Ingeniería de Software I

Resumen Final

Grupo número

Integrante	LU	Correo electrónico
Avendaño, Santiago	113/06	santiavenda2@hotmail.com

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar/

Índice general

Capítulo 1

Ingeniería de Software

1.1. Introducción

1.1.1. Definición

La *Ingeniería de Software* es la aplicación de un acercamiento sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento de un software.

La Ingeniería de Software(IS) reúne conocimientos, herramientas y métodos para definir requerimientos de software y realizar diseño, construcción, testing y mantenimiento de software.

Objetivo:

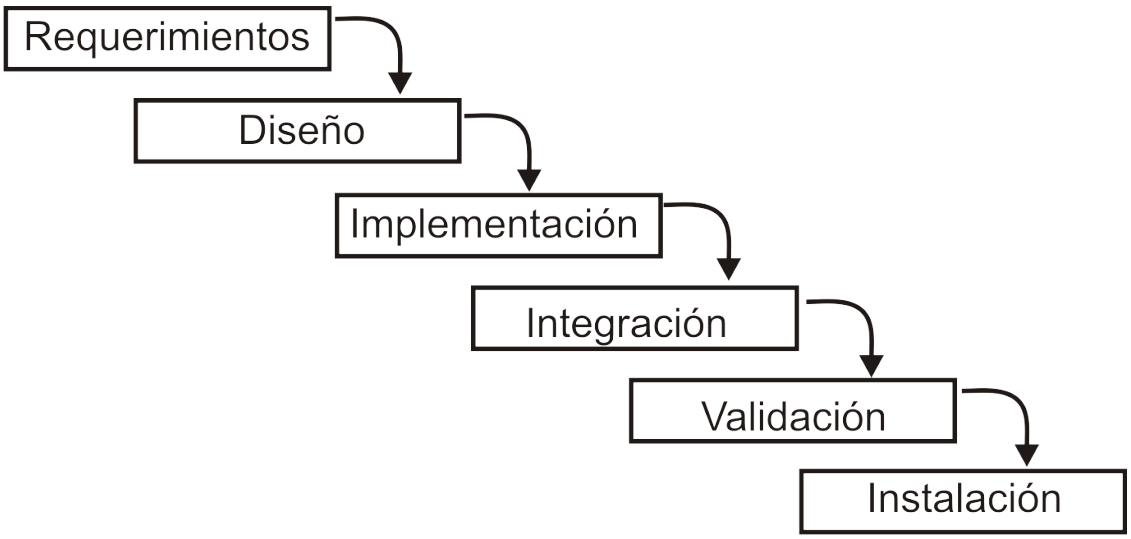
Se ocupa de construir un producto de software de buena calidad, lidiando con múltiples restricciones (tiempo, presupuesto, requerimientos, etc)

Problemáticas:

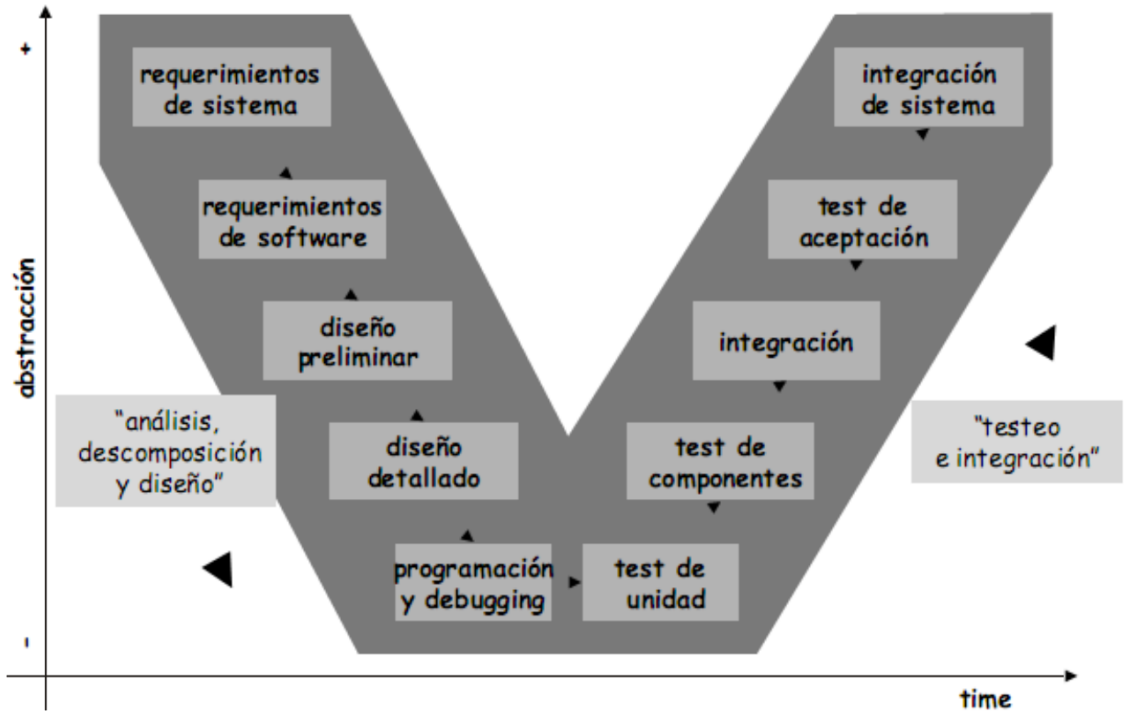
- Lidiar con la escala y complejidad de sistemas de software
- Identificar que significa buena calidad
- Disciplina joven, víctima de su propio éxito

