mientras modelados, durante el proceso de diseño, el diseñador puede intentar de entender cómo las diferentes clases se relacionan y cómo interactúan para proveer la funcionalidad deseada.

Diagrama de clases

Un diagrama de clases de UML es la principal preza en un diseño o modelado. Como el nombre lo sugiere este diagrama describe las clases que hay en el diseño.

Un diagrama de clases define:

1. Clases que existen en el sistema: además del nombre de la clase de especifica campos claves y los principales métodos de la misma.
2. Asociación entre clases: describe el tipo de relación entre las diferentes clases.
3. Subtipos y relaciones de herencia.

Una clase se representa usualmente como una caja con tres espacios, la superior indica el nombre, la central los atributos y la inferior describe los métodos principales de la clase y si se quiere los parámetros de estos métodos.

A su vez existen estereotipos como $<< Interfaces >>$ que suelen en general agregarse arriba del nombre de la clase.

A su vez en un diagrama de clases puede explicitarse aridades en las relaciones, $1 \ldots *$, herencias (Triángulos) Y Componentes (Rombos).

Importante: el diagrama de clases es distinto a uno de objetos

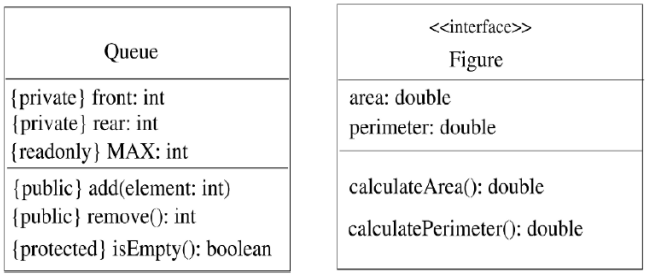

Figura 52: Clases y estereotipos

6.3.3. Una metodología de diseño

Diagramas de secuencia y Colaboración

Los diagramas de clases representan la estructura estática del sistema. Es decir, capturan la estructura del código a implementarse, y cómo las diferentes clases en el código están relacionadas. Los diagramas de clases, sin embargo no representan el comportamiento dinámico del sistema. Es decir, cómo el sistema reacciona cuando se realiza alguna de sus funciones no puede representarse en un diagrama de clases. Esto es lo que intenta capturar un diagrama de secuencias o de colaboración, llamados en conjunto **diagramas de interacción**.

Un diagrama de interacción típicamente captura el comportamiento de un caso de uso y modela cómo los diferentes objetos en el sistema colaboran para implementar un caso de uso. Primero hablaremos de los diagramas de secuencia que son los más conmúnmente usados:

Diagramas de Secuencia

Un diagrama de secuencia muestra la serie de mensajes enviados entre objetos y su orden temporal, cuando objetos colaboran para proveer alguna funcionalidad deseada del sistema. El diagrama de secuencia generalmente dibuja o se usa para representar un caso de uso.

Cuando capturamos el comportamiento dinámico el rol de las clases está limitado a cómo interactúan los objetos en la ejecución

6.3.4. Ejemplo

El problema de conteo de palabras de un archivo

El análisis inicial muestra que hay un objeto File, que está formado por muchos objetos Words. Además uno puede considerar que hay un objeto contador que almacena el número de diferentes Words

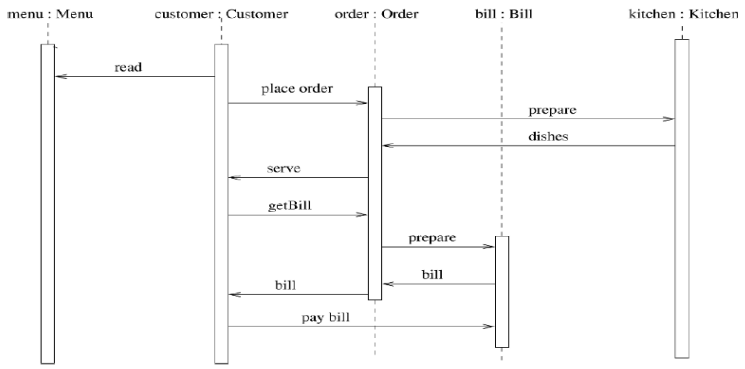

Figura 53: ejemplo de diagrama de secuencia para el restaurant

En el estado de análisis básico sólo encontramos estos tres objetos, sin embargo un análisis posterior revelará que es necesario algún mecanismo de histórico es necesario para verificar si una palabra es única.

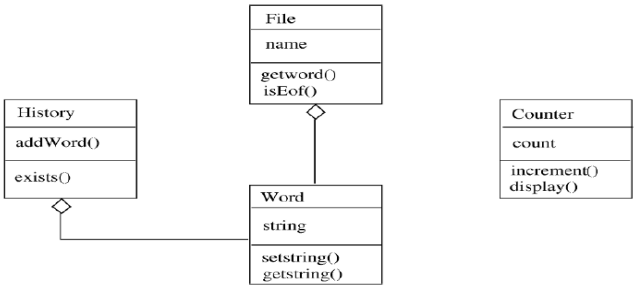

Figura 54: Diagramas de clases para el problema de conteo de palabras

Ahora vamos a considerar el modelado dinámico para este problema. Este es esencialmente un problema de procesamiento de batch, donde un archivo se da como entrada y se devuelve algún tipo de salida por el sistema. Por lo tanto el caso de uso y el escenario para este problema está claro.

Por ejemplo, el escenario para el caso normal puede ser:

1. El sistema pide ingresar el nombre del archivo.
2. El usuario ingresa el nombre del archivo.
3. El sistema verifica la existencia del archivo.
4. El sistema lee las palabras del archivo.

5. El sistema imprime el conteo

Para este simple escenario, no descubrimos nuevas operaciones y nuestro diagrama de clases permanece sin cambios. Ahora consideremos el modelo funcional. Un posible modelo funcional se muestra en la siguiente figura:

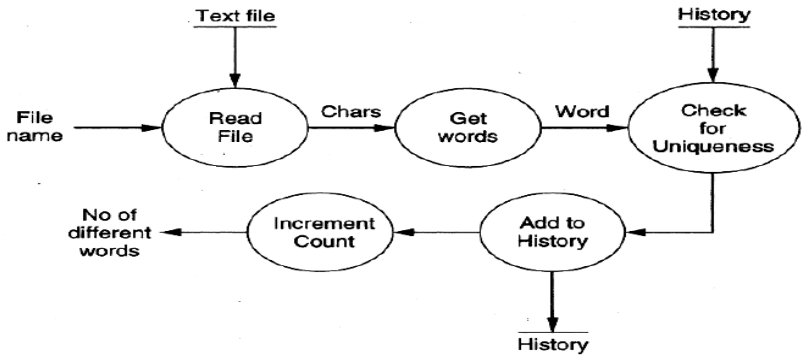

Figura 55: Modelo funcional para el problema de conteo de palabras

El modelo refuerza la necesidad de algún objeto que almacene la historia de las palabras vistas. Este objeto es usado para chequear la unicidad de las palabras. También muestra varias operaciones como Increment(), IsUnique(), AddToHistory() que son necesarias. Estas operaciones deben aparecer como operaciones de clases o como combinación de varias.

En este ejemplo la mayoría de los procesos se reflejan como operaciones en las clases y ya están incorporadas en el diseño.

Ahora estamos en los últimos dos pasos, la metodología de diseño donde nos preocupamos por la implementación y optimización para el modelo de objetos. La primer decisión que tomamos es que el mecanismo de historia se implementará con un árbol binario de búsqueda. Por lo tanto, en lugar de la clase History cambiamos a un árbol binario e búsqueda. Luego para la clase Word, debemos agregar varias operaciones para el manejos de Strings, por lo tanto el modelo final no sufre muchos cambios.

