Algoritmos y Programación III

María Inés Parnisari

27 de noviembre de 2012

Índice

Parte I
Resumen

1 Notación UML

Modelo: captura de una vista o perspectiva de un sistema, destinado a comunicar algo a un público determinado. Para un mismo sistema puede haber distintos modelos para distintas perspectivas. Es una abstracción. Se suelen representar mediante diagramas.

UML (*Unified Modeling Language*): lenguaje para la visualización, especificación y documentación de software. Define una notación que se expresa con diagramas. UML define 13 tipos de diagramas.

Usos de UML:

- Para discutir el diseño antes del código
- Para generar documentos que sirvan después de la construcción

1.1 Diagrama de estados

Diagrama de estados: modelo dinámico que muestra los cambios de estado que sufre un objeto a través del tiempo. La representación gráfica consiste en un grafo con nodos para los estados y arcos para las transiciones.

Los diagramas de estado son útiles para diseño, programación y pruebas, pero debe balancearse la información que incluyen para que sean útiles. Se utilizan para modelar:

- Comportamiento complejo de objetos
- Estados concurrentes
- Pruebas de caja blanca

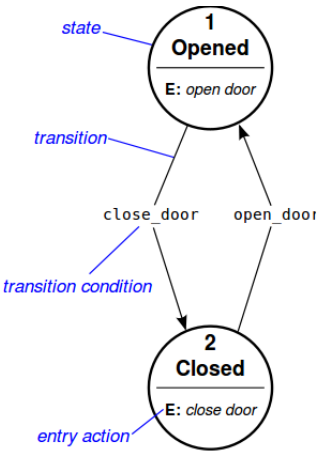


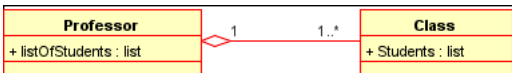
Figura 1: Diagrama de estados.

Para realizar un diagrama de estados:

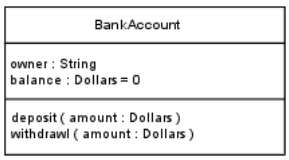
1. Decidir a qué eventos puede responder el objeto.
2. Definir precondiciones y postcondiciones de los estados inicial y final.
3. Definir los estados estables.
4. Probar el diagrama, haciendo un seguimiento manual de los eventos, transiciones y estados.

1.2 Diagrama de clases

Diagrama de clases: modelo estático del sistema a construir o de parte del mismo. En él se muestran las clases, sus relaciones y sus jerarquías.



(a) Agregación de clases



(b) Clase en notación UML

- Los atributos, métodos y propiedades precedidos por un "-" son privados, los precedidos por un "+" son públicos, y los precedidos por un "#" son protegidos (sólo pueden utilizarlos clases descendientes).
- Las líneas que vinculan clases se llaman asociaciones. Las asociaciones pueden tener cardinalidad, lo cual indica de cuántos atributos estamos hablando. Si la línea no tiene punta, entonces ambas clases se conocen entre sí; si no, la que no tiene la flecha conoce a la que sí.
- Una clase se representa con un rectángulo con tres divisiones: el nombre de la clase, los atributos, y los métodos.

- Los atributos y métodos de clase se pueden representar subrayados.
- La agregación se representa con un rombo vacío del lado de la clase contenedora.
- La composición se representa con un rombo lleno del lado de la clase contenedora.
- La herencia se representa con flechas vacías hacia las clases ancestros.
- Las clases y métodos abstractos van en *cursiva*.
- Las interfaces se heredan con líneas discontinuas, y se representan con el estereotipo `<<interfaz>>`.

Algunas cuestiones prácticas:

- No es necesario representar todas las clases con todos sus detalles.
- Conviene representar siempre clases, asociaciones y generalización.
- Conviene mantenerlos actualizados.
- No suelen mostrarse las relaciones de dependencia débil (ejemplo: excepciones).

1.3 Diagrama de secuencia

Diagrama de secuencia: modelo dinámico que muestra cómo interactúan un conjunto de objetos en un escenario determinado.

- Los objetos participantes se representan como rectángulos arriba de su línea de vida.
- Los mensajes son flechas entre objetos. El objeto que recibe la punta de la flecha es el que responde al mensaje.
- El orden de los mensajes está dado por el eje vertical, de arriba hacia abajo.
- La destrucción de un objeto se representa con una "X" en la línea de vida.
- Los mensajes asíncronos se representan con flechas de punta vacía.
- Conviene colocar los objetos más importantes a la izquierda del diagrama.

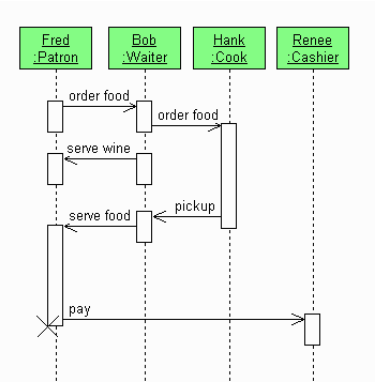


Figura 2: Diagrama de secuencia.

2 Programación orientada a objetos (POO)

Programación estructurada o procedural	*Programación orientada a objetos*
Abstracción de procesos	Abstracción de datos
Tipos por valor	Tipos por referencia
Funciones	Métodos dentro de clases
Diseñar la solución	Diseñar el modelo del dominio del problema

La **programación orientada a objetos** es un conjunto de disciplinas que desarrollan y modelizan software que facilitan la construcción de sistemas complejos a partir de componentes:

- Modulariza en base a las entidades del dominio del problema. Esto permite una representación más directa del modelo de mundo real en el código.
- Comparado con la programación estructurada, las reducciones de código van desde un 40 %.
- Suele trabajar con **tipos por referencia**. Una variable de este tipo no contiene directamente los datos almacenados, sino la dirección de memoria del lugar donde están esos datos.
- Se basa en un conjunto de objetos que se envían mensajes. Estos objetos son agregados de otros objetos.
- Los objetos deben tener una clase, y se crean en tiempo de ejecución.
- Todos los objetos de una misma clase admiten los mismos mensajes.
- Los lenguajes orientados a objetos (Java, C#, Smalltalk, etc.) suelen tener mecanismos de **recolección de basura**: se eliminan los objetos que no están siendo referenciados por nadie.

Los tres pilares de POO son:

1. Clases
2. Herencia
3. Polimorfismo

2.1 Objetos

Objeto: entidad capaz de almacenar estado y responder a comportamiento. Son instancias de clases, es decir, cada uno de los individuos que pertenecen a una determinada clase. Se crean en tiempo de ejecución y son referenciados por variables. Sus características son:

- *Estado*: conjunto de valores internos que representa la situación en la que está. Puede ser observable u oculto. Se lo almacena en variables internas dentro de los objetos - los **atributos**. Los cambios de estado que sufren los objetos se deben a estímulos o mensajes recibidos de otros objetos, cuyos efectos son eventos. Un **evento** es un estímulo que puede disparar una transición de estados. Un evento puede ser:
  - Asíncrono cuando el emisor de la señal puede seguir trabajando sin esperar una respuesta,
  - Síncrono cuando debe quedarse esperando la respuesta del receptor.

  Una **transición** entre estados es el paso de un estado a otro.
- *Comportamiento*: conjunto de respuestas que un método exhibe ante solicitudes de otros objetos. Describe los servicios que brinda.
- *Identidad*: es lo que diferencia a objetos de una misma clase. Se otorga una identidad al objeto en el momento de la creación del mismo, y se mantiene durante toda la vida del mismo. Es única para cada objeto, aunque tengan los mismos atributos.

Interfaz de un objeto: conjunto de métodos y propiedades aplicables a ese objeto.

Implementación de un objeto: definición de las operaciones y las estructuras de datos.

Mensaje: invocación de un método sobre un objeto. Para que un objeto le envíe un mensaje a otro, debería conocerlo. El objeto que envía la solicitud se llama **cliente**, y el que la recibe y responde a ella es el **servidor**.

Corolario 1. *Los objetos deben manejar su propio comportamiento; no deberíamos poder manipular su estado desde afuera.*

Propiedades: atributos conceptuales.

2.2 Clases

Clase: tipo de un objeto. Conjunto de objetos con la misma estructura y el mismo comportamiento. Molde desde el que se crean los objetos. Es un módulo fundamental de las aplicaciones orientadas a objetos. Si son definidas por el programador su implementación requiere de:

1. Determinación de las operaciones necesarias sobre el tipo.
2. Elección de una estructura de datos sobre el cual se materializa la implementación.
3. Implementación de la estructura de datos y las operaciones.

Precauciones:

- Cada clase debe tener un propósito simple y claro: "una clase por abstracción y una abstracción por clase".
- Los nombres de las clases deberían ser sustantivos y no verbos.
- Cuidado con las clases sin estado o sin comportamiento.
- Clase = interfaz + implementación.

Corolario 2. *Lo realmente importante es que la interfaz de una clase sea chica y sencilla.*

Paquete: agrupamiento de clases.

2.2.1 Métodos

Constructor: método de clase que, al ser invocado, crea un objeto en memoria. Se los utiliza para inicializar los valores de los atributos y dejar al objeto que está siendo creado en un estado válido.
Uso de los métodos y propiedades en POO:

- Definir y obtener comportamiento del objeto actual.
- Consultar y cambiar el estado del objeto actual.
- Devolver un valor calculado a partir del objeto actual.