1.2. Ciclo de Vida de Desarrollo de Software

1.2.1. Modelo Cascada (Royce, 1970)



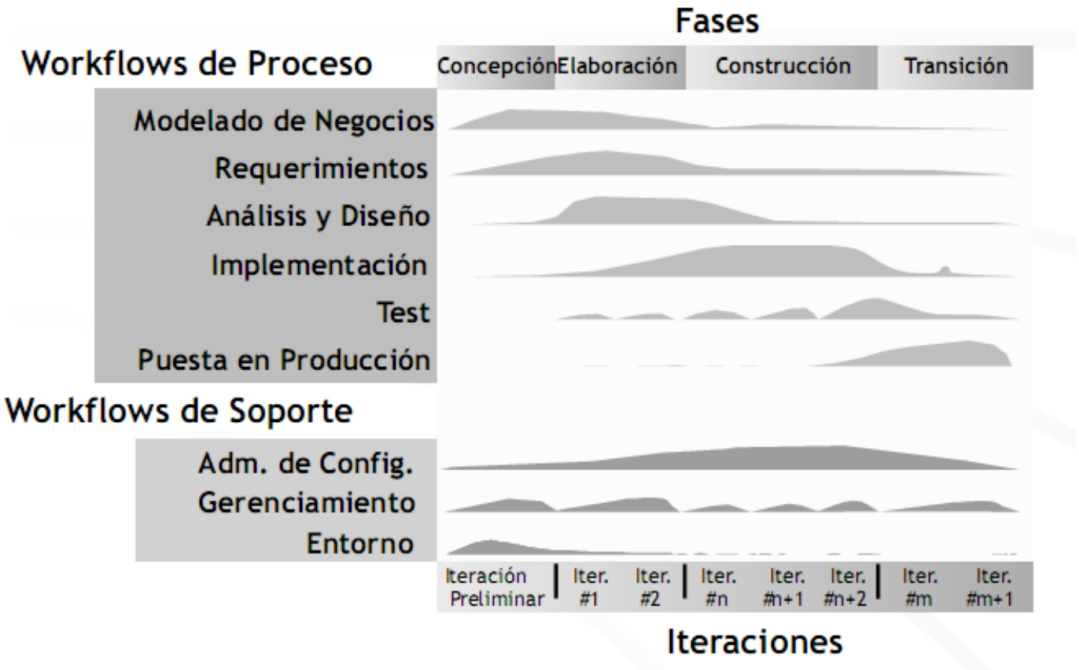
1.2.2. Modelo V



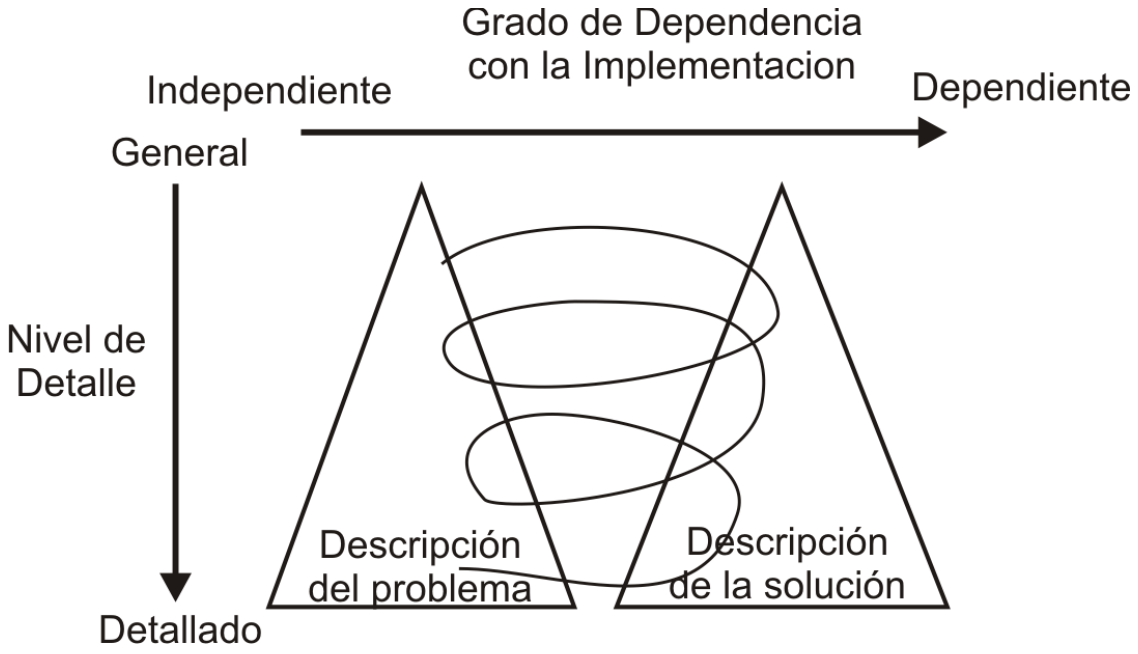
1.2.3. Modelo Espiral (Boehm,1988)

1.2.4. Unified SW Development Process (Jacobson, 1999)

División en fases e iteraciones



1.2.5. Modelo Twin Peaks



1.3. Modelos

Un *modelo* es una representación de la realidad.
Los modelos son construidos con el objetivo de lidiar con la complejidad de los sistemas.
Los modelos son efectivos porque:

- Describen un aspecto del problema a resolver
- Omiten detalles no relevantes al análisis para el cual se construye el modelo
- Permite comunicar en forma precisa aspectos del problema y la solución a otras personas.
- Son más baratos de construir
- Facilita el análisis (permite detectar errores y falencias tempranamente)