El paso final es la especificación del diseño como un documento. Usamos clases de c++ para nuestra especificación.

```
1
2 Class Word {
3     private:
4         char *String; //String que representa el Word
5     public:
6         bool operator == (Word); // Verifica por la igualdad
7         bool operator < (Word);
8         bool operator > (Word);
```

```
9           Word operator = (Word); // operador de asignación
10          void setWord (char *); //setea el string para la palabra
11          char * GetWord(); // Obtiene el string desde un Word
12  }
```

```
1
2   Class File {
3   private:
4       File inFile;
5       char *fileName;
6   public:
7       Word getWord(); // obtiene el word, llama al operador de Word
8       bool isEof(); //Verifica el fin del archivo
9       void fileOpen (char *);
10  }
```

```
1
2   Class Counter {
3   private:
4       int Counter;
5   public:
6       void increment();
7       void display();
8   }
```

```
1
2   Class Btree: GENERIC in <ELEMENT_TYPE> {
3   private:
4       ELEMENT_TYPE element;
5       Btree<ELEMENT_TYPE> *left;
6       Btree<ELEMENT_TYPE> *right;
7   public:
8       void insert(ELEMENT_TYPE); //insertar
9       bool lookUp(ELEMENT_TYPE); // vacío ?
10  }
```

Como puede verse se explicita todo salvo la implementación.

6.4. Diseño Detallado

En las dos secciones previas vimos dos enfoques diferentes para el diseño del sistema, uno basado en abstracción funcional y el otro en objetos. En el diseño del sistema nos concentramos en los módulos en un sistema y cómo ellos interactúan con otros. Una vez que los módulos se identifican y especifican durante el más alto nivel de diseño, la lógica interna puede implementarse y en esto nos enfocaremos en esta sección.

La actividad de diseño detallado a veces no se realiza formalmente y alcanzar que el código sea consistente con el diseño es muchas veces imposible. Debido a esto, el desarrollo del diseño detallado se realiza muchas veces sólo para los módulos más complejos e importantes.

6.4.1. Diseño Lógico / Algorítmico

El objetivo básivo en el diseño detallado es especificar la lógica para los diferentes módulos que se especifican durante el diseño del sistema. Especificar la lógica requiere desarrollar un algoritmo que implemente la especificación. Aquí vamos a considerar algunos principios para el diseño de algoritmos o lógica que implemente las especificaciones dadas.

El término algorimo es muy general y es aplicable a una muy gran veriedad de áreas. Para el software podemos considerar que un algoritmo es un procedimiento inambiguo para solucionar un problema. Un procedimiento es una secuencia finita de pasos bien definidos u operaciones, donde cada una requiere una cantidad finita de memoria y tiempo para resolverse. En esta definición asumimos que la terminación es una propiedad esencial de los procedimientos.

Hay un número de pasos que uno debe realizar mientras desarrolla un algoritmo.

El paso inicial en el diseño de algoritmos es el **estado del problema**. El problema para el cual vamos a diseñar el algoritmo tiene que estar apropiadamente definido, debe ser claro y preciso. Para el diseño detallado, el estado del problema proviene del diseño del sistema. Es decir, ya está disponible cuando comienza el diseño detallado.

El paso siguiente es el desarrollo de un modelo matemático para el problema. En el modelado, uno tiene que elegir las estructuras matemáticas más adecuadas para el problema. En la mayoría de los casos estos modelos son similares a otros ya realizados.

El siguiente paso es el diseño del algoritmo, durante este paso de decide la estructura de datos y la estructura del programa. Una vez que se diseña el algoritmo debe verificarse que sea correcto.

El método más común utilizado para el diseño de algoritmos o la lógica de un módulo es usar la técnica de refinamientos por paso.

Esta técnica divide el diseño lógico en una serie de pasos, que el desarrollador puede realizar gradualmente. El proceso comienza convirtiendo la especificación del módulo en una descripción abstracta de un algoritmo conteniendo unas pocos estados abstractos.

En cada paso se van dividiendo estos estados y detallando con instrucciones. Este refinamiento sucesivo cuando todas las instruciones son lo suficientemente precisas. Generalmente no se realiza en ningún lenguaje pero si se consideran ciclos y condicionales.

6.4.2. Estado de Modelado de Clases

Para el diseño orientado a objetos, el enfoque que acabamos de discutir para obtener el diseño detallado puede utilizarse para el diseño de la lógica de los métodos. Pero una clase no es una abstracción funcional y no puede ser vista como una mera colección de funciones o métodos.

La técnica para obtener un mayor entendimiento de las clases es ver las clases como un ciclo, sin hablar acerca de la lógica de los diferentes métodos, debe ser

diferente al del enfoque basado en el refinamiento.

Un objeto de una clase tiene un estado y muchas operaciones en él. Para comprender operaciones y debe entenderse el efecto de la interacción de varias operaciones. Esto puede ser visto como uno de los objetivos de la actividad de diseño detallado para el desarrollo orientado a objetos. Una vez que se comprende el comportamiento general pueden desarrollarse los algoritmos para varios métodos.

Por ejemplo tomemos el siguiente diagrama de estados para una Pila:

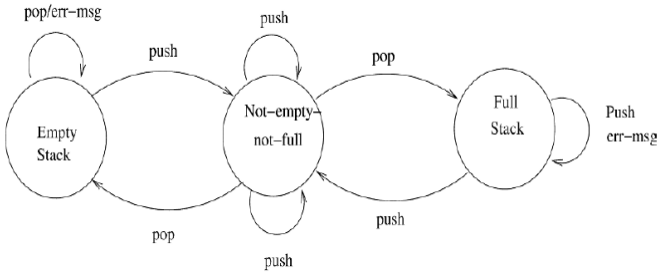

Figura 56: Automata de estados para una Pila

6.5. Verificación

La salida de la actividad de diseño debe ser verificada antes de proceder con las actividades de la siguiente fase. Si el diseño está expresado en alguna notación formal para el cual existen herramientas formales de análisis entonces podemos chequear consistencia interna fácilmente.

En caso contrario, por ejemplo si se realizó en algún lenguaje ejecutable, pueden usarse otras herramientas para verificar el enfoque más común para verificaciones es realizar revisiones en el diseño.

El propósito de las revisiones de diseño es asegurar que el diseño satisfaga los requerimientos y que sea de calidad. Si hay errorres durante el proceso de diseño se reflejarán en el código y en el sistema final. Como el costo de remover fallas causadas por errores que ocurrren durante el diseño incrementa con el tiempo que demorar en detectar los errores, es mejor si los errores se detectan tempranamente, antes de que se manifiesten en el sistema. Detectar errores en el diseño es el propósito de las revisiones del diseño.

El proceso de revisión es similar al proceso de inspección, en donde un grupo de personas tiene una charla discutiendo sobre el diseño con el ánimo de revelar errores de diseño o de propiedades no deseadas. El grupo de revisión debe incluir un miembro de los que realizaron la SRS, el autor y encargado del mantenimiento del diseño y un ingeniero independiente encargado de la calidad.

Como cualquier revisión el objetivo es encontrar errores no intentar disimularlos o resolverlos.

El error más importante en el diseño es que no soporte enteramente algunos requerimientos. Por ejemplo algún caso de excepción que no pueda ser manejado o alguna restricción que el diseño no puede cumplir. Para la calidad del diseño, la modularidad es el principal criterio. Sin embargo, la eficiencia puede ser otro aspecto clave a validar para ver la calidad de un diseño y su evaluación.

6.6. Métricas

El tamaño es siempre una métrica del producto de interés. Para el tamaño de un diseño, el número total de módulos es comúnmente la métrica usada (usando un promedio del tamaño de los módulos se puede estimar y comparar con otros proyectos).

Otra métrica de interés es la complejidad de los módulos

6.7. Complejidad de Métricas para el diseño orientado a funciones

Impuresa del Grafo $= n - e - 1$
Donde n es el número de nodos en el diagrama de estructura.
e números de lados.

Métricas de información de flujo

$$D_c = size * (IN_{flow} * OUT_{flow})^2$$

- IN_{flow} Entrada a un módulo
- OUT_{flow} salida de un módulo
- D_c Complejidad de Diseño.

Una variante

$$D_c = fan_{in} * fan_{out} + IN_{flow} * OUT_{flow}$$

- fan_{in} número de módulos que llaman a este módulo
- fan_{out} número de módulos al que llama a este módulo

Luego los módulos se clasifican en :

Propenso a Error	$D_c >$ Complejidad promedio + Desvío estándar
Complejo	Complejidad promedio $< D_c <$ Complejidad promedio+ Desvío estándar
Normal	Caso contrario

6.8. Complejidad de Métricas para el diseño orientado a Objetos

- WMV: Weight Method per Class

$$WMC = \sum_{i=1}^{n} C_i$$

- Profundidad del árbol de herencia.
- Acoplamiento entre clases: (CBC) , número de clases conectadas.
- Responses for a Class: cuantifica el número de métodos que pueden ser invocados por un objeto de esta clase.