Cuestiones a tener en cuenta al diseñar métodos de una clase:

- Evitar código duplicado.
- Ocultar el manejo de punteros dentro de un método (si el lenguaje maneja punteros).
- Utilizar nombres descriptivos, sin que nos asuste la longitud.
- Si se define una operación, debería existir la operación inversa.
- Si existen relaciones de orden entre los objetos, debería haber métodos que los comparan.
- En lo posible, los parámetros de una función deben ser pocos.
- Tratar de que la cantidad de métodos de una clase sea lo más chico posible.

Acoplamiento: interdependencia entre módulos.

Corolario 3. *El bajo acoplamiento garantiza que los módulos de un sistema sean relativamente independientes.*

Cohesión: necesidad de que un módulo haga o se refiera a una sola cosa simple.

Corolario 4. *Los módulos de un sistema exhiben alta cohesión cuando cada entidad y proceso está representado por un módulo aislado.*

Situaciones que afectan negativamente al acoplamiento y a la cohesión:

1. *"Envidia de características": Surge cuando diseñamos una clase que llama en gran parte a métodos y propiedades de otras clases.*

2. *"Cambios divergentes": Una clase se cambia por más de una razón de cambio funcional.*

MARTIN FOWLER

Calificadores de métodos:

- *Abstract*: nunca deben ser invocados, no tienen implementación. Deben ser redefinidos en algún nivel inferior de la jerarquía. Deben ser virtuales.
- *Virtual*: son vinculados dinámicamente o tardíamente. Agregan ineficiencias pero garantizan la reutilización de código.
- *Override*: redefinen los métodos virtuales de las clases ancestros.
- *Sealed*: no se pueden heredar.
- *Static*: método de clase.

Sobrecarga: una clase admite un método con distintas firmas.

Algoritmo 1 Sobrecarga

```
public String toString (Double d) {
return Double.toString(d);
}
public String toString(Integer i) {
return Integer.toString(i);
}
```

Redefinición: una clase modifica un método de su clase base.

2.3 RTTI (Run-Time Type Information)

RTTI: sistema que proporciona información de tipos en tiempo de ejecución. La clase debe conocerse en tiempo de compilación. El compilador abre y examina el archivo .class en tiempo de compilación.

Desventajas:

- Compromete la extensibilidad
- Evita el uso de polimorfismo

2.4 Reflexión

Reflexión: sistema que proporciona información de tipos en tiempo de ejecución, aun cuando ésta no fuera conocida en tiempo de compilación. El archivo .class se abre y examina en tiempo de ejecución.

Algoritmo 2 Reflexión

```
// Sin reflexion
new Foo().hello();
// Con reflexion
Class<?> cls = Class.forName("Foo");
cls.getMethod("hello").invoke(cls.newInstance());
```

El código con reflexión tiende a ser más lento que el que no lo tiene.

Ejemplos de uso:

- Herramientas que construyen diagramas UML a partir de código compilado.

3 Delegación, agregación y composición

Reutilización: uso de clases u objetos, desarrollados y probados en un determinado contexto, para incorporar esa funcionalidad en una aplicación diferente a la de origen.

Extensión: aprovechar las clases desarrolladas para una aplicación, utilizándolas en la construcción de nuevas clases para la misma u otra aplicación. La delegación y la herencia son herramientas claves para lograrla. La extensión es una forma de reutilización.

Delegación: mecanismo de reutilización por el cual una clase implementa parte de su funcionalidad delegando la misma en otra clase. El mecanismo se materializa cada vez que un método delega parte de su implementación en un método de otra clase.

Algoritmo 3 Delegación

```
public class Segmento {
int p1;
int p2;
public int longitud() {
return p1.distanciaA(p2);
}
```

- *Agregación*: hay una relación de contenido a contenedor, o del tipo "tiene" o "agrupa". Ejemplo: una inmobiliaria es una agregación de inmuebles.
- *Composición*: hay una relación de contenido a contenedor, del tipo "es parte de". El contenedor es responsable del ciclo de vida de la parte, y la destrucción del contenedor implica la desaparición de sus partes constitutivas. Ejemplo: no tiene sentido un domicilio si no existe un inmueble.

Corolario 5. *Se usa delegación cuando se necesitan algunos aspectos de la clase contenida, pero no su comportamiento completo. Se reutiliza sin mantener la interfaz.*

4 Herencia

Herencia: mecanismo por el cual se define un caso particular de una clase, agregándole métodos, propiedades y atributos.

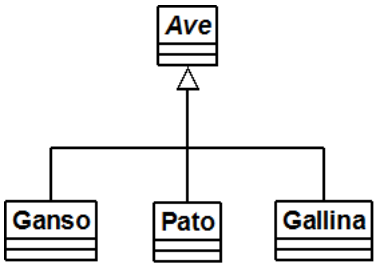


Figura 3: Herencia.

- Representa relaciones del tipo "es un".
- Cada clase puede tener una clase **ancestro** (base o superclase) y clases **descendientes** (derivadas o subclases) que heredan atributos y operaciones de su clase ancestro. Las subclases pueden redefinir métodos o agregar nuevos, pero no pueden eliminar métodos o atributos de sus superclases.
- Responde al mecanismo de especialización y generalización.
- Las clasificaciones no tienen que ser completas, pero sí excluyentes.
- Algunas clases pueden no tener instancias: son **clases abstractas**. Su finalidad es declarar atributos, métodos y propiedades comunes que luego se utilizarán en las clases descendientes.
- Las clases que no se pueden heredar se denominan **selladas** o finales.
- En muchos lenguajes existe una clase base por omisión. En Pharo, esa clase es `ProtoObject`.
- Los constructores no se heredan: cada clase tiene el suyo. Conviene llamar al inicializador de la clase ancestro al comienzo del inicializador propio.

Corolario 6. *Se usa herencia cuando se quiere tener la misma interfaz de la clase ancestro.*

Herencia múltiple: admitimos más de una clase ancestro por cada descendiente. Es una herramienta que debe utilizarse con cuidado. No todos los lenguajes la admiten.

Redefinición: mecanismo de volver a definir un método o propiedad en una clase especializada. Se utiliza cuando el comportamiento de la clase ancestro no debería ser exactamente igual para la descendiente.

- Es obligatoria cuando el comportamiento de una clase descendiente debe ser distinto al de la clase ancestro.
- Es opcional cuando, por razones de eficiencia o claridad, queremos modificar la implementación del método del ancestro.
- Debe preservar la semántica de la definición del método del ancestro (misma cantidad y tipo de parámetros, mismo tipo de valor devuelto).
- Se pueden redefinir métodos y propiedades, pero no atributos.

4.1 Mecanismos de abstracción

Abstracción: es uno de los medios más importantes mediante el cual nos enfrentamos a la complejidad inherente al software. Es la propiedad que permite representar las características esenciales de un objeto, sin preocuparse de las restantes características. Una abstracción se centra en la vista externa de un objeto.

- *Clasificación*: define un objeto como perteneciente a una clase. Ejemplo: Argentina es un país.
- *Agrupación*: define un objeto como una agregación de otros. Ejemplo: curso es un conjunto de alumnos y un profesor.
- *Composición*: define un objeto como compuesto por otros objetos; si el objeto compuesto muere los objetos componentes también. Ejemplo: empleados en una empresa.
- *Generalización*: define una clase como un caso general de varias clases. Ejemplo: los paralelogramos, los rectángulos y los rombos son cuadriláteros.
- *Especialización*: es la inversa de la generalización.

5 Polimorfismo

5.1 Basado en implementación

Polimorfismo: posibilidad que tienen los métodos y propiedades de mantener una respuesta unificada, con la misma semántica, aunque con distinta implementación, a través de la jerarquía de clases. Es decir: un mismo mensaje puede provocar la invocación de operaciones distintas.

El polimorfismo está asociado a la **vinculación tardía**: se retarda la decisión sobre el tipo de objeto hasta el momento en que vaya a ser utilizado el método o propiedad (es decir, en tiempo de ejecución). Los compiladores de los lenguajes orientados a objetos aseguran que la función existe y realizan verificaciones de tipos de los argumentos y del valor de retorno, pero no conocen el código exacto a ejecutar.

Corolario 7. *La esencia del polimorfismo es que, en vez de preguntarle a un objeto su estado e invocar uno u otro comportamiento según la respuesta, debemos invocar ese comportamiento y que el polimorfismo lo resuelva según el tipo.*

El polimorfismo se da en dos grados:

- *A nivel del objeto actual* (con herencia y redefinición): un método actúa, no sólo sobre objetos de la clase para la cual fue declarado, sino para cualquier clase descendiente de aquella.
- *A nivel del código del método* (con métodos virtuales y vinculación tardía): se demora hasta el tiempo de ejecución la decisión sobre qué métodos invocar dentro del código de otro método.

5.1.1 *Multiple Dispatch*

Es una característica de algunos lenguajes OO en la que una función o método se despacha dinámicamente dependiendo del tipo de su argumento.