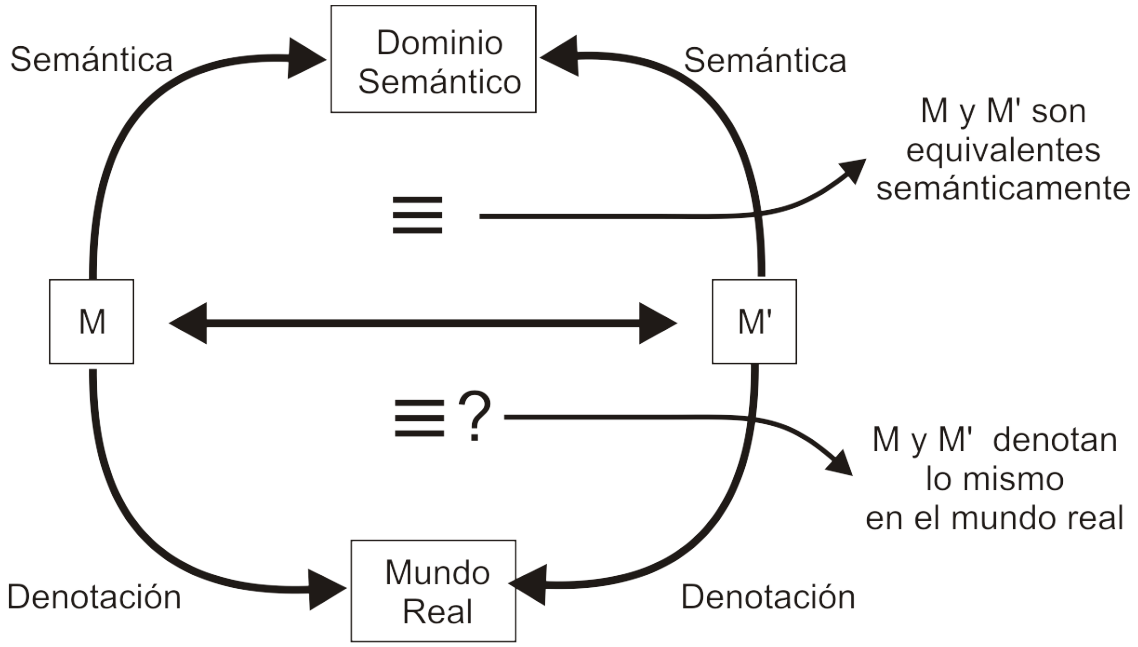
Los modelos tienen un Scope y un Span

Scope(aspecto): tipo de fenómeno que se capta

Span(segmento): individuos descriptos

Los modelos en Ingeniería de Software son lenguajes formales con una denotación precisa en el mundo real
Los lenguajes formales tienen:

- una *sintaxis*(gráfica/texto) para describirlos: cuáles son los garabatos permitidos
- una *semántica* para abstraer accidentes sintácticos: cuáles de estos garabatos significan lo mismo.



Un sistema se analiza desde múltiples modelos complementarios. Cada modelos enfoca una aspecto(scope) del sistema, para permitir análisis rigurosos y escalables.

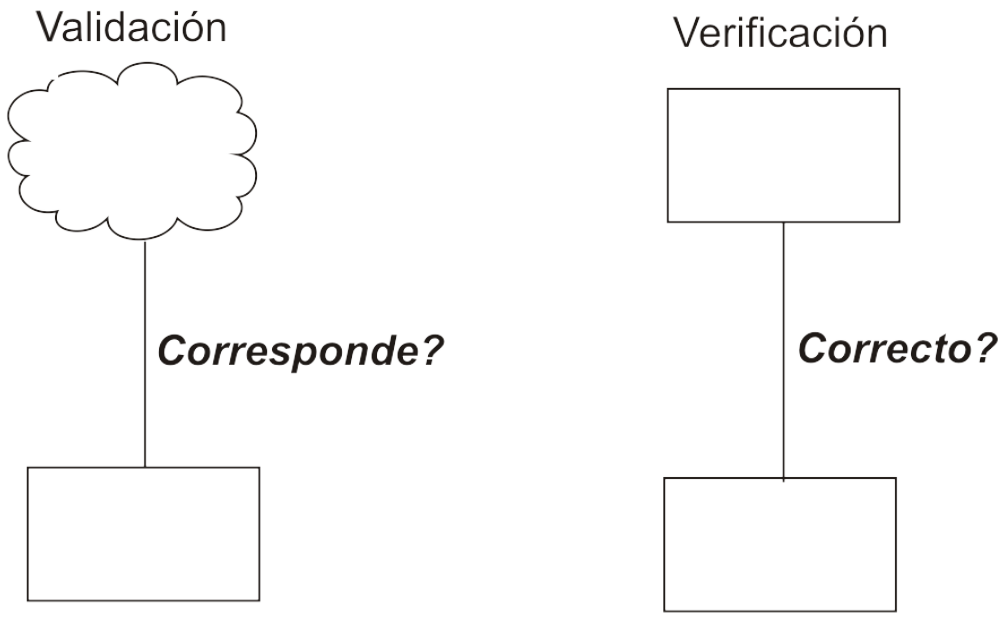
Es imposible vincular los distintos modelos formalmente, manteniéndolos analizables y entendibles(útiles).

Ejemplos de modelos: de Objetivos, de Agentes, de Operaciones, Conceptuales, de Comportamiento, Arquitectónicos.

1.4. Validación y Verificación

Validación: $\longrightarrow$ Mundo vs. Modelo Proceso cuyo objetivo es *incrementar la confianza* de que una descripción formal *se corresponde* con la realidad. Es imposible de realizar.

Verificacion: $\longrightarrow$ Modelo vs. Modelo Proceso cuyo objetivo es *garantizar* que una descripción formal *es correcta* con respecto a otra.



1.5. Resumen

- Que es la IS? Definición, objetivos, problemas
- Ciclo de vida de desarrollo de Software
- Modelos: definición, ventajas
  - Scope y Span
  - Sintaxis, Semántica y Denotación
  - Verificación y Validación

Capítulo 2

Ingeniería de Requerimientos

2.1. Introducción

2.1.1. Definición

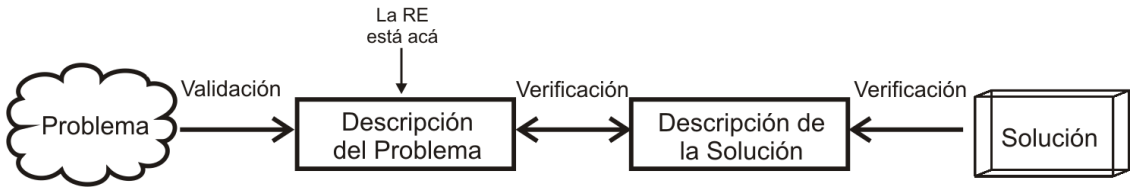
Todo Software se desarrolla con un *propósito*. Luego podemos definir *calidad* como el grado con que el software cumple con el propósito.

La ***Ingeniería de Requerimientos (RE)*** es un conjunto de actividades concernientes a identificar y comunicar el propósito de un sistema intensivo de software y el contexto en que será usado.

Tanto el problema como la solución son el resultado del comportamiento emergente de la interacción entre los componentes del sistema. Por eso hablamos de Ingeniería de Requerimientos de *Sistemas Intensivos en Software* (Software + Hardware + Entorno).

Por lo tanto la Ingeniería de Requerimientos actúa como puente entre las necesidades del mundo real de los usuarios, clientes y otros individuos afectados por el sistema de software, y las capacidades y oportunidades surgidas a partir de las tecnologías de software intensivo.

2.1.2. Separación del problema y la solución



La Ingeniería de Requerimientos requiere controlar que:

- El problema se corresponde con lar verdaderas necesidades (validación)
- La solución correctamente resuelve la descripción del problema (verificación)

2.1.3. Actividades de Ingeniería de Requerimientos

Elicitación: evocar una respuesta de alguien como reacción a preguntas o acciones.