6.9. Resumen

Ejercicios

7. Testing

En el desarrollo de un proyecto de software los errores pueden introducirse durante cualquier etapa de desarrollo. Aunque algunos errores pueden detectarse luego de cada fase por técnicas como inspecciones, otros permanecen ocultos. En última instancia estos errores se reflejan en el código. Por lo tanto, el código final contiene algunos errores de requerimientos y del diseño sumados a los errores introducidos durante la codificación. Para asegurar la calidad del software que finalmente será entregado estos defectos deben ser removidos.

Existen dos tipos de enfoques para identificar defectos en el software, estáticos y dinámicos. En el análisis estático, el código no se ejecuta pero sí se evalúa a través de algún proceso o de herramientas para hallar errores. Inspecciones de código por ejemplo sigue un enfoque estático. En el análisis dinámico, el código se ejecuta y la ejecución se usa para determinar defectos. Testing es la actividad más común que sigue este enfoque y a su vez juega un rol muy importante para asegurar la calidad.

Durante el testing el software bajo tests (SUT), se ejecuta con un conjunto finito de casos de test y el comportamiento del sistema para esos casos de test es evaluado para determinar si el sistema realiza lo esperado. El propósito básico del testing es incrementar la confianza en el funcionamiento del software testeado. Como el testing es extremadamente caro y puede consumir demasiado esfuerzo, un objetivo práctivo es alcanzar tanto la confianza como la eficiencia. Claramente la efectividad y eficiencia del testing depende críticamente de los casos de test elegidos. Por lo tanto la mayoría del capítulo está dedicado a la elección de estos casos de test.

7.1. conceptos de Testing

En esta sección primero daremos la definición de los términos que comúnmente son usados cuando hablamos de testing y luego hablaremos de algunas normas básicas relacionadas a cómo se realiza el testing y la importancia de la psicología del tester.

7.1.1. Error, Defecto (Fault) y desperfecto (Failure)

Cuando hablamos de testing comúnmente hablamos de error, defecto y desperfecto. Comenzaremos definiendo estos tres conceptos claramente.

El término **error** es usado de diferentes maneras. Se refiere a la discrepancia entre el valor computado, observado o medido y el verdadero, especificado o teóricamente valor concreto. Es decir el error se refiere a la diferencia entre la actual salida del software y la correcta salida. En esta interpretación, el error es esencialmente una medida de la diferencia entre el actual y lo ideal. El error también es usado para referirse a una acción humana que resulta que el software contenga un defecto (Fault). Este concepto es general y abarca todas las fases del proyecto.

Un **defecto** (Fault) es una condición que causa que la función falle respecto de los requerimientos. Un defecto es la razón básica para el mal funcionamiento del software y es prácticamente un sinónimo de lo que usualmente se denomina **Bug**.

Un **desperfecto** (Failure) es la incapacidad del sistema o de una componente de realizar la función requerida de acuerdo a las especificaciones. Un desperfecto del software ocurre si el comportamiento del software difiere del comportamiento especificado. Desperfectos pueden ser causados por factores funcionales o de performance. Note que la definición no implica que un desperfecto sea observable. Es posible que un desperfecto pueda ocurrir pero no pueda ser detectado.

Hay algunas implicaciones de estas definiciones. La presencia de un error implica que hay un desperfecto, y observar un desperfecto implica que un defecto puede estar presente en el sistema.

Esto tiene consecuencias directas durante el testing. Si durante el testing no observamos ningún tipo de error no podemos decir nada acerca de la presencia o ausencia de defectos en el sistema. Si, por otra parte, observamos algún tipo de desperfecto, podemos decir que hay algún tipo de defecto en el sistema. Esta relación entre defecto y desperfecto hace que la tarea de elegir casos de test sea muy desafiante, un objetivo durante la elección de casos de test es elegir aquellos que revelen los defectos, si es que existen. Idealmente quisiéramos aquel conjunto de casos de tes que si algún tipo de defecto existe en el sistema algún caso de test lo revele, algo imposible de alcanzar en la mayoría de las situaciones.

7.1.2. Test Case, Test Suite and Test Horness

Un caso de test (a veces llamado test) puede ser considerado como un conjunto formado por tests inputs y condiciones de ejecución que están diseñadas para ejecutar el software de una manera en particular. Generalmente, un caso de test también especifica la salida esperada de la ejecución del test bajo esas condiciones y entradas. Un grupo de casos de test relacionados que generalmente se ejecutan juntos para verificar algún tipo de comportamiento o aspecto específico generalmente es referido como un **test suite**.

Note que en un caso de test, las entradas y las condiciones de ejecución son mencionadas por separado. Tests inputs son los valores específicos de los parámetros u otros inputs que se dan al sistema por el usuario u otro programa. Las condiciones de ejecución, por otra parte reflejan el estado del sistema y el ambiente que también tendrá impacto sobre el sistema. Entonces por ejemplo mientras testeamos una función para agregar un registro a una base de datos si el registro no existe actualmente, el comportamiento de la función dependerá tanto de los parámetros que le pasemos a la función como del estado actual de la base de datos. Por lo tanto un caso de test necesita abarcar ambas condiciones.

El testing puede realizarse con un tester que ejecute cada suite test. Esto suele ser un proceso muy pesado, especialmente si la lista de casos de test es muy larga. Por lo tanto actualmente se está investigando e intentando de automatizar el testing.

Generalmente automatizar es llamar a una función que corra algún script

sobre el software y vaya mostrando el resultado de la comparación entre lo esperado y lo obtenido.

Actualmente existes muchos frameworks que permiten realizar estas actividades de una manera más simple. Un framework de test es usualmente llamado **test harness**.

7.1.3. Psicología del Testing

Como mencionamos, disponer de un conjunto de casos de test que nos garanticen que se detectarán todos los errores no es factible. Más aún, no hay métodos formales o precisos para elegir casos de test. Incluso aunque hay un gran número de heurísticas y reglas para determinar los casos de test, elegir los casos de test es una actividad creativa basada en el ingenio del tester. Por esta razón la psicología de una persona que realiza test se vuelve importante.

El propósito básico del testing es detectar errores que pueden estar presentes en el programa. Por lo tanto, uno no empieza el testing con la intención de mostrar que el programa funciona, más aún debiera intentar que el programa no funciona. Debido a esto, el testing puede ser definido como el proceso de ejecutar un programa con la intención de encontrar errores.

Esta intención propia del testing no es trivial ya que los casos de test son elegidos por humanos que son consciente o inconscientemente quieren demostrar que el programa funciona bien lo harán, por el contrario si quieren demostrar que el programa no funciona, lo tomará como un desafío de intentar encontrar la mayor cantidad de errores. Testing es esencialmente un **proceso destructivo**, donde el tester debe tratar al programa como un adversario al quien quieren vencer encontrando errores. Esta es una de las razones por la cual muchas organizaciones tercerizan el testing o tienen personal especializado en testing que no formó parte de la construcción del sistema.

7.1.4. Niveles de Testing

Testing está usualmente relacionado a la detección de defectos (Faults) existentes desde etapas tempranas, sumado a los defectos durante la codificación propiamente dicha. Debido a esto, diferentes niveles de testing se usan en el proceso de testing, cada nivel de test intenta verificar diferentes aspectos del sistema.

Los niveles básicos son:

1. Unit Testing
2. Integration Testing
3. System testing
4. Acceptance Testing

Estos distintos niveles intentan detectar distintos tipos de defectos, la relación de defectos introducidos en distintas fases y los diferentes niveles de testing se muestran en la siguiente figura.