Algoritmo 4 Double Dispatch

```
interface Collideable {
void collideWith(Collideable other);
}
class Asteroid implements Collideable {
public void collideWith(Collideable other) {
if (other instanceof Asteroid) {
// handle Asteroid-Asteroid collision
} else if (other instanceof Spaceship) {
// handle Asteroid-Spaceship collision
}
}
}
class Spaceship implements Collideable {
public void collideWith(Collideable other) {
if (other instanceof Asteroid) {
// handle Spaceship-Asteroid collision
} else if (other instanceof Spaceship) {
// handle Spaceship-Spaceship collision
}
}
}
```

5.2 Basado en interfaces

Interfaz: clase abstracta sin atributos, con todos sus métodos abstractos, que puede ser heredada (o implementada) por una clase que ya sea descendiente de otras clases.

- Sirven para encapsular un conjunto de métodos y propiedades, sin asignar esta funcionalidad a ningún objeto en particular ni expresar nada respecto del código que los va a implementar.
- Una interfaz "expone" métodos y propiedades, pero no restringe su uso a ninguna clase en particular.

Corolario 8. *Una clase implementa una interfaz cuando la clase se compromete a implementar los métodos de la interfaz.*

6 Genericidad y tipos como parámetros

Necesitamos poder escribir el mismo texto cuando podemos hacer la misma implementación de un concepto, aplicada a distintos tipos de objetos.

Genericidad: paradigma de desarrollo de software en donde clases y métodos admiten tipos como parámetros. Sus ventajas:

- Permite definir colecciones heterogéneas.
- Facilita la construcción de jerarquías de iteradores.
- Permite la reusabilidad.
- Permite el polimorfismo.
- Soportada por muchos lenguajes.

C++	*C#*	*Java*
"Templates"		

`template <T>`
`class xyz {`
`private: T a, b;`
`public: method (T p, T q) {`
`a = p; b = q}`
`}`
	`class List <T> {....}`	`public interface Iterator`
		`<T> {`
		`T next ();`
		`Boolean hasNext();`
		`}`

	Se implementa en la versión 5, mantiene compatibilidad con las versiones anteriores.	No mantiene compatibilidad con las versiones anteriores a la 2.
Durante la compilación se instancian tantas clases como tipos aparezcan; mecanismo de Search & Replace.	Durante la compilación del tipo genérico, las instrucciones de lenguaje generadas contienen espacios en blanco para los tipos. La información de genericidad no se elimina luego de tiempo de compilación.	Durante la compilación se genera un archivo de clase único. Se transforma el código fuente en bytecode, y sustituye `<T>` por `<Object>`.
	Durante la ejecución, si se referencia a un tipo genérico, se chequea si ya existe una instancia de ese tipo; si no, se la instancia.	Durante la ejecución se realizan casteos de tipos. No hay manera de conocer nada de los tipos genéricos.
`xyz<Foo>` no es subclase de `xyz<Bar>`		`xyz<Foo>` no es subclase de `xyz<Bar>`
	Pueden ponerse restricciones a los tipos: `class List <T>` `where T: Icomparable`	
	Los tipos genéricos pueden ser por referencia o primitivos.	Los tipos genéricos no pueden primitivos (pues no heredan de `Object`).
Son débilmente tipados.	Son fuertemente tipadas.	
	No tiene excepciones chequeadas.	Tiene excepciones chequeadas y no chequeadas.

7 Excepciones

Excepción: objeto, instancia de la clase `Exception`. Sirven para manejar situaciones en las que no tenemos suficiente información de contexto en un método para manejar un problema previsto, pero que no podemos dejar pasar.

- Las excepciones se "arrojan", "elevan" o "lanzan" ante situaciones inesperadas o inusuales, y son "atrapadas" o "capturadas" por el módulo que invocó al método que está corriendo.
- Si viene un dato incorrecto, el programa sólo se interrumpe si nadie se hace cargo de la excepción.
- Surgieron como solución al problema de los mensajes de error poco amigables y que cortaban la ejecución de un programa.
- Se provocan cuando no se cumple una precondición, una postcondición o un invariante.
- La mayoría de las clases de excepciones son vacías, sin atributos ni métodos.
- La jerarquía influye en la captura.

Cuestiones prácticas:

- Capturar las excepciones en el primer nivel que se pueda.
- Lanzar excepciones en constructores sólo para indicar que éste no terminó bien.
- Prever un mecanismo de registro de excepciones para permitir análisis posteriores (*logs*).

Corolario 9. *Cuando todo falle, lance una excepción.*

Hay dos tipos de excepciones:

1. *De negocio*: provocadas por problemas de concepción de la aplicación (ejemplo: división por cero).
2. *Técnicas*: provocadas por problemas imprevisibles durante la ejecución del programa (ejemplo: interrupción de la conexión a Internet).

Hay cinco formas de tratar excepciones:

1. Finalización súbita: cuando estamos seguros de que el problema hace fallar a toda la actividad; cuando los objetos implicados no van a ser utilizados nunca más; cuando la pretensión de seguir trabajando puede empeorar las cosas.
2. Continuación ignorando las fallas: cuando la falla no tiene consecuencias relevantes.
3. Vuelta atrás: cuando el estado del objeto puede ser modificado a su estado antes de la falla.
4. Avance y recuperación: cuando no es factible la vuelta atrás. Se sigue adelante, tratando de restablecer el estado de los objetos implicados.
5. Nuevo intento: cuando se espera que la falla sea transitoria o reversible. Se vuelve a probar luego de que se restablecieron los estados de los objetos. Se puede utilizar la técnica de retardo exponencial, o abandonar luego de un número de intentos o tiempo transcurrido.
6. No resolverla y enviar la misma u otra excepción al contexto invocante.

Manejador de excepciones: porción de código que decide qué hacer con la excepción que recibe.

Algoritmo 5 Manejador de excepciones

```
try {
// codigo que puede provocar excepciones
}
catch (ClaseExcepcion1 e1) {
// codigo que maneja las excepciones de un tipo 1
}
catch (ClaseExcepcion2 e2) {
// codigo que maneja las excepciones de un tipo 2
}
finally {
// codigo que se ejecuta siempre, haya habido excepciones o no
// sirve para liberar recursos
}
```

Algoritmo 6 Lanzador de excepciones

```
public int division (int a, int b) throws ZeroDivisionException {
if (b == 0)
throw new ZeroDivisionException;
else
return (a / b);
}
```

Ventajas de trabajar con excepciones:

- Hace a los programas más robustos y seguros.
- Le da más cohesión a los métodos.
- En proyectos grandes, hace al código más legible.

Corolario 10. *Un módulo nunca debe dar la impresión de que no pasó nada cuando algo falló.*

7.1 Excepciones Chequeadas

Son un tipo especial de excepciones que necesitan que la firma del método que las lanza contenga a una referencia a ella. Ejemplo: `public int dividir(int a, int b) throws ZeroDivisionException {...}`

Ventajas	*Desventajas*
Son seguras.	▪ Molesta tener que capturarlas siempre. ▪ Limita la redefinición al no poder agregar nuevas excepciones. ▪ No tienen sentido en lenguajes interpretados.

8 Disciplinas y metodología

8.1 Disciplinas operativas

1. **Captura de requisitos**: qué quiere el cliente (generalmente no está claro). Aquí trabajan los "analistas funcionales" (deben decirle al usuario qué es factible de realizar y qué no).

   a) Diagnóstico de la situación y de la necesidad

      1) Requisitos funcionales: procesos que debe hacer el sistema
      2) Requisitos operativos: escalabilidad, seguridad, facilidad de mantenimiento, desempeño, etc.

   b) Comprensión del problema

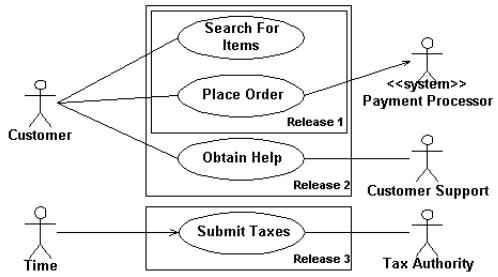


Figura 4: Caso de uso

      1) "Casos de uso": diagrama que muestra la interacción entre uno o más usuarios y el sistema. Los actores o roles se representan como personas, los casos de uso como elipses, y las comunicaciones con líneas.

   c) Decisión de qué tiene que hacer el sistema

2. **Análisis**: qué se debe desarrollar, qué vamos a construir. Es independiente de la tecnología. Aquí trabajan los "analistas funcionales".

   a) Estructura de los objetos, cómo se relacionan, cómo se organizan, jerarquías

   b) Comportamiento de los objetos: estados posibles, transiciones y eventos

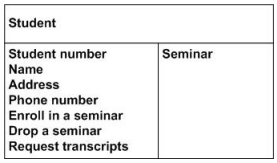


Figura 5: Tarjeta CRC

      1) "Tarjetas CRC": herramienta para la búsqueda de clases de análisis y su comportamiento. Muestra a cada objeto con sus responsabilidades y colaboradores. Deben ser papeles lo más chicos posibles.