Modelado: documentar rigurosamente para análisis (abstraer y estructurar lo elicitado)

Análisis: verificación inter e intra modelos

Validación: ¿los modelos se corresponden con la realidad?

Priorización: establecer criterios de preferencia con respecto a alcance y objetivos.

Negociación: unificar criterios entre los interesados

Especificación: documentación completa y detallada (documento final)

2.2. Modelo de Jackson

completar

2.3. Modelo de las 4 variables

completar

2.4. Diagrama de Contexto

completar

2.5. Modelo de Objetivos

completar

2.6. Modelo de Operaciones

completar

2.7. Modelo Conceptual

completar

2.8. Modelos de Comportamiento

completar

Capítulo 3

Diseño

3.1. Definición

Diseñar involucra estructurar la solución utilizando abstracciones y relaciones entre las abstracciones para:

- Documentar y comprender la propuesta de solución
- Razonar sobre su grado de adecuación a los requerimientos
- Comunicarla e implementarla
- Verificar/Validar el producto final
- Modificar/Adaptar la solución cuando cambien los requerimientos

A diferencia de requerimientos, en diseño:

- Se denotan conceptos del mundo de la solución y no del problema, pero incluye fenómenos de la interfaz mundo-máquina.
- Se describe propiedades localizadas (unidad, componente, módulo) y son de naturaleza operacional.

Objetivos de la etapa de diseño

- Descomponer el sistema en entidades de diseño "más chicas"
- Determinar la relación entre entidades
- Fijar mecanismos de interacción
- Especificar interfaces y funcionalidades de entidades
- Identificar oportunidades para el reuso
- Documentar lo anterior y los fundamentos de las elecciones

El diseño es un proceso iterativo, enfocado al *"que y cómo"* (en contraposición al *"que y porque"* de la etapa de relevamiento de Requerimientos).

3.2. Vistas

La descripción de un sistema complejo no es unidimensional. Es clave saber cuáles son las vistas relevantes y vinculadas.

Las vistas enfatizan aspectos e ignoran otros para que el problema sea abordable. Cada vista enfoca en una problemática en particular, permite responder cierta clase de preguntas y admite varios niveles de abstracción y técnicas de modelado.

Los diferentes tipos de vistas pueden clasificarse en

3.2.1. Vista Estática o Modular

Relacionada con el agrupamiento de código

Módulo: unidad de código que implementa un conjunto de responsabilidades (una clase, una colección de clases o cualquier descomposición de código). Es una unidad *design-time*.

Entidades: métodos, procedimientos, clases, librerías, DLLS, APIs, paquetes. módulos.

Relaciones: es-parte-de, depende-de, es-un.

Ejemplos: diagrama de clases, diagrama de paquetes.

La manera de modularizar suele determinar características como: modificabilidad, portabilidad y reuso.
Análisis que permite realizar:

- Trazabilidad de requerimientos
- Análisis de impacto
- Comunicación a otras personas
- Nivel de granularidad

No sirve para:

- realizar inferencias sobre comportamiento del sistema en su ejecución
- realizar análisis de performance.

3.2.2. Vista Dinámica o de Componentes y Conectores

Relacionada con las entidades *run-time*.

Entidades: procesos, threads, objetos, protocolos.

Las entidades *run-time* son instancias de tipos de conector o componente. Son entidades que consumen recursos y contribuyen al comportamiento en ejecución del sistema.

Relaciones: se-comunica-con, bloquea, contiene, crea, destruye.

Ejemplos: máquinas de estado, diagrama de secuencias y colaboración, diagrama de objetos, diagrama de componentes.
Análisis que permite realizar:

- Confiabilidad
- Performance: tiempo de respuesta, tiempo de latencia y volumen de procesamiento
- Recursos requeridos: almacenamiento y procesamiento
- Funcionalidad
- Protocolos
- Seguridad

3.2.3. Vista de Despliegue (Deployment)

Entidades: recursos y repositorios

Relaciones: procesos ejecuta sobre server, código de módulos almacenado en directorios, equipo de trabajo desarrolla paquete.

Ejemplos: Diagrama de Despliegue

3.3. Principios de Diseño

3.3.1. Descomposición

Divide and conquer: se parte cada parte del problema en subproblemas o componentes más pequeños siguiendo una estrategia adecuada (como ser pasos de ejecución, datos, tiempos, funcionalidades, etc), determinar las relaciones entre dichos componentes e iterar.

Es importante tener una estrategia de composición, no sólo de división (Divide to Conquer and reunite to rule)

3.3.2. Abstracción

Se basa en suprimir detalles de la implementación y posponer decisiones de diseño que ocurren a distintos niveles del análisis. El objetivo es simplificar el análisis, comprensión y justificación de decisiones.

La abstracción puede ser procedural (funciones, métodos, etc), de datos (TADs) o de control (loops, iteradores, multitasking).

3.3.3. Encapsulamiento

Se basa en mantener una clara separación entre interface e implementación. Con esto se logra no conocer ni usar más de lo que la interfaz promete.

Control Inversion Principle: Uso de frameworks, implementaciones parciales que el usuario debe rellenar para lograr la funcionalidad deseada; la diferencia principal con las librerías es que es el framework el que invoca el código del usuario y no al revés.

3.3.4. Information Hiding

Relacionado estrechamente con encapsulamiento.

Se base en esconder decisiones de diseño que pueden llegar a cambiar. Con esto se busca minimizar el impacto de cambios futuros. Para esto se programa orientado a interfaces.

Dependency Inversion Principle: Las dependencias se hacen hacia interfaces o clases abstractas, no hacia las implementaciones concretas.

3.3.5. Modularidad

Un módulo es una entidad estática que agrupa ciertas funcionalidades (superior a una clase). Tiene una interfaz bien separada de su implementación, garantiza alta cohesión y bajo acoplamiento.

Indicios de una buena modularización es tener una jerarquía de módulos lo suficientemente independientes, con responsabilidades claras y delimitadas, donde un cambio en uno impacta lo menos posible en el resto del sistema.

Cohesión: Es el grado de unión (cuán bien trabajan juntos) que tienen los distintos elementos de un módulo. Alta cohesión provee: robustez, confiabilidad, reusabilidad, comprensibilidad, testeabilidad y mantenibilidad.