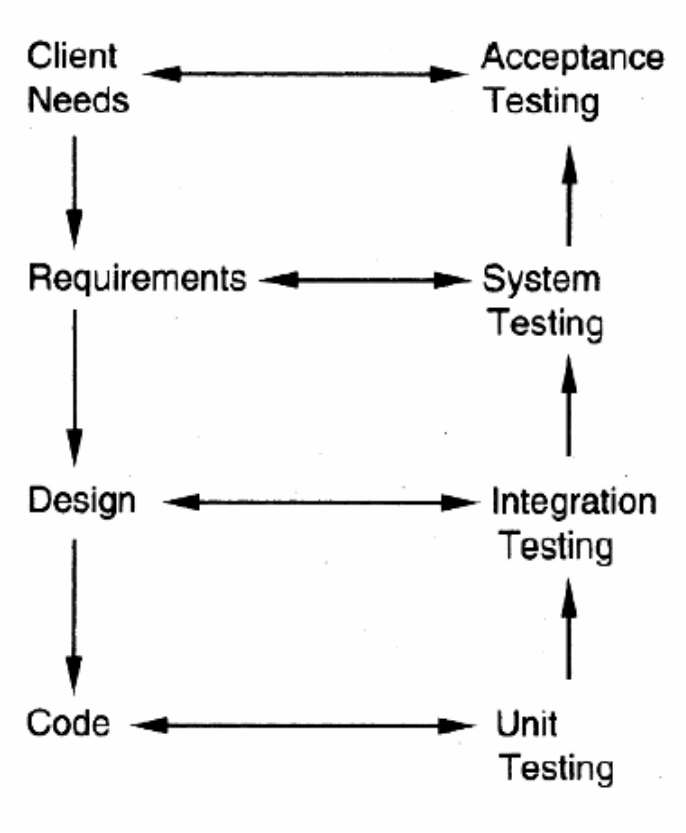

Figura 57: Niveles de Testing

Unit testing: es esencialmente para la verificación de un código producido por programadores individualmente y es típicamente hecho por el programador del módulo. Generalmente un módulo ofrecido por un programador se entrega y usa por otros sólo después de que el test de unidad fue satisfactorio.

El siguiente nivel es el llamado **test de integración** en el cual se testean las relaciones entre módulos que forman subsistemas. El objetivo aquí es ver si los módulos pueden integrarse apropiadamente. Por lo tanto el énfasis está en las interfaces de los módulos. Esta actividad puede ser vista como el testing del diseño.

Los siguientes niveles son el testing del sistema y testing de aceptación. Aquí el sistema entero es verificado. El documento de referencia para este proceso es el documento de requisitos y el objetivo es ver si el software cumple los requisitos. Esto a veces se hace muy grande para proyectos largos, pudiendo tardar de semanas a meses. Es esencialmente un ejercicio de validación y en muchas situaciones es la única actividad con este fin. El **testing de aceptación**

es a veces realizado con datos realistas del cliente para demostrar que el software trabaja satisfactoriamente. A su vez puede realizarse en las condiciones en que correrá el sistema. El test de aceptación esencialmente verifica si el sistema resuelve el problema para el cual se realizó.

Estos niveles de testing se realizan desde que el sistema comenzó a codificarse en módulos.

Existe otro nivel de testing llamado **testing de regresión** que se realiza cuando se realizan cambios en un sistema actual. Es decir debemos verificar que el nuevo diferencial agregado al sistema funcion acordemente y que no cause fallas en el sistema que venía funcionando.

Para el testing de regresión pueden mantenerse algunos casos de test del viejo sistema. Estos casos de tests se ejecutan nuevamente en el sistema modificado y debe compararse su salida con la anterior para asegurarse que el sistema esté funcionando igual en esos casos de test. Esto generalmente es la tarea más demandante.

La regresión completa de test de grandes sistemas puede tomar una gran cantidad de tiempo, incluso si se automatiza. Por esta razón si pequeños cambios se realizaron sólo se prueba el subconjunto de tests relacionados. Esta tarea requiere la elección adecuada de las partes a las que puede afectar este nuevo cambio.

7.2. El proceso de Testing

El objetivo básico del proceso de desarrollo de software es producir software que no tenga errores o muy pocos. El testing es la actividad que se enfoca en identificar defectos (que deben ser removidos).

Hemos visto que diferentes niveles de testing son necesarios para detectar defectos inyectados durante varias tareas en el proyecto.

Y por cada nivel muchos sistemas pueden ser testeados. Para testear cada sistema deben diseñarse casos de test y luego ejecutarse. En general el testing en un proyecto es una tarea compleja que a su vez consume el esfuerzo máximo. Por lo tanto, el testing en un proyecto debe realizarse apropiadamente. El proceso de testing para un proyecto consiste de tres tareas a alto nivel:

1. Planeamiento de Tests
2. Diseño de casos de Tests
3. Ejecución de los tests

7.2.1. Plan de Tests

En general, en un proyecto, el testing comienza con un plan de test y finaliza con una ejecución satisfactoria de los test de aceptación. Un plan de test es un documento general para el proyecto entero que define el alcance, el enfoque que se tomará y la agenda de testing, a su vez identifica los items a testear y el personal responsable para las diferentes actividades de testing.

El planeamiento de test puede hacerse también antes de que el testing actual comience y puede hacerse en paralelo con la codificación y actividades de diseño.

Las entradas para formar el plan de test son:

1. Plan de proyecto
2. SRS
3. Arquitectura o documento de Diseño.

El plan del proyecto es necesario para asegurar que el plan de tests es consistente con el plan de calidad general para el proyecto y la agenda de testing matchee con el plan del proyecto. El documento de requisitos y el documento de diseño son los documentos básicos usados para elegir los tests de unidad y decidir el enfoque que se usará durante el testing. Un plan de test debiera contener lo siguiente:

- Especificación de test de unidad.
- Features que serán testeados.
- Enfoque para el testing.
- Desarrollo de tests.
- Agenda y tareas de distribución de recursos.

Como vimos anteriormente, diferentes niveles de testing deben hacerse en un proyecto. Los niveles se especifican en el plan de test identificando las unidades de test para el proyecto. Una unidad de tests es un conjunto de una o más módulos que forman el software a testear. La identificación de las unidades de tests establece los diferentes niveles de testing que se realizarán en el proyecto. Generalmente, un número de unidades de test a formarse durante el testing, empezando desde los módulos de nivel más bajos, que deben testearse individualmente se especifican como unit test. Luego se especifican las unidades de nivel más alto que pueden ser una combinación de unidades ya testeadas o pueden ser combinaciones de unidades testeadas y no testeadas.

La idea básica detrás del test units es asegurarse que el testing se realice incrementalmente, donde cada incremento incluye sólo una poca cantidad de aspectos que necesitan testearse.

Un importante factor mientras formamos una unit test es la "testability" de una unidad. Una unidad debiera sea fácil de testear. En otras palabras, debiera ser posible formar unos pocos casos de test y ejecutar la unidad sin mucho esfuerzo con estos casos de tests.

Features a testearse incluye todos los features de software y combinaciones de estos que debieran ser testeados. Un feature es una característica del software específica o implicada por los requerimientos o el diseño, esto puede incluir funcionalidad performance, restricciones de diseño y atributos.

El enfoque para el testing especifica el enfoque general a seguirse por el actual proyecto. Las técnicas que se usarán para juzgar el esfuerzo de testing

también deberían ser especificadas. Esto a veces es llamado **criteria testing** o el criterio para la evaluación del conjunto de casos de test usados en el testing.

testing deliverables podría ser una lista de casos de test a usarse, resultados detallados del testing incluyendo la lista de defectos encontrados, el reporte de resumen de test y cualquier otro dato afín.

También el plan de test especifica la agenda y el esfuerzo a gastarse en distintas actividades de tests y herramientas que deben ser usadas. Esta agenda debiera ser consistente a la planificación general del proyecto.

7.2.2. Diseños de casos de Test

El plan de test se enfoca en cómo avanzará el testing según el proyecto, qué unidades serán testeadas y qué enfoques y herramientas serán usadas durante varias etapas de testing. Sin embargo, no habla de los detalles de una unidad de testing.

El diseño de los casos de test debe hacerse para cada unidad por separado basado en el enfoque especificado en el plan de test y las features a ser testeadas, se diseñan y especifican los casos de test para testear cada unidad. La especificación de los casos de test otorga, para cada unidad a ser testeada todos los casos de test, los inputs a usarse en los casos de test, las condiciones en que debe correrse el caso de test y las salidas esperadas para ese caso de test, si los casos de test se especifican en un documento, la especificación lucirá como la siguiente tabla.