3. **Diseño**: cómo vamos a realizar lo que definimos en el análisis. Es dependiente de la tecnología. Aquí trabajan los "diseñadores".

   a) ¿Es necesario el diseño? Si, y más en los proyectos grandes.

   b) ¿Por qué se diseña? Para manejar la complejidad, aislar detalles de implementación, agrupar operaciones...

   c) ¿Cuál es el resultado del diseño? Diagramas.

   d) ¿Qué se define en el diseño? Subsistemas o componentes de software, despliegue en hardware, integración con otros sistemas, interfaz de usuario, estructuras de datos y algoritmos, persistencia, definiciones tecnológicas...

   e) ¿Cuáles son los niveles de diseño?

1	Arquitectura macro
2	Arquitectura de paquetes
3	Diseño de clases
4	Diseño "micro" (atributos y métodos de las clases)

4. **Implementación**: construcción del producto.
5. **Pruebas**: determinar que el producto construido responde a los requisitos del cliente.
6. **Despliegue**: poner la aplicación en la computadora del cliente.
7. **Mantenimiento**: reparar, extender, mejorar el producto o adaptarlo a nuevos ambientes.

8.2 Métodos

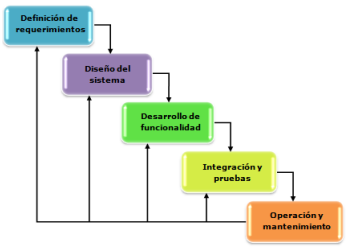


Figura 6: Método en cascada

Método en "cascada" (ver 16): método de desarrollo de software que toma como base el modelo de cadena de montaje de la industria manufacturera. Se basa en cumplir una serie de etapas, cada una separada de las otras, y sólo se empieza una tarea cuando terminó la anterior.

Desventajas:

- El proceso es lento
- Cuesta cambiar algo de una etapa ya terminada
- Las pruebas sólo vienen después de terminada la programación
- El usuario sólo ve el sistema luego de que está terminado
- La comunicación, mediante documentos, no siempre funciona bien
- No acepta cambios de requisitos frecuentes

Método en espiral: desarrollo incremental.

Ventajas:

- Podemos mostrar al usuario final versiones parciales del sistema
- Los programadores trabajan más tranquilos
- Permite cambios de requisitos

- Los errores aparecen antes
- Una persona puede participar en más de una fase

Desarrollo con prototipos: se construye un prototipo del sistema, siguiendo todas las etapas (incluso pruebas y entrega al usuario). Luego se prueba el sistema y se detectan errores, y se itera nuevamente, corrigiendo las fallas y se sigue adelante.

- **Prototipo**: versión preliminar del sistema, incompleta y en menor escala. Se deben desarrollar en poco tiempo, debe ser fácil de modificar, y el usuario debe poder probarlo.

Métodos formales y ágiles:

Formales	Ágiles (XP, Scrum...)
Atención a procesos y documentación	Atención a personas
Comunicación formal y escrita	Comunicación informal y cara a cara
Documentación de requisitos	Colaboración con el cliente
Seguimiento de planes	Agilidad en respuesta a los cambios de planes
Documentación completa	Software que funciona

Scrum: "marco" (no define el proceso ni los entregables) para métodos de desarrollo ágil, iterativo e incremental. Scrum estructura el desarrollo del producto en ciclos que se llaman *sprints*. Un *sprint* fija objetivos y el trabajo acordado debe estar terminado al finalizar el mismo. Los *sprints* son de longitud fija (1 a 4 semanas). Durante un *sprint* no se pueden cambiar ni los integrantes ni las tareas; lo único que se puede hacer es cancelar el *sprint*.

Hay 3 roles:

1. Dueño del producto: hace las veces del cliente. Elabora la lista de requisitos (*product backlog*) y les asigna prioridades. Define qué requisitos habrá que desarrollar en un *sprint* (*sprint backlog*).
2. Scrum Master: el líder del equipo.
3. Miembros del equipo: diseñan, codifican y prueban las funcionalidades.

Algunas características:

- Se realizan reuniones diarias de scrum para analizar el avance del día.
- Día a día se actualiza el *sprint burndown chart*, que muestra cuánto trabajo falta realizar dentro del sprint.
- Al finalizar el *sprint* se realiza una reunión para sacar conclusiones de lo aprendido.

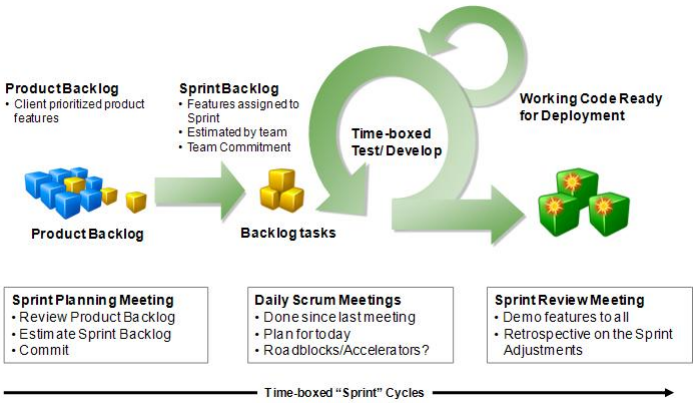


Figura 7: Scrum.

8.3 Disciplinas de soporte

1. Administración del proyecto
2. Gestión de cambios
3. Administración de la configuración
4. Gestión de RR.HH.
5. Gestión del ambiente de trabajo
6. Gestión de calidad

9 Diseño

Ocultamiento de implementación: técnica de desarrollo de software mediante la cual impedimos que el usuario del programa haga un uso indebido de los datos u algoritmos. Permitimos usar la interface de los tipos de datos pero no su implementación. Implica separar el *qué* hace el módulo del *cómo* lo hace.
Reglas del ocultamiento:

- Atributos: privados.
- Propiedades y métodos: públicos.

Las ventajas del ocultamiento son:

- Permite cambios de implementación.
- Impide violaciones de restricciones entre datos internos.

Encapsulamiento: conjunción de abstracción y ocultamiento de implementación.
Las ventajas del encapsulamiento son:

- Evita errores, como por ejemplo cuando invocamos un programa con un dato de otro tipo.
- Permite la división de un programa en módulos, es decir, permite aumentar la complejidad de los sistemas sin abrumar al cliente de la clase.

Refactorización (*refactoring*): cambio de un diseño una vez que se desarrolló una funcionalidad y la misma ya está codificada.

- Son riesgosas: estamos cambiando código que sabemos que funciona por otro que, aunque presumimos que será de mejor calidad, no sabemos si funcionará.
- Son beneficiosas: mejoran la legibilidad del código, eliminan código duplicado, mantienen alta la calidad del diseño.
- Conviene trabajar con pruebas unitarias automatizadas.

Modularidad: propiedad que permite subdividir una aplicación en partes más pequeñas, llamadas "módulos", cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las demás partes.
Granularidad fina:

9.1 Principios SOLID

- *Single Responsibility*: una clase debe tener una sola responsabilidad, para mantener alta la cohesión.
- *Open-Closed*: clases cerradas para modificación y abiertas para reutilización.
- *Liskov*: las clases base no deberían saber de la existencia de sus subclases para funcionar.
- *Interface Segregation*: los clientes no deberían depender de métodos que no utilizan.
- *Dependency Investment*: preferir clases abstractas o interfaces por sobre clases concretas.

  Corolario 11. *Programe contra interfaces, no contra implementaciones.*

9.2 Diseño con delegación, herencia e interfaces

- Si las clases comparten datos pero no comportamiento => Delegación.
- Si las clases comparten comportamiento pero no datos => Clase abstracta.
- Si en el ejemplo anterior, el comportamiento compartido es abstracto, debiendo implementarlo en las clases descendientes solamente => Interfaz.
- Si las clases comparten tanto datos como comportamiento => Herencia.

9.3 Diseño de clases por contrato

Programadores clientes: utilizan las clases que les proveen los **programadores proveedores**.

La clase que usa los servicios de otra es una clase cliente. A la que provee los servicios la llamamos servidora. Una clase u objeto pueden ser clientes y servidores, dependiendo del momento.

Contrato: define todo lo que un cliente debe conocer de su servidor y todo lo que debe cumplir el cliente si pretende que el proveedor lo sirva. Un contrato está formado por:

- Firma de métodos: constructores y parámetros.
- *Precondiciones*: condiciones bajo las cuales es válido llamar a una operación. Asegura que el cliente cumpla su parte del contrato. Se chequean en los métodos, y si no se cumplen se lanza una excepción.
- *Postcondiciones*: estado en que deben quedar los datos a la salida de la operación, suponiendo que a la entrada se cumplían las precondiciones. Asegura que el proveedor del servicio cumpla su parte al retornar de la operación que está implementando. Se chequean con tests unitarios.
- *Invariantes*: condiciones que deben cumplirse siempre para los datos del tipo.

9.4 Diseño de clases por TDD

Test-Driven Development: técnica de programación en la que se escriben las pruebas primero y luego los programas. No se escribe código hasta que las pruebas no fallan.

1. Test-First: se escribe el código de prueba antes del código productivo.
2. Automatización: las pruebas se corren luego de cada cambio al código productivo, y deben poder crecer en el tiempo.
3. Refactorización: mejora de calidad del diseño sin cambiar la funcionalidad.