Pueden definirse distintos tipos de cohesión, de peor a mejor, considerándose aceptables sólo los tres últimos:

- **Coincidental:** Ej. mis funciones de uso frequente, utils.lib
- **Lógico:** existe una categoría lógica que agrupa elementos aunque hagan cosas muy distintas (ej. todas las rutinas de I/O)
- **Temporal:** agrupadas por el momento en que se ejecutaran (ej. funciones que atajan un error de output, crean un error en un log y notifican al usuario)
- **Procedural:** agrupadas por pertenecer a una misma secuencia de ejecución o política (ej. funciones que chequean permisos y abren archivos)
- **Comunicacional:** agrupadas por operar sobre los mismo datos (ej. objetos, operaciones sobre clientes)
- **Secuencial:** agrupadas porque el output de uno es el input de otro
- **Funcional:** agrupadas porque contribuyen a una tarea bien definida del módulo

Single Responsibility Principle: A class should have only one reason to change; busca obtener un alto grado de cohesión, una clase debe tener una y solo una responsabilidad.

Acoplamiento: Grado de dependencia del módulo sobre otros módulos y en particular las decisiones de diseño que estos hacen. Generalmente proporcional al grado de cohesión.

Un alto acoplamiento conlleva una alta propagación de cambios y necesidades de testing, dificulta la comprensión de los módulos de forma aislada y trae problemas al reuso y retesteo.

Los tipos de acoplamiento son, de mayor a menor:

- **Contenido:** Cuando un módulo modifica o confía en el lo interno de otro
- **Común:** Cuando comparten datos comunes
- **Externo:** Cuando comparten aspectos impuestos externamente al diseño (ej. formato de datos, protocolo de comunicación, interfaz de dispositivo)
- **Control:** Cuando un módulo controla la lógica del otro
- **Estampillado(Stamp):** Cuando comparten una estructura de datos pero cada uno usa sólo una porción
- **Datos:** Módulos se comunican a través de datos en parámetros
- **Mensajes:** Módulos se comunican a través de mensajes, posiblemente no se conocen explícitamente

Interface Seggregation Principle: muchas interfaces para los diferentes clientes son mejores que una única interface de propósito general. Busca separar interfaces para minimizar dependencias y reducir el fanning.

3.3.6. Extensibilidad

El diseño debe ser capaz de acomodarse a los cambios de requerimientos sin sufrir modificaciones, siendo extendido con facilidad.

Open/closed Principle: la entidades de software deben ser abiertas para la extensión pero cerradas para la modificación be open. Suele implementarse mediante polimorfismo e interfaces.

Liskov Substitution Principle: las subclases deben ser sustituibles por sus clases bases.

Ley de Demeter: No hablar con extraños, se basa en que un método de un objeto sólo puede llamar métodos del propio objeto, sus parámetros o aquellos objetos que constituyen el objeto de manera directa o fueron creados por él. Se evita llamar métodos de objetos remotos retornados por otros métodos.

Facilita la mantenibilidad y adaptabilidad pero tiende a generar wrappers molestos y poco cohesivos.

3.4. Programación Orientada a Objetos(POO)

3.4.1. Definiciones

Objeto: entidad *run-time* que actúa como una unidad de encapsulamiento. Encapsula estado (vía variables internas) y comportamiento (vía métodos internos).

Los objetos encapsulan atributos/campos permitiendo acceso a ellos únicamente a través de los métodos.

Clase entidad *design-time*[1]. Representa una clasificaciçon de objetos. Define el comportamiento y atributos de un grupo de objetos con estructura y comportamiento similar (un objeto es una instancia de una clase).

La *instanciación* es el proceso de creación de un objeto a partir de la definición de una clase. Se realiza mediante el llamado a un constructor.

3.4.2. Relaciones entre clases

- **Dependencia:** define una relación *"usa"* o *"conoce a"*. Indica que el cambio de una clase puede afectar a la otra. Es la relación más débil.
- **Agregación:** indica una relación *"parte de"* o *"tiene un"*. Es un tipo especial de dependencia.
- **Composición:** es un tipo especial de agregación donde:
  - las partes no se comparten
  - la existencia de las partes depende de la existencia del contenedor.
- Generalización/Especialización (Herencia): denota la relación *"es un"*
  - Permite reuso y factorización de código
  - Permite polimorfismo por medio de subtipado
  - Es el mecanismo de extensión sugerido por medio de subtipado

3.4.3. Polimorfismo

Se refiere a tratar de manera uniforme estructuras que pueden tener más de un forma.

Conceptos relacionados:

- *Herencia:* es el proceso de derivar una clase especializada de una clase ya existente.
- *tipo real(actual):* es el tipo que tiene un objeto en run-time (`new relojDigital`)
- *tipo aparente:* es el tipo que deriva estáticamente del compilador (`Reloj r`)
  - *Regla: el tipo aparente debe ser supertipo del tipo real.*
- *Binding:* resolución del método invocado (si es Late-Binding se hace en tiempo de ejecución)
- *Upcasting* y *Downcasting*

[1] No es estrictamente verdad, depende del leguaje (Ej: en Smaltalk las clases son objetos)

3.4.4. Clase vs. Tipo

Un **tipo** T denota un conjunto de objetos que satisface un predicado asociado al tipo T.

Un **subtipo** S de T denota un subconjunto de elementos de T que satisfacen un predicado más restrictivo que asociado al tipo T.

Principio de Sustitución de Liskov/Wing *Cualquier propiedad que podamos probar sobre objetos del tipo/clase T, también debe poder probarse para objetos del subtipo S.*

Es una manera razonable de lograr subtipado sano en jerarquías de clases OO (algunos programadores lo encuentran muy restrictivo)

Significa que las subclases preservan el comportamiento de las superclases.

El requerimiento se cumple si pedimos **contravarianza** en los argumentos y **covarianza** en los resultados.

Poner grafico de Liskov

3.4.5. Clases Abstractas e Interfaces

Una **Clase Abstracta** es una clase que no puede ser instanciada. Agrupa una implementación común a un conjunto de clases. No dan una implementación concreta (métodos abstractos).

Una **interface**(Java) es un conjunto de métodos y constantes identificadas con un nombre. Los métodos de una interface no son implementados por ella. Las clases (abstractas o no) derivadas de una interface, deben implementar todos los métodos de la interface.

Las interfaces son utilizadas para separar (desacoplar) la especificación disponible al usuario de las implementaciones. Además se puede "simular" herencia múltiple haciendo que una clase implemente dos interfaces.