Requirement Number	Condition to be tested	Test data and settings	Expected output

Figura 58: documento de Test

A veces se proveen una pocas columnas extras para guardar la salidas de distintas rondas de testing. Es decir, a veces el documento de la especificación de los casos de test se usa para registrar el resultado del test. Generalmente poniendo un pass or Fail.

Con frameworks de testing y test automatizados, los scripts de test pueden ser considerados como la especificación de los casos de test, ya que ellos muestran claramente la entrada y salida esperada.

El diseño de los casos de test es la principal actividad en el proceso de testing. Una selección cuidadosa de los casos de test que satisface los criterios y el enfoque especificado es esencial para un testing apropiado.

Existen muy buenas razones por las que los casos de tests pueden especificarse antes de que sean usados para testing. Es importante asegurar que los casos de test usados sean de alta calidad. La evaluación de los casos de test se realiza generalmente a través de una revisión. Como en cualquier revisión, es necesario un documento formal para la revisión de los casos de tests por lo tanto se puede ir revisando y mejorando la calidad de los tests elegidos.

Otra razón para especificar los casos de test en un documento o script es que cuando se está haciendo esto, el tester puede ver si los testing son o no de calidad. Este tipo de evaluación es difícil de hacer en el aire, a su vez permite ir optimizando el número de casos de test de un suite test si se considera que alguno es redundante.

7.2.3. Ejecución de los casos de Test

Luego de la especificación de los casos de tests el siguiente paso en el proceso de testing es su ejecución. Esto no siempre es tan lineal ya que puede requerir la construcción de módulos para luego recién poder ser ejecutados. Este paso es lineal en caso de usar frameworks de test.

Durante la ejecución de los casos de tests se encuentran defectos, estos deben ser formalmente advertidos y apropiadamente logueados para que puedan ser corregidos.

Ahora veremos el ciclo de vida de un defecto. Un defecto puede ser encontrado por cualquiera en cualquier momento. Cuando un defecto se encuentra debe ser logueado en un control de defectos del sistema, con la información suficiente acerca del defecto. Luego este defecto está en estado "submitted", luego este defecto debe ser acomodado al que (generalmente) escribió ese código, el autor se entra y comienza su etapa de "debugging", el defecto ahora está en estado "fixed". Luego que se corrije pasar por una verificación antes de ser cerrado, generalmente es otra persona el que verifica y para al estado "closed".

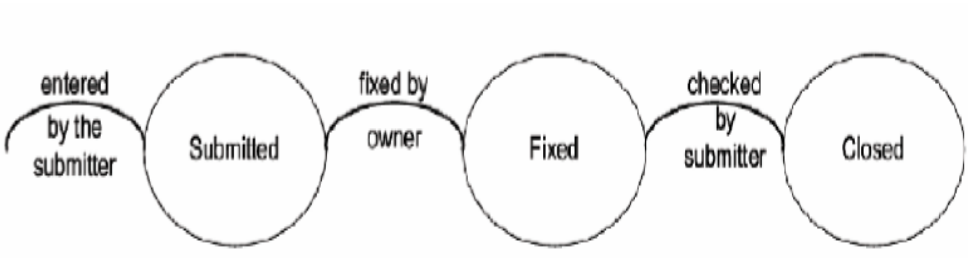

Figura 59: Ciclo de vida de un Defecto

7.3. Testing de Caja Negra

Como hemos visto un buen diseño de los casos de test es la clave para un adecuado testing de un sistema. El objetivo mientras testeamos software es encontrar la mayoría de los defectos o todos con la menor cantidad de test como sea posible. Debido a este principio la elección de los casos de test debe realizarse cuidadosamente.

Existen dos enfoques básicos para diseñar los casos de test que se usarán durante el testing: caja negra y caja blanca. En el testing de caja negra no se considera la estructura del programa. Los casos de tests se eligen únicamente basado en los requerimientos o la especificación de los módulos.

En el testing de caja negra, el test sólo conoce las entradas que puede darle al sistema y qué salidas éste debería devolver. En otras palabras las bases para decidir los casos de tests es la SRS o especificación de los módulos. Esta forma de testing a veces es llamada **funcional** o testing de comportamiento.

No hay reglas formales para diseñar los casos de test para caja-negra. Sin embargo, hay un gran número de técnicas o heurísticas que pueden ser usadas para elegir adecuadamente los casos de tests efectivamente.

7.3.1. División por clases de Equivalencia

Porque no podemos hacer un testing exhaustivo, el siguiente enfoque natural es dividir el dominio de entrada en un conjunto de clases de equivalencia, entonces si el programa funciona bien para un valor entonces funcionará correctamente para todos los valores de esa clase. Si podemos identificar todas las clases entonces testear para un valor de cada clase es equivalente a hacer un testing exhaustivo del programa.

Sin embargo al desconocer la estructura interna del sistema e incluso conociéndola suele ser imposible determinar todas las clases de equivalencia por lo que este enfoque intenta acercarse lo más posible al número total de clases.

Cada grupo de entradas que sabemos que va a producir un comportamiento distinto del sistema debe estar en una clase separada. Por ejemplo un test para un programa que devuelve el valor absoluto de un número real, debe como mínimo contener dos clases números reales positivos y los negativos, a su vez si queremos un software más robusto debieramos probar que funciona para entradas inválidas.

Un enfoque común para determinar las clases de equivalencia es el siguiente: si hay alguna razón para creer que el total rango de entradas no va a ser tratado de la misma manera entonces el rango debe dividirse en dos o más clases de equivalencia, cada una conteniendo los valores para los que consideremos que el comportamiento sea distinto.

A su vez cualquier valor que consideremos que el comportamiento general difiere es entonces una clase de equivalencia por ejemplo el cero en los enteros.

Una vez que se eligieron las clases de equivalencia para cada entrada entoces debemos elegir los casos de test adecuadamente.

También se deben considerar las clases de equivalencia de los datos de salida. Generar los casos de test para estas clases eligiendo apropiadamente las entradas

Hay dos maneras diferentes de elegir los casos de test

1. Elegir un test que contemple la mayoría de las entradas válidas y uno por cada entrada inválida.

2. Dar un caso de test que cubra a lo sumo una clase válida para una entrada y uno por cada entrada inválida.

Ejemplo:Considerar un programa que toma dos entradas: un string s de longitud N y un entero n. El programa determina los n caracteres más frecuentes.

Quien realiza el test cree que el programador puede haber tratado separadamente a los distintos tipos de caracteres.

Entrada	Clases de equivalencia Válidas	Clases Inválidas
s	1. Contiene números 2. Contiene letras mayúsculas 3. Contiene letras minúsculas 4. Contiene caracteres especiales 5. longitud del String $\leq$ N	1. Caracteres no ASCII 2. N $<$ longitud del String
n	6. Está en el rango válido	3. Está fuera del rango válido

Casos de test (i.e. los valores de s y n)

- Según el método 1:
  - s con long $< N$ y conteniendo mayúsculas, minúsculas, números y caracteres especiales, y n=4.
  - Más un caso de test por cada clase de equivalencia inválida.
  - Total: $1 + 3 = 4$ casos de test.
- Con el método 2:
  - Un string aparte por cada caso de test (i.e. uno para números, uno para minúsculas, etc.)
  - Más los casos inválidos.
  - Total: $5 + 3 = 8$ casos de test.

7.3.2. Análisis de valores límites

Los programas generalmente fallan sobre valores especiales. Estos valores usualmente se encuentran en los límites de las clases de equivalencia.

Los casos de test que tienen valores límites tienen alto rendimiento.

También se denominan casos extremos.

Un caso de test de valores límites es un conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencias de la entrada o la salida.

Para cada clase de equivalencia: Elegir valores en los límites de la clase. Elegir valores justo fuera o dentro de los límites. (Además del valor normal).

Ejemplo: $0 \leq x \leq 1$

0 y 1 están en el límite, $-0{,}1$ y 1,1 están apenas afuera y 0,1 y 0,9 están dentro.