Sus ventajas son:

- No hay una sugestión causada por haber escrito el código funcional en forma previa.
- Permite determinar la interfaz de un método o de toda una clase cuando ésta no está del todo clara.
- Suele llevar a pensar en nuevas capacidades o restricciones del uso de las clases.

9.5 Patrones de diseño

Patrón: solución a un problema en un determinado contexto. Indica cómo utilizar clases y objetos de formas ya conocidas. Es el concepto de reutilización llevado al diseño (no se reutiliza código, sino experiencia de otras personas).

Un patrón debe:

- Ser concreto
- Resolver problemas técnicos
- Utilizarse en situaciones frecuentes
- Favorecer la reutilización

9.5.1 Patrones de implementación directa

Suelen estar disponibles en los lenguajes en forma nativa.

Iterator	*Template Method*	*Observer*
Un objeto recorre colecciones sin ser parte de las mismas.	Una clase abstracta que contiene la lógica principal de un algoritmo, y clases descendientes que implementan cuestiones particulares.	Dos objetos desacoplados deben actuar al unísono (Observado y Observador). Paradigma de suscripción y notificación.
Implementado como interfaz.		Se utiliza una interfaz Observador.
		Maneras de notificar: - **Pull**: El observado avisa a los observadores que cambió sin más datos adicionales. - **Push**: El observado avisa a los observadores que cambió y proporciona dicha información. - Listas con diferentes temas para los observadores, y se notifican sólo los cambios importantes para cada lista.

9.5.2 Patrones de creación

Tienen como objetivo crear objetos de maneras no convencionales.

Factory Method	*Singleton*	*Null Object*
Encapsular la creación de objetos descendientes de una clase, o que implementen una interfaz.	Permitir una sola instancia para la clase, y dar un punto de acceso global a la misma.	Definir una clase que redefine los métodos que ameritan un comportamiento diferente en el caso de tener el atributo con una referencia nula.
	Crear un solo objeto y reutilizarlo, manteniendo el constructor privado.	

9.5.3 Patrones de cambio de comportamiento

Strategy	*Decorator*	*Command*	*State*	*Container*
Incluir una referencia a un tipo interfaz dentro de un objeto, que permite variar el comportamiento interno de un método de forma dinámica.	Agregar comportamiento a uno más métodos de una clase en forma dinámica.	Encapsular un método en un objeto sin estado.	Permitir que un objeto cambie su comportamiento cuando cambia su estado, de modo que parezca que cambia su clase.	Representar componentes y contenedores mediante una interfaz común.

9.5.4 Patrones estructurales

Adaptor	*Proxy*
Dar a una clase una interfaz diferente a la que tiene.	Un objeto Proxy aplica la misma interfaz de los objetos a los que representa.
El cliente se comunica con el adaptador.	

9.5.5 Patrones de diseño macro

MVC (Model-View-Controller)	*Separación en capas*
▪ **Modelo**: conjunto de clases correspondientes a la lógica de la aplicación. ▪ **Vista**: parte del sistema que administra la visualización y presentación de la información. Representa el estado del modelo. ▪ **Controlador**: responsable de manejar el comportamiento global de la aplicación. Recibe los eventos del usuario y decide qué hacer.	Cada módulo o **capa** depende de una capa inferior y brinda servicios a una o más capas superiores.
Ciclo MVC: 1. El controlador captura los eventos del usuario. 2. Interpreta las acciones y envía mensajes al modelo para generar cambios de estado. 3. El modelo actúa en consecuencia. 4. El controlador o el modelo envían una notificación a la vista informando que el modelo se actualizó. 5. La vista refleja dicho cambio.	
Mecanismos de notificación: 1. Consulta al estado y respuesta 2. Mediante eventos 3. Con observadores o vistas asociadas, conocidos por el modelo	

10 Depuración de programas

Pruebas automatizadas: pruebas que ejecuta la computadora en forma automática, en **frameworks** (programas que invocan nuestro código desde afuera).

- Garantiza la independencia del factor humano.
- Facilita correr las pruebas en forma conjunta, incluyendo todas las escritas hasta el momento.

Aserciones: expresiones que se incluyen para facilitar la depuración de los programas, y para documentar estados particulares de objetos. Son aseveraciones lógicas que deben ser verdaderas en un punto determinado del programa.

- Se utilizan para errores que nunca deberían ocurrir.
- Ejemplo de uso:

```
Debug.Assert (b == 0, "La variable b debe valer cero antes de comenzar");
```

Comprobación de estados: condiciones que deben cumplir los estados de ciertos objetos antes o después de realizar una operación. Hay dos maneras de enfocar el problema:

1. *Enfoque conservador*: "comprobar primero, actuar después". Sólo se puede aplicar a la verificación de precondiciones.

2. *Enfoque optimista*: "actuar primero, analizar después". A menudo se utilizan para verificar postcondiciones. La manera más común de manejar la dependencia de estados con este enfoque es mediante excepciones.

Pruebas: constan de tres partes.

1. *Arrange*: declaro las variables y creo los objetos.
2. *Act*: ejecuto el código que voy a probar.
3. *Assert*: verifico que el resultado obtenido sea correcto.

Parte II

Lecturas obligatorias

11 "*What's a Model For?*" (Martin Fowler)

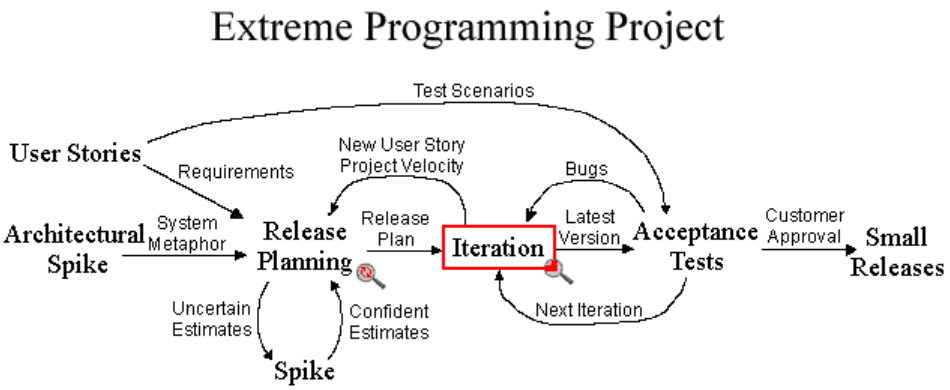
El desarrollo de UML es un importante avance. Sin embargo, a los clientes no les interesa: ellos solo quieren ver el software. Si el modelo mejora la calidad o reduce el costo del software, entonces tiene valor.

Modelar lleva tiempo. Existen herramientas que transforman texto en diagramas. Sin embargo, hay que ser cuidadoso con el nivel de detalle que se incluyen en los diagramas. Conviene usar un "modelo esquelético" que muestre los huesos del sistema. Este modelo es más fácil de mantener, porque los detalles importantes no cambian seguido. Pero este tipo de modelo hay que hacerlo uno mismo, pues las computadoras no saben distinguir los detalles que son importantes de los que no lo son.

Funciones de un modelo:

1. Dar mayor entendimiento del problema
2. Resaltar detalles importantes: modelo esquelético

12 *Extreme Programming*



Es un conjunto de prácticas ágiles de desarrollo.
Principios del XP:

1. *Test-Driven Design*: Primero escribir los programas de prueba y después la aplicación.
2. *Pair Programming*: programar de a dos. Uno escribe y el otro observa el trabajo.
3. *Continuous Integration*: ver *15.* Integraciones frecuentes y una máquina especial para las integraciones.
4. Pruebas unitarias, de integración y de aceptación.
5. Refactorizaciones al final.
6. Estándares de codificación precisos y estrictos.
7. Desarrollo incremental.
8. Comunicación frecuente con el cliente. Le mostramos el software y nos adaptamos a los cambios pedidos, y no a la inversa.
9. Implementar primero lo que tenga mayor valor para el cliente.
10. No escribir código por adelantado, ni código que no sabe si se usará. Se escribirá lo que se pida.
11. Compilaciones rápidas (10 minutos).
12. Documentación actualizada.

¿Cuándo usar XP?

- Cuando los requerimientos del cliente no son claros, cuando la funcionalidad del software pedido será cambiante.
- Cuando la fecha de entrega sea muy cercana, cuando haya mucho riesgo.
- Cuando el grupo de programadores sea pequeño (entre 2 y 12).
- Cuando haya comunicación constante entre todos (programadores y clientes).
- Cuando puedan realizarse pruebas.

13 "Documentación y Pruebas" (Pablo Suárez, Carlos Fontela)

Documentación: es muy importante.
Clasificada según las personas a las cuales está dirigida:

- *Para los desarrolladores*: comunican estructura y comportamiento del sistema y sus partes. Se deben documentar todas las fases de desarrollo. Los diagramas UML son muy útiles, siempre y cuando se mantengan actualizados. Al documentar debemos cuidar el nivel de detalle, y hacer diagramas que se concentren en un aspecto a la vez.
- *Para los usuarios*: todo aquello que necesita el usuario para la instalación, aprendizaje y uso del producto. Incluye guías de instalación, guías del usuario, manuales de referencia y guías de mensajes. Si bien es cierto que hay que hacer manuales, su existencia no nos libera de la responsabilidad de hacer productos amigables.
- *Para los administradores o soporte técnico (manual de operaciones)*: contiene toda la información sobre el sistema terminado, que describe errores posibles y los procedimientos de reparación.