3.5. Patrones de Diseño

completar

Capítulo 4

Testing

4.1. Verificación

Recordemos algunos conceptos [1]

Validación: proceso cuyo objetivo es *incrementar la confianza* de que una descripción formal se *corresponde* con la realidad.

Verificación: proceso cuyo objetivo es *garantizar* que una descripción formal es *correcta* con respecto a otra.

La verificación plantea la pregunta: ¿Estamos haciendo el producto correctamente? (en relación con un componente anterior que describe nuestro producto)

Verificación Dinámica: consiste en ejecutar y observar el comportamiento de un producto.

- Testing
- Run time monitoring
- Run time verification

Verificación Estática: consiste en realizar un análisis de una representación estática del sistema para descubrir problemas.

- Inspecciones, Revisiones, Walkthrough (seguimiento)
- Análisis de Reglas Sintácticas sobre código
- Análisis Data Flow sobre código
- Model Checking
- Prueba de teoremas

4.2. Calidad de Software

Conceptos relacionados: Confiabilidad, Usabilidad, Corrección, Robustez, Facilidad de Mantenimiento, Seguridad, Reusabilidad datos, Verificabilidad + Claridad, Funcionalidad, Interoperabilidad.

La calidad del producto depende de tareas realizadas durante todo el proceso de desarrollo. Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos.

Definiciones

Falla: diferencia entre los resultados esperados y reales (manifestación de un defecto).

Defecto: está en el texto del programa, una especificación, un diseño, y desde allí se hace visible una falla.

Error: equivocación humana.

Un error lleva a uno o más defectos. Un defecto lleva a 0, 1 o más fallas.

[1] ver 1.4

4.3. Testing

Definición: verificación dinámica de la adecuación del sistema a los requerimientos (de distinto tipo). Es el proceso de ejecutar un producto para verificar que satisface los requerimientos e identificar diferencias entre el comportamiento real y el comportamiento esperado.

No prueba la correctitud del software!!!

El testing puede demostrar la presencia de errores, nunca su ausencia (Dijkstra)

El objetivo del proceso de Testing es encontrar fallas importantes del sistema
Se realiza ejecutando el programa y comparando resultados contra un oráculo (el mismo tester) [2].

4.3.1. Proceso de testing

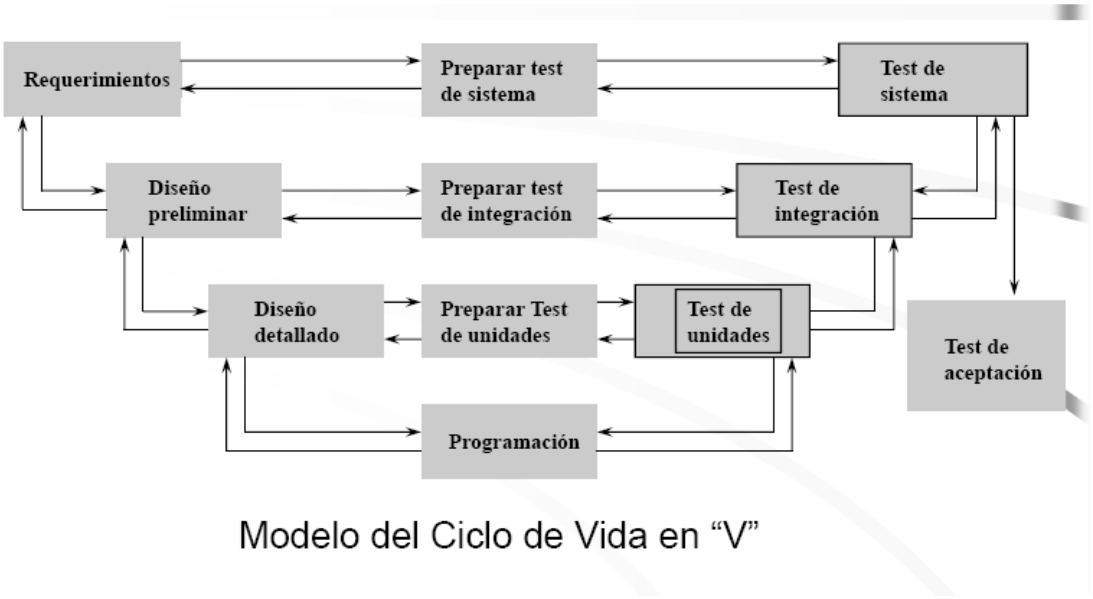
El testing debe ser un proceso paralelo al desarrollo, no una actividad al final.

Niveles de test

Test de Sistema: testea la funcionalidad del sistema completo con respecto a lo requerido por el cliente. Si el cliente participa se lo denomina **Test de aceptación**.

Test de Integración: test orientado a verificar que las partes de un sistema, que funcionan bien aisladamente, también lo hacen en conjunto

Test de Unidad: test realizado sobre una unidad de código pequeña y claramente definida (método, procedimiento, etc). Comúnmente es realizado por los desarrolladores.



Modelo del Ciclo de Vida en "V"

El **el test funcional o de sistema** se prepara inicialmente en función de los requerimientos. Luego este test dará lugar al **test aceptación**.

Sobre la base anterior se realiza un diseño preliminar que incluye al **test de integración**, y el diseño detallado el **test de unidades**. Una vez todo listo se comienza con la programación.

Asociado a la implementación de los tests, está la creación del ambiente, la ejecución de los casos, la documentación, el seguimiento, etc.

Actividades del proceso de Testing

Asociadas a arquitectura y diseño detallado

- Planificación de test de integración y unidad
- Generación de casos de test funcionales

[2] Se asume que se puede ejecutar el programa, que se conoce el resultado esperado y que el resultado esperado puede compararse con el resultado obtenido.

Asociadas a la implementación

- creación de ambiente de ejecución de tests
- generación de casos de test estructurales
- ejecución de test de unidad
- ejecución de test de integración
- documentación de resultados
- seguimiento de errores
- test de regresión

4.3.2. Terminología

Requerimientos de Test: qué quiero testear.

Especificaciones de Test: supuestos y definiciones que sirven para generar los casos de test para el requerimiento de test.

Casos de Test: descripciones de condiciones o situaciones a testear.

Dato de Test: asignación de valores concretos de parámetros para ejecutar un caso de test.

Test Suite: conjuntos de datos de test con los que se testea el programa.

Un test suite T es *exitoso* para un programa P si P es correcto para todo elemento de T.