Ejemplo: para una lista acotada: la lista vacía y la lista de mayor longitud.

Considerar las salidas también y producir casos de test que generen salidas sobre los límites.

En primer lugar se determinan los valores a utilizar para cada variable.

Si la entrada tiene un rango definido $\Rightarrow$ hay 6 valores de límite más un valor normal (total: 7).

Para el caso de múltiples entradas hay dos estrategias para combinarlas en un caso de test:

1. Ejecutar todas las combinaciones posibles de las distintas variables (si hay n $\Rightarrow$ hay 7^n casos de test!).

2. Seleccionar los casos límites para una variable y mantener las otras en casos normales y considerar el caso de todo normal (total: $6n + 1$).

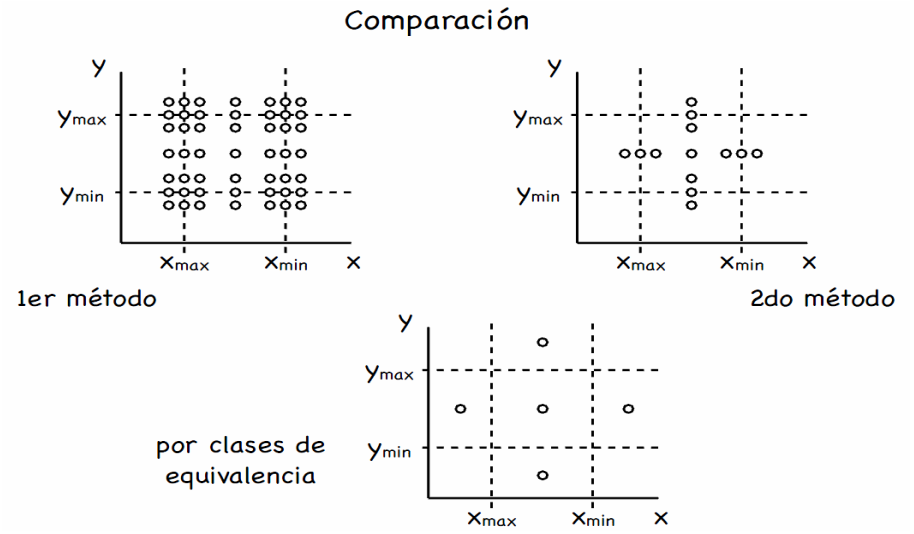

Figura 60: Comparación de los distintos tests suites

7.3.3. Grafo de causa Efecto

Los análisis de clase de equivalencia y valores límites consideran cada entrada separadamente.

Para manipular las entradas distintas combinaciones de las clases de equivalencia deben ser ejecutadas.

La cantidad de combinaciones puede ser grande: si hay n condiciones distintas en la entrada que puedan hacerse válidas o inválidas $\Rightarrow$ hay 2^n clases de equivalencia.

El grafo de causa-efecto ayuda a seleccionar las combinaciones como condiciones de entrada.

Identificar las causas y efectos en el sistema

Causa: distintas condiciones en la entrada que pueden ser verdaderas o falsas.

Efecto: distintas condiciones de salidas (v/f también).

Identificar cuáles causas pueden producir qué efectos; las causas se pueden combinar.

Causas y efectos son nodos en el grafo.

Las aristas determinan dependencia: hay aristas "positivas" y "negativas"

Existen nodos "and" y "or" para combinar la causalidad.

A partir del grafo de causa-efecto se puede armar una tabla de decisión.

Lista las combinaciones de condiciones que hacen efectivo cada efecto.

La tabla de decisión puede usarse para armar los distintos casos de test.

Ejemplo

Una base de datos bancaria que permite dos comandos:

- Acreditar una cant. en una cuenta.
- Debitar una cant. de una cuenta.

Requerimientos:

- Si se pide acreditar y el número de cuenta es válido $\Rightarrow$ acreditar.
- Si se pide debitar, el número de cuenta es válido, y la cantidad es menor al balance $\Rightarrow$ debitar.
- Si el comando es inválido $\Rightarrow$ mensaje apropiado.
- Causas:
  - C1: El comando es "acreditar"
  - C2: El comando es "debitar"
  - C3: Número de cuenta es válido
  - C4: La cantidad es válida
- Efecto:

- E1: Imprimir “Comando inválido”
- E2: Imprimir “número de cuenta inválida”
- E3: Imprimir “cant. débito inválida”
- E4: Acreditar en la cuenta
- E5: Debitar de la cuenta

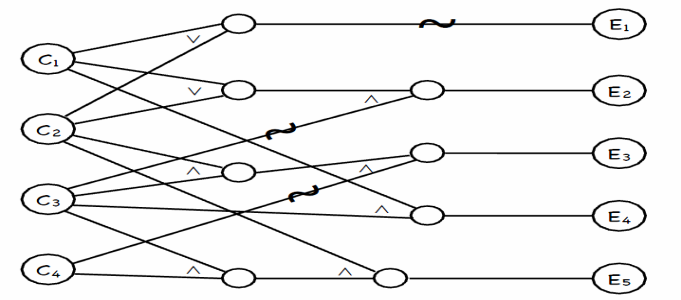

Figura 61: Grafo de causa Efecto del ejemplo

#	1	2	3	4	5	6
C1	0	1	x	x	x	1
C2	0	x	1	1	1	x
C3	x	0	0	1	1	1
C4	x	x	x	0	1	x
E1	1					
E2		1	1			
E3				1		
E4					1	
E5						1

7.3.4. Testing de a Pares

Usualmente muchos parámetros determinan el comportamiento del sistema.

Los parámetros pueden ser entradas o seteos y pueden tomar distintos valores (o distintos rangos de valores).

Muchos defectos involucran sólo una condición (defecto de modo simple).

Ejemplo: el sw no puede imprimir en un tipo dado de impresora

Los defectos de modo simple pueden detectarse verificando distintos valores de los distintos parámetros.

Si hay n parámetros con m (tipos de) valores c/u, podemos testear cada valor distinto en cada test por cada parámetro.

I.e: $n * m$ casos de test en total.

Pero no todos los defectos son de modo simple: el sw puede fallar en combinaciones:

Ej: el sw de la cuenta del tel. calcula mal las llamadas nocturnas (un parámetro) a un país particular (otro param.)

Ej: el sistema de reserva falla en las reservas en clase ejecutiva (param. 1) para menores (param. 2) que no abarcan un fin de semana (param. 3)

Los defectos de modo múltiple se revelan con casos de test que contemplen las combinaciones apropiadas.

Esto se denomina **test combinatorio**.

Pero el test combinatorio no es factible:

n parámetros / m valores $\Rightarrow$ hay n^m casos de test.

Si $n = m = 5$, $n^m = 3125$. Si cada test 5 min $\Rightarrow$ más de un mes testeando.

Se investigó que la mayoría de tales defectos se revelan con la interacción de pares de valores.

I.e. la gran mayoría de los defectos tienden a ser de modo simple o de modo doble.

Para modo doble necesitamos ejercitar cada par $\Rightarrow$ se denomina testing de a pares.

En testing de a pares, todos los pares de valores deben ser ejecutados. Si n parámetros / m valores, para cada par de parámetros tenemos $m * m$ pares:

El 1er param. tendrá $m * m$ contra n-1 otros.

El 2do param. tendrá $m * m$ contra n-2 otros

El 3er param. tendrá $m * m$ contra n-3 otros

etc.

Total: $m^2 * n * (n - 1)/2$

En el ejemplo: $5^2 * 5 * 4/2 = 250$ (menos de 3 días)

Un caso de test consiste en algún seteo de los n parámetros

El menor conjunto de casos de test se obtiene cuando cada par es cubierto sólo una vez.

Un caso de test puede cubrir hasta $\sum_{i=1}^{n} i = \frac{n*(n-1)}{2}$ pares.

En el mejor caso, cuando cada par es cubierto exactamente una vez, tendremos m^2 casos de test distintos que proveen una cobertura completa.

En el ejemplo $5^2 = 25 \Rightarrow$ hasta un mínimo de 2 hs 5 min

La generación del conjunto de casos de test más chico que provea cobertura completa de los pares no es trivial.