Calidad: no se la puede agregar al software después de desarrollado. La calidad puede ser objeto de pruebas que determinan su grado.
Tiene que ver con:

- Ausencia de errores
- Usabilidad
- Costo
- Consistencia
- Confiabilidad
- Compatibilidad
- Utilidad
- Eficiencia
- Apego a los estándares

Error: comportamiento distinto del que espera un usuario razonable. No necesariamente un apego a los requisitos y un perfecto seguimiento de las etapas nos lleva a un producto sin errores. Puede haber errores de diseño y errores de implementación, e incluso errores en la etapa de pruebas y depuración.
Pruebas: determinan que un programa esté libre de errores. Suelen determinar la presencia de errores, pero nunca su ausencia. No tienen el objeto de prevenir errores sino de detectarlos. Se efectúan sobre el trabajo realizado. En lo posible, las pruebas deben ser realizadas por gente que disfrute de encontrar errores. En los sistemas triviales conviene generar casos de prueba para cubrir todas las posibles entradas y salidas, pero en los sistemas complejos conviene hacer pruebas sobre las partes en las que un error represente un mayor riesgo. Las pruebas son la mejor descripción de los requerimientos.

Corolario 12. *Pague por la prueba ahora, o pague por el mantenimiento después.*

Clasificadas según lo que se está controlando:

- *Pruebas centradas en la verificación*: para chequear que cumplimos con las expectativas del cliente.
- *Pruebas centradas en la validación*: para chequear que cumplimos con las especificaciones que realizó el analista.

Clasificadas según sobre qué actúan:

- *De caja blanca*: evalúan el contenido de los módulos.
- *De caja negra*: se trata a los módulos como cajas cerradas, sirven para verificar especificaciones.

Tipos de prueba (no son intercambiables):

- **Revisiones de código**

  - Pruebas de escritorio: rinden muy poco. Se usan para buscar anomalías típicas (variables u objetos no inicializados, bucles infinitos, etc.)
  - Recorridos de código: rinden mucho más. Son exposiciones del código escrito frente a pares. Se usan para encontrar errores.
  - Inspecciones de código: son reuniones en conjunto entre los programadores y testers. Se usan para chequear que el código escrito cumpla con las normas y para verificar la eficiencia.

- **Pruebas unitarias**: se realizan para controlar el funcionamiento de pequeñas porciones código. Se puede confeccionar un grafo de flujo con la lógica del código a probar. Así se determinan todos los caminos por los que el hilo de ejecución puede llegar a pasar. Luego de elaborar el grafo de flujo, se identifican los caminos independientes, y luego preparar los casos que fuercen la ejecución de estos caminos. Se procede a probar el código, documentando todas las salidas y dejando constancia de los errores.

  Un grafo de flujo está compuesto por:

  - Nodos: acciones del módulo. Los nodos predicados contienen condiciones, por lo que de ellos emergen varias aristas (`if`, `while`, `repeat`, `case`).
  - Aristas: flujo de control entre los nodos.

- **Pruebas de integración**: se toma a cada módulo como una caja negra. Tienen como base las pruebas unitarias. Consisten en una progresión ordenada de testeos para los cuales los módulos son ensamblados y probados hasta integrar el sistema completo. No es necesario terminar todas las pruebas unitarias para comenzar con las de integración.

  Hay dos tipos de integración:

  - Integración incremental: combinar el conjunto de módulos ya probados con los siguientes módulos a probar. Luego se va incrementando progresivamente el número de módulos unidos hasta formar el sistema completo.
    - Ascendente: se comienza integrando primero los módulos de más bajo nivel. Se eligen los módulos de bajo nivel y se escribe un módulo "impulsor".
    - Descendente: parte del módulo de mayor nivel para luego incorporar los módulos subordinados progresivamente. El módulo mayor hace de "impulsor" y se escriben módulos "ficticios" simulando a los subordinados.
  - Integración no incremental: se combinan todos los módulos de una vez.
  - Integración incremental sándwich: combina facetas de los métodos ascendente y descendente. Consiste en integrar una parte del sistema en forma ascendente y la restante en forma descendente, uniendo ambas partes en un punto intermedio.

- **Pruebas de sistema**: se realizan una vez integrados todos los componentes. Su objetivo es ver la respuesta del sistema en su conjunto frente a diversas situaciones. Se simulan situaciones y se prueba la eficacia y eficiencia de la respuesta obtenida.

  - Negativas
  - De recuperación
  - De rendimiento
  - De estrés
  - De seguridad
  - De instalación
  - De compatibilidad

- **Pruebas de aceptación**: se realizan una vez integrados todos los componentes. Están concebidas para que sea un usuario final quien detecte los posibles errores.

  - Pruebas *Alfa*: se realizan por un cliente en un entorno controlado. Primero se crea un ambiente con las mismas condiciones que se encontrarán en las instalaciones del cliente. Luego se realizan las pruebas y se documentan los resultados.

- Pruebas *Beta*: se realizan en las instalaciones propias de los clientes. La elección de los *beta testers* debe realizarse con cuidado. Cada usuario realiza sus propias pruebas, documenta errores, y realiza sugerencias.

Depuración: corrección de errores que sólo afectan a la programación (no provienen de fallas en el análisis o el diseño). Existen herramientas de depuración que facilitan esta tarea:

- Invariantes
- *Asserts*
- Herramientas de recorrido hacia atrás
- *Logs*

Pasos de la depuración:

1. Reproducir el error.
2. Diagnosticar la causa.
3. Corregirla (cuidado: a menudo se introducen nuevos errores).
4. Verificar la corrección.

14 "Principios de Diseño de Smalltalk" (Daniel Ingalls)

1. **Dominio personal**: si un sistema es para servir al espíritu creativo, debe ser entendible para un individuo.

   Cualquier barrera entre el sistema y el usuario es una barrera a la expresión creativa.

2. **Buen diseño**: un sistema debe ser construido con un mínimo conjunto de partes no modificables; esas partes deben ser genéricas, y todas las partes deben estar mantenidas en un esquema uniforme.

3. **Propósito del lenguaje**: proveer un esquema para la comunicación (implícita y explícita).

   Los modelos de computación deben ser compatibles con la mente.

4. **Alcance**: el diseño de un lenguaje debe tratar con modelos internos, medios externos, y con la interacción entre ellos tanto en el humano como en la computadora.

5. **Objetos**: un lenguaje debe soportar el concepto de "objeto" y proveer una manera de referirse a ellos.

   El administrador de almacenamiento de Smalltalk asocia un entero único a cada objeto del sistema. Cuando todas las referencias a un objeto desaparecen, el objeto se esfuma, y se recupera su espacio en memoria.

6. **Administrador de almacenamiento**: para ser orientado a objetos, un sistema debe proveer administración "automática" del almacenamiento.

7. **Mensajes**: la computación es una capacidad de los objetos que pueden ser invocados enviándoles mensajes.

   En Smalltalk: envía el nombre de la operación deseada, con los parámetros necesarios, entendiendo que el receptor sabe cómo realizar esa operación. Tenemos un universo de objetos que cortésmente se piden unos a otros realizar sus deseos.

8. **Metáfora uniforme**: un lenguaje debe ser diseñado alrededor de una metáfora que pueda ser aplicada uniformemente en todas las áreas.

   Ejemplo: Smalltalk está construido sobre el modelo de objetos que se comunican. Todo objeto en Smalltalk tiene un conjunto de mensajes, un protocolo, que define la comunicación a la que ese objeto puede responder.

9. **Modularidad**: ningún componente de un sistema debe depender de los detalles internos de otro componente.

   Al incrementarse la cantidad de componentes de un sistema, aumenta la probabilidad de interacciones no deseadas. Los lenguajes deberían minimizar la probabilidad de estas interdependencias. El acceso al estado interno de un objeto es a través de su interfaz de mensajes.

   La complejidad de un sistema puede ser reducido agrupando componentes similares: en Smalltalk esto se da a través de clases. Una clase describe otros objetos.

10. **Clasificación**: un lenguaje debe proveer un medio para clasificar objetos similares, y para agregar nuevas clases de objetos.

11. **Polimorfismo**: un programa sólo debería especificar el comportamiento esperado de los objetos, no su representación.

12. **Factorización**: cada componente independiente de un sistema sólo debería aparecer en un solo lugar.

    Una falla en la factorización implica una violación a la modularidad. Smalltalk promueve diseños bien factorizados a través de la herencia.

13. **Reaprovechamiento**: cuando un sistema está bien factorizado, un gran reaprovechamiento está disponible.

14. **Máquina virtual**: una especificación de máquina virtual establece un marco para la aplicación de tecnología.

    En Smalltalk: modelo orientado a objetos para almacenamiento, modelo orientado a mensajes para procesamiento, modelo de mapa de bits para despliegue visual de información.

15. **Principio reactivo**: cada componente accesible al usuario debe ser capaz de presentarse de manera entendible.

    La interfaz al usuario es un lenguaje visual. La estética y la flexibilidad son esenciales.