4.3.3. Criterios de Selección de Casos de Test

Una de las mayores dificultades es encontrar un conjunto de tests adecuado:

- suficientemente grande para abarcar el dominio y maximizar la probabilidad de encontrar errores
- suficientemente pequeño para poder ejecutar el proceso de testing con cada elemento del conjunto y minimizar el costo del testing

La intuición nos dice que posiblemente haya inputs "parecidos", tal que testear el programa para uno de ellos equivaldría a testearlo para todos. Esto no es verdad pero sirve y es la base de la mayoría de las técnicas de test.

Un *criterio* es un subconjunto de conjuntos finitos del dominio de inputs del programa P (expresado con predicados)

Un conjunto de datos T satisface un criterio $C \Leftrightarrow T \in C$

Un criterio C es *Consistente* para P sii para todo par T_1 y T_2 de test sets que satisfacen C, T_1 es exitoso para $P \Leftrightarrow T_2$ lo es. Es decir, para los conjuntos de datos que considero equivalente, el programa devuelve el mismo resultado esperado.

Un criterio C es *Completo* para P sii si P es incorrecto entonces hay un test set T que satisface C, tal que T no es exitoso para P (i.e. si P tiene un bug, entonces algún caso de C lo encuentra).

Sin embargo estas nociones no son efectivas, es decir, no hay manera de evaluar si un determinado criterio es completo y/o consistente (mucho menos generar uno).

Como es imposible encontrar un criterio Completo y/o Consistente para un programa arbitrario(probaría corrección) se utilizan heurísticas para generarlos.

Los criterios pueden ser

Criterios de Caja Negra: se desentienden de la estructura interna del programa pero no de su especificación.

Criterios de Caja Blanca: los casos de test se definen a partir de la estructura interna del programa.

Es importante que los casos de test generados se testeen tanto condiciones válidas y esperadas, así como condiciones inválidas e inesperadas.

4.4. Testing Funcional

El testing funcional debe testear que un programa p implementa una funcionalidad f correctamente, es decir, que para todo elemento del dominio el resultado de p coincide con el de f.

Además debe avisar que la entrada no pertenece al dominio, y en caso de errores no destruir el sistema (o colgarse) sino simplemente notificar del error.

Se busca testear que el programa haga lo que debe y no haga lo que no debe.

4.4.1. Criterios de Testing Funcional

Con estas técnicas se busca encontrar encontrar casos que tengan una alta probabilidad de encontrar errores a un bajo costo. La idea es buscar casos generales, sistematizables y semi-automatizables.

Category Partition

Método:

1. Elegir una funcionalidad que pueda testearse en forma independiente.
2. Determinar sus parámetros u otros objetos del ambiente(*contexto*) que puedan afectar sus funcionamiento(*factores*).
3. Determinar las características relevantes de cada factor y de las relaciones entre factores(*categorías*).
4. Determinar las posibles elecciones (*choices*) para cada categoría.
5. Restringir casos con clasificaciones: *errores*, *únicos*, *restricciones* y *properties* (`if`).
6. Armar casos de test.

Conclusiones

- El método se puede aplicar a cualquier descripción de funcionalidad (formal, semiformal o informal).
- Es necesario usar los requerimientos para planificar el testing. Esto genera preguntas de incompletitud, ambigüedad, inconsistencia e incorrección en un momento en que es "fácil" corregir requerimientos.
- No hay una única solución: dependiendo como se realice el proceso se obtendrá un conjunto particular de casos de test.
- Los casos de test pueden conocerse incluso antes de la implementación

Notación Arbórea

Es un método similar a *category-partition* donde la combinación de factores se realiza mediante un árbol. Es útil cuando hay un orden de ingreso apara los parámetros.

Método

- Se arma un árbol donde cada nodo indica una categoría sobre un parámetro combinación entre un parámetro y otro anteriormente descripto en el árbol.
- Los ejes que salen son los choices compatibles con las condiciones acumuladas desde la raíz.
- Los casos de test especificados y ejecutados son los caminos del árbol.

Grafo Causa-Efecto

Permite definir combinaciones relevantes de categorías binarias sobre inputs para definir casos (diferencia con category-partition).

Se utiliza cuando hay mucha dependencia entre inputs y outputs. Ayuda a detectar ambigüedades e incompletitud en la especificación.

Método

Se busca generar todos los outputs admisibles con la siguiente heurística:

- Si hay un "or" entonces todas las opciones con sólo una señal en True.
- Si hay un "and" entonces todas las opciones con sólo una señal en False.

Esto reduce la cantidad de combinaciones de $O(2^n)$ a $O(n * k * o)$ donde:

- $n \rightarrow$ cantidad de categorías binarias sobre el input.
- $k \rightarrow$ profundidad del diagrama.
- $o \rightarrow$ cantidad de combinaciones de outputs válidas.

Arreglos ortogonales(OATS, Orthogonal Array Testing Strategy)

Se basa en *2-wise testing*, que sostiene que la mayoría de los errores se dan por combinaciones de uno o dos parámetros (errores simples o dobles). Se busca poder realizar tests más económicos sin caer en la explosión combinatoria de partición de dominios, testeando la interacción entre pares de factores.

Para esto se generan arreglos ortogonales de los factores tomados de a dos: se generan todas las posibles combinaciones de todos los pares de factores. En las tablas, se toman las filas como los casos de test y las columnas como los factores. Luego se unen todas las combinaciones obtenidas en una tabla de casos de test.

Se define *level* como la cantidad de valores (choices) posibles para un determinado factor. Los tests se tabulan como $L_{runs}(Levels^{Factors})$; por ejemplo, $L_{18}(3^6 6^1)$ implica que se usaron 18 corridas para testear 7 factores: uno de seis niveles y seis de tres niveles. Se define *strength* como la cantidad de columnas tales que las $Levels^{Strength}$ posibilidades aparecen la misma cantidad de veces.

Conclusiones

- Ninguna técnica es completa (cada una ataca diferentes problemas).
- Lo mejor es combinar varias de estas técnicas para complementar las ventajas de cada una.
- Depende del programa a testear.
- Sin especificación de requerimientos todo es más difícil.
- Hay que ser sistemático y documentar las suposiciones sobre el comportamiento o el modelo de fallas.

4.5. Testing Estructural de Unidades

Se representa el flujo de control de un programa con un grafo de flujo (*flowgraph*). Los flowgraph pueden representar programas secuenciales con un único punto de ingreso y un único punto de terminación.