Existen algoritmos eficientes para generación de casos de test que pueden reducir el esfuerzo de testing considerablemente.

En un caso de 13 param. c/u con 3 valores, la cobertura de a pares puede resultar en 15 casos de test.

El testing de a pares es un enfoque práctico y muy utilizado en la industria.

Ejemplo:

Supongamos un producto de sw multiplataforma que usa browsers como interfaz y debe trabajar sobre distintos sistemas operativos:

Tenemos tres parámetros:

- SO (param A): Linux, MacOSX, Windows vista.
- Tamaño mem. (B): 512MB, 1GB, 2GB
- Browser (C): Firefox, Opera, Safari/Konqueror

Nro. total de combinaciones de a pares: 27
Cant. total de casos de test puede ser menor.

A	B	C	Pares
a_1	b_1	c_1	$(a_1, b_1)(a_1, c_1)(b_1, c_1)$
a_1	b_2	c_2	$(a_1, b_2)(a_1, c_2)(b_2, c_2)$
a_1	b_3	c_3	$(a_1, b_3)(a_1, c_3)(b_3, c_3)$
a_2	b_1	c_2	$(a_2, b_1)(a_2, c_2)(b_1, c_2)$
a_2	b_2	c_3	$(a_2, b_2)(a_2, c_3)(b_2, c_3)$
a_2	b_3	c_1	$(a_2, b_3)(a_2, c_1)(b_3, c_1)$
a_3	b_1	c_3	$(a_3, b_1)(a_3, c_3)(b_1, c_3)$
a_3	b_2	c_1	$(a_3, b_2)(a_3, c_1)(b_2, c_1)$
a_3	b_3	c_2	$(a_3, b_3)(a_3, c_2)(b_3, c_2)$

7.3.5. Casos especiales

Los programas usualmente fallan en casos especiales.

Estos dependen de la naturaleza de la entrada, tipos de estructuras de datos, etc.

No hay buenas reglas para identificarlos.

Una forma es adivinar cuando el sw puede fallar y crear esos casos de test.

También se denomina "**adivinanza del error**".

La idea es jugar al abogado del diablo y cuestionar los puntos débiles.

Usar la experiencia y el juicio para adivinar situaciones donde el programador pueda haber cometido errores.

Los casos especiales pueden ocurrir debido a suposiciones sobre la entrada, usuario, entorno operativo, negocio, etc.

Ej: Volviendo al programa que contaba las palabras:

archivo vacío, archivo inexistente, archivo que tiene solo blancos, o con una sola palabra, o múltiples blancos entre palabras, o líneas en blanco consecutivas, o blancos al comienzo, palabras ordenadas, blancos al final del archivo, etc.

El testing por adivinanza del error es quizás el más utilizado en la práctica.

7.3.6. Testing Basado en estados

Algunos sistemas no tienen estados: para las mismas entradas, se exhiben siempre las mismas salidas.

En muchos sistemas, el comportamiento depende del estado del sistema, i.e. para la misma entrada, el comportamiento podría diferir en distintas ocasiones

I.e. el comportamiento y la salida depende tanto de la entrada como del estado actual del sistema.

El estado del sistema representa el impacto acumulado de las entradas pasadas.

El testing basado en estado está dirigido a tales sistemas.

Un sistema puede modelarse como una máquina de estados.

El espacio de estados puede ser demasiado grande (es el producto cartesiano de todos los dominios de todas las vars).

El espacio de estados puede particionarse en pocos estados, cada uno representando un estado lógico de interés del sistema.

El modelo de estado se construye generalmente a partir de tales estados.

Un modelo de estados tiene cuatro componentes:

- Un conjunto de estados: son estados lógicos representando el impacto acumulativo del sistema.
- Un conjunto de transiciones: representa el cambio de estado en respuesta a algún evento de entrada.
- Un conjunto de eventos: son las entradas al sistema.
- Un conjunto de acciones: son las salidas producidas en respuesta a los eventos de acuerdo al estado actual.

El modelo de estado muestra la ocurrencia de las transiciones y las acciones que se realizan.

Usualmente el modelo de estado se construye a partir de las especificaciones o los requerimientos.

El desafío más importante es, a partir de la especificación de los requerimientos, identificar el conjunto de estados que captura las propiedades claves pero es lo suficientemente pequeño como para modelarlo.

Ejemplo: Una pila de tamaño acotado.

El interés está en los casos en que la pila está vacía, llena, o ni uno ni lo otro.

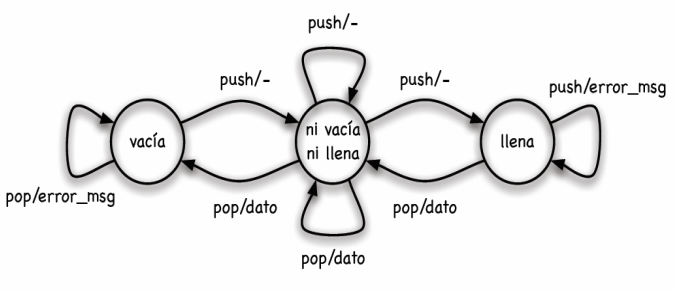

Figura 62: Estados lógicos de una Pila

El modelo de estado puede crearse a partir de la especificación o del diseño.

En el caso de los objetos, los modelos de estado se construyen usualmente durante el proceso del diseño.

Los casos de test se seleccionan con el modelo de estado y se utilizan posteriormente para testear la implementación.

Existen varios criterios para generar los casos de test. Ej: Cobertura de transiciones: el conjunto T de casos de test debe asegurar que toda transición sea ejecutada. Cobertura de par de transiciones: T debe ejecutar todo par de transiciones adyacentes que entran y salen de un estado. Cobertura de árbol de trans.: T debe ejecutar todos los caminos simples (del estado inicial al final o a uno visitado).

El test basado en estados se enfoca en el testing de estados y transiciones. Se testean distintos escenarios que de otra manera podrían pasarse por alto.

El modelo de estado se realiza usualmente luego de que la información de diseño se hace disponible.

En este sentido, se habla a veces de testing de **caja gris** (dado que no es de caja negra puro).

7.4. Test de Caja Blanca

El testing de caja negra se enfoca sólo en la funcionalidad:

Lo que el programa hace no lo que éste implementa.

El testing de caja blanca se enfoca en la implementación:

El objetivo es ejecutar las distintas estructuras del programa con el fin de descubrir errores. Los casos de test se derivan a partir del código.

Se denomina también testing estructural.

Existen varios criterios para seleccionar el conjunto

Tipos de testing estructural:

1. Criterio basado en el flujo de control
   - Observa la cobertura del grafo de flujo de control.
2. Criterio basado en el flujo de datos.
   - Observa la cobertura de la relación definición-uso en las variables.
3. Criterio basado en mutación.
   - Observa a diversos mutantes del programa original.

Nos vamos a enfocar en los dos primeros criterios (el tercero lo leen del libro!)

7.4.1. Criterios basados en control de Flujo

Considerar al programa como un grafo de flujo de control.

Los nodos representan bloques de código, i.e., conjuntos de sentencias que siempre se ejecutan juntas.

Una arista (i, j) representa una posible transferencia de control del nodo i al j.

Suponemos la existencia de un nodo inicial y un nodo final.

Un camino es una secuencia del nodo inicial al nodo final.

Criterio de cobertura de sentencia

Cada sentencia se ejecuta al menos una vez durante el testing.

I.e. el conjunto de caminos ejecutados durante el testing debe incluir todos los nodos.

Limitación: puede no requerir que una decisión evalúe a falso en un if si no hay else:

```
1 Abs(x){
2     if (x ≥ 0) x = −x ;
3 return(x);
4 }
```

El conjunto de casos de test x = 0 tiene el 100 % de cobertura pero el error pasa desapercibido.

No es posible garantizar 100 % de cobertura debido a que puede haber nodos inalcanzables.

Criterio de cobertura de ramificaciones

Cada arista debe ejecutarse al menos una vez en el testing.

I.e. cada decisión debe ejecutarse como verdadera y como falsa durante el testing.

La cobertura de ramificaciones implica cobertura de sentencias.