16. **Sistema operativo**: es una colección de cosas que no encajan dentro de un lenguaje. No debería existir.
17. **Selección natural**: los lenguajes y sistemas que son de buen diseño persistirán, sólo para ser reemplazados por otros mejores.

15 *"Continuous Integration"* (Martin Fowler)

Continuous Integration: práctica de desarrollo de software donde los miembros de un equipo integran su trabajo frecuentemente, generalmente a diario. Cada integración es verificada por una compilación y testeos automatizados para detectar errores de integración lo más rápido posible.

- Es muy útil utilizar servidores de integración, por ejemplo, CruiseControl.
- Un programa con errores no debe mantenerse así por mucho tiempo.

Mainline: rama del proyecto que está siendo desarrollado. Pueden existir ramas aparte, pero conviene que no sean muchas.
Explicación del funcionamiento:

1. Tomar una copia del código fuente integrado actual en la máquina de desarrollo.
2. Modificar el archivo con los cambios necesarios (alterar el código + alterar las pruebas).
3. Compilar en nuestra máquina.
   a) Si hay errores, corregirlos.
   b) Si no hay errores:
      1) Actualizar nuestra copia con el código fuente del *mainline* (pudo haber cambiado mientras trabajábamos).
      2) Actualizar el mainline con nuestro código.

Prácticas del *Continuous Integration*:

1. Mantener un repositorio único.

   Existen herramientas que permiten administrar el código fuente, por ejemplo, Subversion. Todos los archivos deben estar en este único lugar.

   Corolario 13. *Se debe poder copiar el proyecto del mainline a una máquina cualquiera y compilarlo sin problemas.*

2. Automatizar la compilación.

   Corolario 14. *Se debe poder copiar el proyecto del mainline a una máquina cualquiera y compilarlo sin problemas, con un solo comando.*

   Existen herramientas para reducir el tiempo de compilación que comparan las fechas de modificación de los archivos y solo compilan si la fecha es más reciente.

3. Automatizar las pruebas.

   Se necesita una *suite* de pruebas automatizadas que busquen errores, y que indiquen si alguna prueba falló.

4. Todos integran con el *mainline*.

   El desarrollador actualiza su copia actual con el *mainline*, resuelve conflictos, y luego compila en su máquina. Si la compilación resulta exitosa, se puede actualizar la copia del *mainline*. Conviene integrar de forma diaria, pues cuanto más rápido se integra, más rápido se encuentran los errores.

5. Todas las compilaciones se hacen en una máquina de integración.

   Conviene tener una máquina de integración.

   a) Compilación manual: el desarrollador va a la máquina de integración, saca su copia, y compila.
   b) Servidor de CI: cuando un desarrollador actualiza la copia del programa en la máquina de integración, ésta lo compila y le avisa al programador el resultado.

   Muchas empresas realizan compilaciones programadas, por ejemplo, todas las noches. Esto tiene una desventaja: los errores no se detectan hasta el día siguiente.

6. Mantener rápida la compilación.

Corolario 15. *10 minutos para una compilación.*

El cuello de botella en las compilaciones suele darse en las pruebas. Conviene hacer compilaciones en partes:

a) *Commit build*: compilación + pruebas unitarias.

b) *Secondary build*: pruebas de integración (por ejemplo, con accesos a bases de datos).

7. Probar en máquinas clones.

   Diseñar un ambiente para correr las pruebas que sean una réplica exacta de la máquina del usuario final (misma base de datos, mismas versiones, mismo sistema operativo, mismos puertos, mismas direcciones IP, mismo hardware, etc.) Hoy en día es útil la virtualización.

8. Fácil acceso al ejecutable.

   Todos los involucrados en el desarrollo de software (incluyendo el cliente) deben poder obtener una copia del último ejecutable, para demostraciones, pruebas, y para ver cambios recientes. Esto sucede porque es más fácil ver qué cambios hacerle al programa una vez que lo vemos.

9. Todos ven lo que sucede.

   Todos deben poder ver cómo está avanzando el desarrollo del programa y los cambios recientes. Esto puede hacerse posible mediante herramientas que informan visualmente el estado del proyecto.

10. Automatizar el despliegue de archivos.

Ventajas:

- Todo el tiempo se sabe lo que anda, lo que no, y los errores del sistema.
- Es más fácil encontrar errores, y más rápido arreglarlos.
- Hay menos errores.
- Hay menos barreras entre los usuarios y los desarrolladores.

Cómo introducir *Continuous Integration*:

1. Automatizar la compilación.
2. Automatizar las pruebas.
3. Aumentar la velocidad de compilación.

16 *"Domain-Driven Design"* (Eric Evans)

Método de modelización efectiva:

1. Vincular el modelo con la implementación.
2. Usar un lenguaje basado en el modelo.
3. Desarrollar un modelo rico en información.
4. Refinar el modelo (entidades + actividades + reglas).
5. *Brainstorming* y experimentación.

Knowledge crunching: transformar la información en modelos valiosos que le den sentido al problema. No es una actividad solitaria: los expertos del dominio y los desarrolladores trabajan en conjunto. Ambos equipos ganan conocimientos: los desarrolladores conocimiento técnico y habilidad para modelar, y los expertos entienden más acerca del tema que están tratando.

Método de cascada: los expertos hablan con los analistas, los analistas digieren y abstraen, y los programadores codifican el software. A este modelo le falta *feedback*. La información va en una dirección, y no se acumula.

17 "*Design in Construction*" (Steve McConnell)

17.1 Desafíos del diseño

Diseño: actividad que enlaza los requerimientos con la codificación y el debugging.

Todos los proyectos, grandes o pequeños, se benefician con el diseño. En los pequeños es útil, en los grandes es indispensable.

1. El diseño es "malvado". Es un problema que se puede resolver sólo cuando lo resuelves.
2. El diseño es un proceso "desprolijo". Se cometen muchos errores. Es difícil saber cuándo llegamos a un buen diseño.
3. El diseño requiere "compromisos y prioridades". Se requiere un balance entre todas las características.
4. El diseño requiere "restricciones". Debido a que los recursos son limitados, se deben restringir posibilidades.
5. El diseño es "no determinista". Hay muchas formas distintas de diseñar un programa.
6. El diseño es un proceso "heurístico". Las técnicas de diseño suelen ser heurísticas.
7. El diseño es "emergente". No surgen directamente en el cerebro humano, sino que evolucionan y mejoran.

17.2 Conceptos de diseño

- **Administrar la complejidad**

  Es el tema más importante del desarrollo de software. Se refiera a concentrarse en una parte a la vez. La complejidad de un problema se reduce cuando dividimos el sistema en subsistemas.

  Hay dos clases de problemas:

  - Esenciales: surgen cuando hay que interactuar con el mundo real.
  - Accidentales: la mayoría ya fueron solucionados. Ejemplo: sintaxis torpes.

  Cómo atacar la complejidad:

  - Minimizar la complejidad que el programador tiene que soportar en cualquier momento.
  - Impedir que la complejidad accidental se prolifere.

- **Características de un buen diseño**

  - Complejidad mínima: hacer diseños simples y fáciles de entender.
  - Fácil mantenimiento: hacer diseños obvios.
  - Alta cohesión: disminuir al mínimo la cantidad de responsabilidades de una clase.
  - Bajo acoplamiento: diseñar clases con la menor cantidad posible de interconexiones.
  - Extensibilidad y encapsulamiento: diseñar clases de modo tal que un cambio en una no afecte a las demás.
  - Reusabilidad: diseñar clases de modo tal que luego puedan ser reutilizadas.
  - *High fan-in*: hacer un buen uso de clases de utilidad en los niveles más bajos.
  - *Low-to-medium fan-out*: una clase dada debe usar pocas clases.
  - Portabilidad: diseñar el sistema de modo que pueda ser fácilmente llevado a otros entornos.
  - *Leanness*: diseñar el sistema para que no tenga partes demás.
  - Estratificación: mantener los niveles de descomposición estratificados.
  - Técnicas estándar: utilizar técnicas comunes y estandarizadas.
  - Principio "abierto-cerrado" (*Bertrand Meyer*): las clases deben s9er cerradas para la modificación, pero abiertas a la reutilización.
  - Favorecer la delegación sobre la herencia.

- Principio de sustitución de Liskov: los subtipos deben ser sustituibles en todas partes por sus tipos base. La aplicación mas elemental de este principio es el chequeo de la relación "es un".
  - Precondiciones no pueden ser más estrictas que las de su ancestro
  - Postcondiciones no pueden ser más laxas que las de su ancestro
  - Invariantes deben ser los mismos
  - Excepciones deben ser las mismas, o derivadas

- **Niveles de diseño**

  1. Software de sistema
  2. Paquetes: cuidar la cantidad de comunicaciones entre los paquetes.
  3. Clases: identificar todas las clases y definir las interacciones entre ellas.
  4. Rutinas.
  5. Diseño interno de las rutinas.

- **"Malos olores" en el diseño**

  - Ciclos muy anidados.
  - Código duplicado, que causa modificaciones paralelas.
  - Métodos complejos o muy largos.
  - Clases muy grandes o con varias responsabilidades.
  - Abundancia de sentencias `switch` o `if` anidados.
  - Largas secuencias de llamados a métodos.
  - Clases sin comportamiento (solo con atributos).
  - Atributos no encapsulados.
  - Uso de tipos primitivos o básicos para conceptos diversos.
  - Comentarios que explican código difícil de leer.