Un camino en un flowgraph desde el nodo asociado al inicio del programa hasta el nodo final se llama *camino completo*. Una ejecución del programa que termina satisfactoriamente está asociada a un camino completo.

Un camino en un flowgraph para el cual no existe input del programa que fuerce su ejecución se dice *camino no factible*.

Cada camino factible puede tener muchos inputs que lo fuercen.

4.5.1. Criterios de Testing Estructural

Un criterio de testing estructural permite identificar entidades que deben cubrirse con los datos de test para satisfacer el criterio.

Basados en Flujo de Control

Cubrimiento de Sentencias o Instrucciones

Criterio: todas las sentencias del programa deben testearse (equivale a cubrir todos los nodos del flowgraph)

Método:

1. Con el código como base dibujamos el grafo de flujo de control.
2. Determinamos un conjunto de caminos que cumple el criterio.
3. Preparamos los datos de test que forzarán la ejecución de cada camino.
4. Evaluamos si satisfacemos el criterio.
5. Eventualmente iteramos.

Cubrimiento de decisiones Branch

El problema en 4.5.1 es que algunas decisiones se testean solo por True o por False.

Criterio: todas las decisiones en el control del programa deben ejercitarse al menos una vez por True y otra vez por False. Esto equivale a cubrir todos los arcos del flowgraph. $Branch \implies Sentencias$ (cubrir ejes de un árbol implica cubrir nodos)

Cubrimiento de Condiciones

El problema en 4.5.1 es que una decisión puede estar compuesta por varias condiciones (and, or) y no se cubren todas.

Criterio: todas las condiciones en el control del programa deben testearse al menos una vez por True y al menos una vez por False.

Branch no implica Condiciones y Condiciones no implica Branch (idem Sentencias)

Cubrimiento de Caminos

Todo camino del flujo de control del programa debe ejercitarse al menos una vez (equivale a cubrir todos los caminos del flowgraph).

Es poco factible ya que podrían ser muchos casos.

Basados en Flujo de Datos (Data-Flow Testing)

Def-Use flowgraph

Definiciones Una sentencia que guarda un valor en la posición de memoria de una variable, crea una *definición*. Una sentencia que trae el valor de la posición de memoria de una variable es un *uso* de la definición activa de esa variable.

- Un uso de x es un *usopredicado* o $p-uso$ si aparece en el predicado de una sentencia que representa una bifurcación de control
- En otro caso, se llama *usocomputacional* o $c-uso$ (aparece del lado derecho de una asignación)

El *def-use flowgraph* de un programa P y una variable X es un flowgraph de P donde cada definición o c-uso de X se asocia a un nodo, y cada p-uso de X a un arco.

Una *DUA (definition-use association)* es una terna $[d, u, x]$ tal que

- la variable x está definida en el nodo d
- la variable x se usa en el nodo u o en el arco u
- hay al menos un camino desde d hasta u que no contiene otra definición de x además de la de d (libre de definiciones para x)

Es decir, es una terna que vincula una definición de x con su uso inmediato.

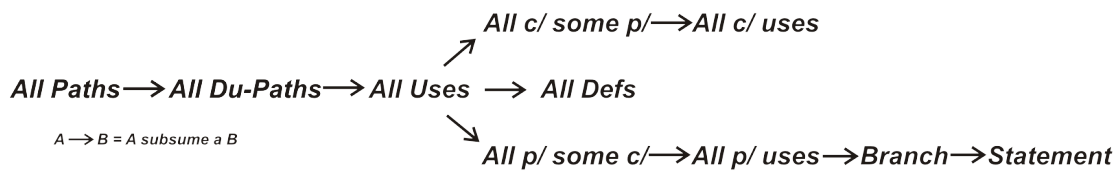
A partir de estos conceptos se generan distintos criterios:

- all defs
- all c-use
- all p-use
- all uses
- all c-use some p-uses
- all p-uses some c-uses
- all du-paths

Cubrimiento All-uses: para cada variable en el programa, deben ejercitarse toda las asociaciones entre cada definición y toda uso de la misma (tal que esa definición esté activa), equivale a cubrir todas las DUAS del programa.

4.5.2. Subsumption de criterios Estructurales

Un criterio A subsume a otro criterio B si un conjunto de datos de test T satisface un criterio A, entonces T satisface B. Por ejemplo, instrucciones subsume a branches, que subsume a caminos.



Los de más arriba tienen más posibilidades de encontrar fallas.

4.6. Test de requerimientos no funcionales

- Test de seguridad: validando disponibilidad, integridad y confidencialidad de datos y servicios
- Test de performance: validando los tiempos de acceso y respuesta del sistema
- Test de stress: validando el uso del sistema en sus límites de capacidad y verificando sus reacciones más allá de los mismos
- Test de Usabilidad

4.7. Ejecución del testing

a. Selección de datos

Se seleccionan y generan los datos que cumplen con los casos de tests diseñados para ejecutarlos.

b. Ambiente de test

El test de unidades se realiza por los programadores en el ambiente de desarrollo, mientras que los tests de aceptación, integración y sistema se realizan en un ambiente de testing separado del de desarrollo. Una vez aceptados, pueden pasar a producción, donde son reabsorbidos por desarollo.

c. Terminación del testing

- Se terminó el tiempo o recursos
- Se corrieron todos los tests derivados sin detectarse ningún error
- Porcentaje de cubrimiento de ciertas técnicas elegidas
- Fault-rate más bajo que un cierto valor especificado (# de errores por unidad de tiempo de testing)
- Se encontró un número predeterminado de errores (% del número total de errores estimado)

d. Documentación

Se deben documentar los casos de test, los criterios utilizados, los datos de prueba y criterios de terminación; también es necesario un reporte de ejecución luego de haberlos ejecutado que informe del ambiente y los resultados obtenidos.

e. Seguimiento

Debe realizarse un seguimiento de los errores desde que son detectados hasta que son finalmente aceptados, previo haber pasado por desarrollo para su corrección mediante debugging.

f. Test de Regresión

El test de regresión consiste en retestear un sistema luego de haber sido modificado para corregir un determinado elemento, adaptarse a un nuevo ambiente, mejorar prestaciones, etc. Los casos de regresión pueden ser:

- Reusables: testeando aspectos no modificados
- Restesteables: testeando aspectos de igual especificación pero distinta implementación
- Obsoletos: testeando una funcionalidad cuyo requerimiento fue modificado