Si hay múltiples condiciones en una decisión luego no todas las condiciones se ejercitan como verdadera y falsa.

```
1 Check(x){
2 if (x ≥ 0) && (x ≤ 20) {
3     check = true;
4 }else{
5     check = false;
6 }
7 return(check)
8 }
```

El programa es incorrecto, pero pasa el conjunto de casos de test:

$\{x = 5, x = -5\}$ que satisface el criterio de bifurcación.

Todo test suite que cubre ramificaciones también cubre sentencias. Pero puede hacer tests suites que cubran ramificaciones que no encuentren el defecto mientras haya tests suites que cubren sentencias que si lo detecten.

- Cobertura de Sentencias: $\{x = -1\}$
- Cobertura de Ramificaciones: $\{x = 0, x = 1\}$

Criterio de cobertura de caminos

Todos los posibles caminos del estado inicial al final deben ser ejecutados.
Cobertura de caminos implica cobertura de bifurcación.
Problema: la cantidad de caminos puede ser infinita (considerar loops).
Notar además que puede haber caminos que no son realizables.
Existen criterios intermedios (entre el de caminos y el de bifurcación):
cobertura de predicados, basado en complejidad ciclomática, etc.
Ninguno es suficiente para detectar todos los tipos de defectos
(ej: se ejecutan todos los caminos pero no se detecta una div. por 0).

Proveen alguna idea cuantitativa de la "amplitud" del conjunto de casos de test.

Se utiliza más para evaluar el nivel de testing que para seleccionar los casos de test.

Se construye un grafo de definición-uso etiquetando apropiadamente el grafo de flujo de control.

Una sentencia en el grafo de flujo de control puede ser de tres tipos:

- def: representa la definición de una variable (i.e. cuando la var está a la izquierda de la asignación)
- uso-c: cuando la variable se usa para cómputo
- uso-p: cuando la variable se utiliza en un predicado para transferencia de control.

El grafo de def-uso se construye asociando vars a nodos y aristas del grafo de flujo de ctrl.

- Por cada nodo i, $def(i)$ es el conjunto de variables para el cual hay una definición en i.
- Por cada nodo i, $c - use(i)$ es el conjunto de variables para el cual hay un $uso - c$.
- Para una arista (i, j), $p - use(i, j)$ es el conjunto de vars. para el cual hay un $uso - p$.

Un camino de i a j se dice **libre de definiciones** con respecto a una variable x si no hay def de x en todos los nodos intermedios.

Criterios:

- Todas las definiciones: por cada nodo i y cada x en def(i) hay un camino libre de defs. con respecto a x hasta un uso-c o uso-p de x.
- Todos los usos-p: todos los usos-p de todas las definiciones deben testearse.
- Otros criterios: todos los usos-c, algunos usos-p, algunos usos-c.

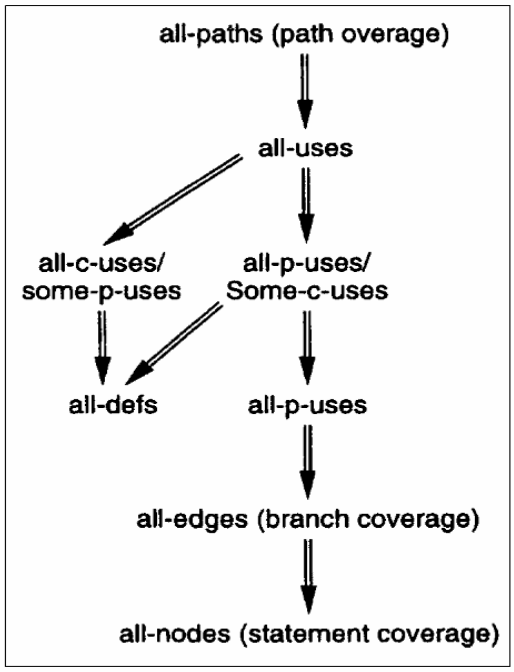

Figura 63:

```
1  scanf(x,y)
2  if (y < 0){
3       pow = 0  - y ;
4  }else{
5       pow = y;
6  }
7  z = 1.0;
8  while (pow ≠ 0){
9       z = z * x;
10      pow = pow - 1;
11 }
12
13 if (y < 0){
14  z = 1/z;
15 }
16 printf(z);
```

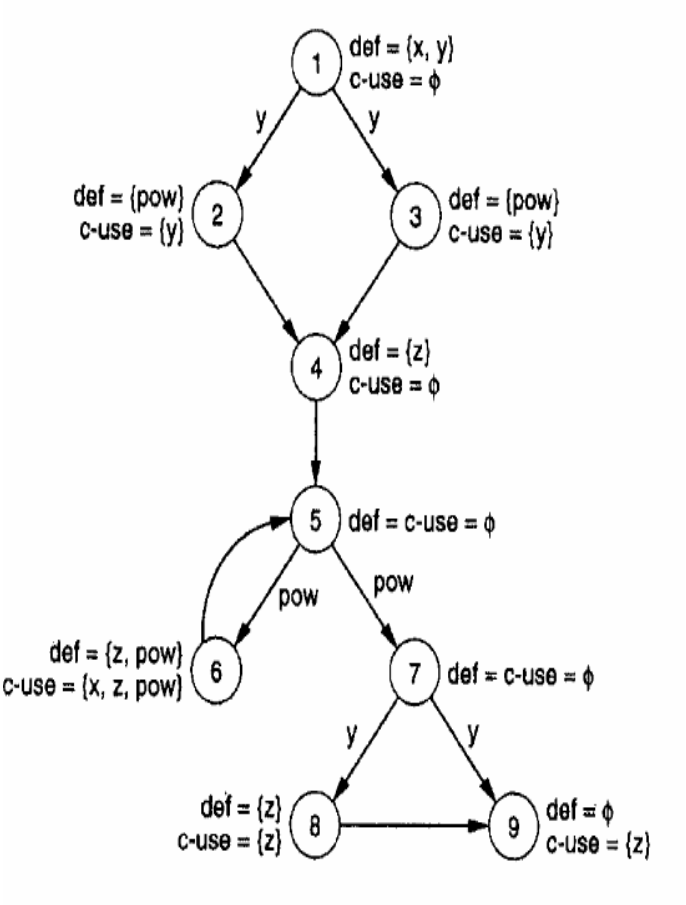

Figura 64: Gráfico de cobertura de caminos

1. Identificar el conjunto de caminos que satisface el criterio deseado
2. Identificar los casos de test que ejecutan esos caminos

- Cobertura de Ramificación:
- Caminos:
  - (1,2,4,5,6,5,7,8,9)
  - (1,3,4,5,6,5,7,9)
- Test
  - $\{x = 3, y = 1\}$

- $\{x = 3, y = -1\}$

- Cobertura de Todas las definiciones:
- Caminos:
  - (1,2,4,5,6,5,6,5,7,8,9)
  - (1,3,4,5,6,5,7,9)
- Test
  - $\{x = 3, y = 2\}$
  - $\{x = 3, y = -1\}$

7.4.2. Generación de casos de test y herramientas de Soporte

Una vez elegido el criterio surgen dos problemas:

1. ¿El test suite satisface el criterio?
2. ¿Cómo generar el test suite que asegure cobertura?

Para determinar cobertura se pueden usar herramientas que usualmente son de asistencia.

El problema de generación de test que cubra un criterio es habitualmente indecidible.

Las herramientas dicen qué sentencias o ramificaciones quedan sin cubrir.

El proceso de selección de casos de test es mayormente manual.

Comparación y uso

Se deben utilizar tanto test funcionales (caja negra) como estructurales (caja blanca).

Ambas técnicas son complementarias:

Caja blanca ⇒ bueno para detectar errores en la lógica del programa (i.e. errores estructurales del programa).

Caja negra ⇒ bueno para detectar errores de entrada/salida (i.e. errores funcionales).

Los métodos estructurales son útiles a bajo nivel solamente, donde el programa es "manejable" (ej. test de unidad).

Los métodos funcionales son útiles a alto nivel, donde se busca analizar el comportamiento funcional del sistema o partes de éste.

7.5. Métricas

7.5.1. Análisis de Cobertura

7.5.2. Confiabilidad

7.5.3. Defect Removal Efficiency

7.6. Resumen

Ejercicios