17.3 Heurísticas de diseño

1. Encontrar objetos del mundo real: sus atributos, sus responsabilidades y tareas, sus interacciones con otros objetos (contención y herencia), sus partes visibles y ocultas, su interfaz.

   a) Formar abstracciones consistentes: "puedes mirar a un objeto desde un nivel de detalle alto".

   b) Encapsular detalles de implementación: "no puedes mirar a un objeto desde un nivel de detalle que no sea alto".

   c) Heredar cuando corresponda: definir diferencias y similitudes entre objetos. El mayor beneficio de la herencia es que es compatible con la abstracción.

   d) Ocultar secretos y complejidad: la interfaz de una clase debería mostrar lo menor posible acerca de su interior. Su uso está recomendado en todos los niveles (desde el uso de constantes y literales hasta el diseño de un sistema). Es útil porque hace a los programas más fáciles de modificar e inspira soluciones efectivas.

   Corolario 16. *Preguntarse frecuentemente: "¿qué información debo ocultar?"*

   e) Identificar áreas con alta probabilidad de cambiar y separarlas del resto. Ejemplos:

   1) Reglas generales
   2) Dependencias de hardware
   3) Entrada y salida
   4) Características del lenguaje que no son estándar
   5) Áreas de diseño difíciles
   6) Variables de estado (usar `enum` en vez de `boolean`; usar rutinas de acceso en vez de acceder a la variable directamente)
   7) Restricciones de tamaño

   f) Mantener bajo el acoplamiento. Criterios a tener en cuenta:

1) Número de conexiones entre módulos
2) Prominencia de las conexiones
3) Facilidad de cambio a las conexiones

Corolario 17. *Las clases y rutinas deberían ayudarnos a reducir la complejidad. Si no lo hacen, no están haciendo su trabajo.*

g) Utilizar patrones de diseño. Sus ventajas:

1) Reducen la complejidad
2) Reducen los errores
3) Tienen valor heurístico
4) Aumentan la agilidad de la comunicación entre programadores que ya saben de patrones de diseño

h) Otras técnicas:

1) Alta cohesión: una clase debe concentrarse en un propósito único.
2) Jerarquías: representaciones generales o abstractas en la cima, y representaciones especializadas o detalladas en la base.
3) Diseño por contrato: definir precondiciones y postcondiciones.
4) Asignar responsabilidades a objetos.
5) Diseñar para las pruebas
6) Considerar posibles errores futuros
7) Considerar usar la fuerza bruta
8) Dibujar diagramas
9) Modularizar

- Guía para el uso de heurísticas

1. Entender el problema.
2. Diseñar un plan. Encontrar la conexión entre los datos y las incógnitas.
3. Llevar a cabo el plan.
4. Mirar atrás. Examinar la solución.

17.4 Prácticas de diseño

1. **Iterar**
2. **División y conquista**
3. **Diseño *top-down & bottom-up***

	Definición	Ventajas	Desventajas
Top-down	Comenzar con niveles de abstracción altos y aumentar progresivamente el nivel de detalle.	Fácil, permite postergar los detalles.	Un cambio en los detalles puede afectar todo.
Bottom-up	Comenzar con los detalles y aumentar progresivamente hacia generalidades.	Diseño compacto y bien factorizado.	Es difícil de usar exclusivamente, en algunos casos no puede aplicarse.

4. **Prototipos experimentales**: escribir el código mínimo necesario para responder a una pregunta de diseño.

a) La pregunta debe ser específica.
b) Los programadores deben escribir lo mínimo necesario, y no pensar si el prototipo será utilizado.
c) Los programadores deben recordar que el código debe ser fácilmente eliminado. Posible solución: programarlo en otro lenguaje que el sistema productivo.

5. **Diseño colaborativo**

Corolario 18. *Dos cabezas piensan mejor que una.*

17.5 Epílogo

Corolario 19. *¿Como minimizar la complejidad?*

- *Encapsular detalles*
- *Minimizar la cantidad de interconexiones*
- *Formar abstracciones consistentes*
- *Ocultar secretos*
- *Utilizar clases y rutinas*

18 Usabilidad (Carlos Fontela)

Usabilidad / Experiencia del Usuario (UX): es un campo multidisciplinario.

- Facilidad con que los usuarios aprenden y utilizan una aplicación + comodidad en el uso
- Grado de efectividad de interacción entre máquina y usuario
- Ausencia de frustración del usuario mientras usa algo, para lo que se supone que sirve

$$\texttt{Usabilidad} \neq \texttt{Desempeño} \neq \texttt{Funcionalidad} \neq \texttt{Estética}$$

¿Por qué es importante?

- Malas interfaces cuestan dinero, o incluso vidas
- Malas interfaces están acompañadas de enormes manuales que nadie lee
- Costos de soporte al cliente
- Improductividad en empresas clientes
- Frustración de usuarios
- Los usuarios no son todos iguales, hay discapacidades (físicas, intelectuales, etc.) y diferencias culturales o intelectuales
- El usuario lo valora ante la competencia y antes de comprar, y si no le gusta nos puede abandonar

Teorema 20. *"Don't make me think!" (Steve Krug)*

Malos ejemplos de usabilidad:

- Mensajes al usuario que no aportan información. Ejemplos:
  - "El servicio no está disponible, intente luego"
  - "No se pudo procesar el pedido"
- Mensajes al usuario que no le permiten interactuar.
- Mensajes al usuario falsos.
- Ventanas emergentes.
- Problemas de consistencia en los diálogos.
- Programas que no son compatibles con uno o más sistemas operativos.

Tipos de usuarios:

Según el tiempo	**Según la experiencia**	**Según la aplicación**
Early-adopter	Experto	Cautivo o semi-cautivo
Usuario común	Intermedio	Libre para comprar
	Novato	Independiente

La elección de la interfaz del usuario depende de:

- Tipo de usuario
- Tipo de tarea
- Medios de control disponibles

18.1 Estilos de interfaces

	Ventajas	Desventajas
Lenguaje de comandos	Rápido y flexible para expertos	Difícil para novatos, requiere memorización, alta tasa de error
Menúes	Organización y agrupación de la información, reduce la necesidad de teclear	Lento para expertos, molesto si hay muchas opciones
Atajos de teclado	Fácil de implementar y aprender	Requiere memorización, disponibilidad limitada
Wizards	Fácil de implementar, fácil para todo tipo de usuarios	A veces es imposible volver atrás, lento para expertos
Formularios	Simplifica la entrada de datos, organización de información	Consume espacio en pantalla, lento para expertos
Manipulación directa	Intuitivo, ameno, rápido	Limitado, difícil de implementar
WYSIWYG	Correspondencia con el resultado real	Difícil de implementar, no muestra la estructura subyacente
Hipermedia	Fácil navegación, independencia de la secuencialidad	Dependencia de un buen diseño, abuso
Rich web	Fácil de usar	Problemas de usabilidad, mucho ruido
Dispositivos móviles	Movilidad	Gráficos de poca calidad
Computadoras embebidas	Información contextual	Limitaciones para entrada de datos

18.2 Aspectos de usabilidad

1. **Accesibilidad**: diseño flexible, para todas las configuraciones posibles, para usuarios con carencias
2. **Navegabilidad**: múltiples formas de acceder a la información o a la funcionalidad; consistencia con el resto de la aplicación
3. **Productividad**: minimizar el esfuerzo del usuario
4. **Contenido de calidad**
5. **Optimización**: tiempos de descarga, ejecución, respuesta; uso de recursos

Características de los usuarios:

- NO leen todo lo que ven
- NO analizan todas las opciones disponibles, sino que eligen la primera opción razonable
- SI valoran la velocidad
- NO les interesa el porqué de la lentitud
- SI están apurados
- SI usan las cosas sin saber cómo funciona ni para qué están realmente hechas

18.3 Consejos

- Identificar el tipo de nuestro usuario y hablar su lenguaje
- Utilizar los principios del diseño gráfico, la simbología, y metáforas
- Prevenir errores, informarlos detalladamente, y facilitar su corrección
- Los detalles importan

- Hacer programas autoexplicativos pues la ayuda no se tiene en cuenta
- Ser intuitivos, consistentes, flexibles, cómodos, correctos y útiles
- No complicar la tarea del usuario, y más si es novato
- Promover el aprendizaje
- No hagas pensar al usuario y no hagas que pierda su tiempo
- Hacer fácil volver atrás
- Utilizar jerarquías de información apropiadas
- No poner varias formas de hacer lo mismo
- Proveer feedback

18.4 *User-Centered Design*

Pilares:

1. Enfocarse tempranamente en el usuario y sus tareas: entender cómo trabajan y piensan
2. Hacer que los usuarios evalúen el producto: se trabaja con él durante todo el diseño
3. Diseñar iterativamente

Métodos:

1. Prototipación: producir prototipo, mostrárselo a los usuarios, corregir errores, producir nuevo prototipo...
2. *Usability Testing*: realizar sesiones con 3 a 5 usuarios reales, en donde ellos prueban tareas, y los diseñadores observan para exponer defectos
3. Evaluación por expertos
4. Estudios de seguimiento
5. Encuestas