Codificación

Objetivo:
Implementar el diseño de la *mejor manera posible*.
La codificación afecta al testing y al mantenimiento.

El objetivo es reducir los costos de testing y mantenimiento.
El código debe ser *fácil de leer y comprender* (no necesariamente fácil de escribir).

Principios y pautas para la programación

Que el código...

- Sea simple
- Sea fácil de leer
- Tenga la menor cantidad de errores

Errores comunes de codificación

- "memory leaks"
- Liberar memoria ya liberada
- Des-referencias de punteros a NULL.
- Falta de unicidad en direcciones
- **Errores de sincronización**: deadlocks, condiciones de carrera, sincronización inconsistente.
- Índice de arreglos fuera de límites
- Excepciones aritméticas
- Off by one (ejemplo: <= en vez de <)
- Uso ilegal de `&` en lugar de `&&`
- Errores de manipulación de strings
- Buffer overflow
- Tipos de datos creados por el usuario (tener cuidado, describir bien las cotas) no es un error per sé

Programación estructurada

En contra del uso indiscriminado de **constructores de control** como los "`goto`".

El objetivo es escribir programas cuya estructura dinámica es la misma que la estática para que sea fácil razonar sobre los programas. (O lo más parecido posible).

- **Estructura estática:** orden de las sentencias en el código (lineal).
- **Estructura dinámica:** orden en el cual las sentencias se ejecutan.

Para mostrar que un programa es **correcto** debemos mostrar que el **comportamiento dinámico** es el esperado. Esto es más simple si la estructura dinámica y la estática son similares.

Como las sentencias se organizan linealmente (estático), el objetivo es desarrollar programas cuyo flujo de control es lineal.

La programación estructurada **simplifica** el flujo de control, facilitando en consecuencia tanto la **comprensión** de los programas así como el **razonamiento** (formal o informal) sobre estos.

El programador debe ser consciente de las suposiciones que hace sobre las llamadas a las funciones.

Ocultamiento de la información

Las soluciones de software siempre contienen estructuras de datos que guardan información.

Solo **ciertas operaciones** se realizan sobre la información; y esta información debería ocultarse de manera que solo quede expuesta a esas pocas operaciones (práctica fundamental en OO).

El ocultamiento de la información **reduce acoplamiento**.

Prácticas de programación

- **Usar pocos constructores de control**: Utilizar algunos pocos constructores estructurados.
- **No usar** `goto` (limitado al caso donde las alternativas son peores).
- **Usar ocultamiento de la información.**
- **Tipos definidos por el usuario:** Usar para facilitar la lectura.
- **Tamaño de los módulos chicos:** Un módulo chico evita baja cohesión.
- **Hacer la interfaz del módulo simple**.
- **Robustez:** Manipular situaciones excepcionales
- **Evitar efectos secundarios:** Evitarlos o documentar.
- **No dejar un bloque** `catch` **vacío**: Poner al menos una acción por defecto.
- **No dejar** `if` **o** `while` **vacíos**: Pésima práctica.
- **Usar default en** `switch case`
- **Leer valores de retorno en lecturas:** para lograr robustez.
- **No usar** `return` **en** `finally`.

- **Usar siempre fuentes de datos confiables** y siempre desconfiar (usar psw, hash, etc).
- **Dar importancia a las excepciones:** los casos excepcionales son los que tienden a hacer que el programa funcione mal.

Estándares de codificación

La **legibilidad del código** aumenta si todos siguen ciertas **convenciones** de codificación.

Cierta dependencia de lenguaje/empresa/comunidad...

Java

Nombres

Nombre de	Convención
Paquetes	En minúscula
Tipos	Sustantivos en mayúscula
Variables	Sustantivos con minúscula
Constantes	Todo en mayúscula
Métodos	Verbos con minúscula
Booleanos	Prefijar con "is"

Archivos

- Los archivos fuentes tienen la extensión `.java`.
- Cada archivo contiene solo una clase externa con igual nombre.
- Longitud de la línea < 80 char.

Sentencias

- *Inicializar* variables cuando se declaran
- Declararlas en el *scope más pequeño* posible.
- Declarar conjuntamente variables que están relacionadas.
- Declarar separadamente variables no relacionadas.
- Las variables de clases nunca deben ser públicas.
- Inicializar las variables de los loops justo antes de estos.
- Evitar uso de `break` y `continue` en loops.
- Evitar sentencias ejecutables en condicionales.
- Evitar uso de `do ... while`.

Comentarios y layout

- Los comentarios de una sola línea para un bloque deben alinearse con el bloque del código.
- Debe haber comentarios para todas la variables más importantes describiendo qué representan.
- Un bloque de comentario debe comenzar con una línea conteniendo sólo /* y finalizar con una línea conteniendo sólo */
- Los comentarios en la misma linea que una sentencia deben ser cortos y alejados a derecha.

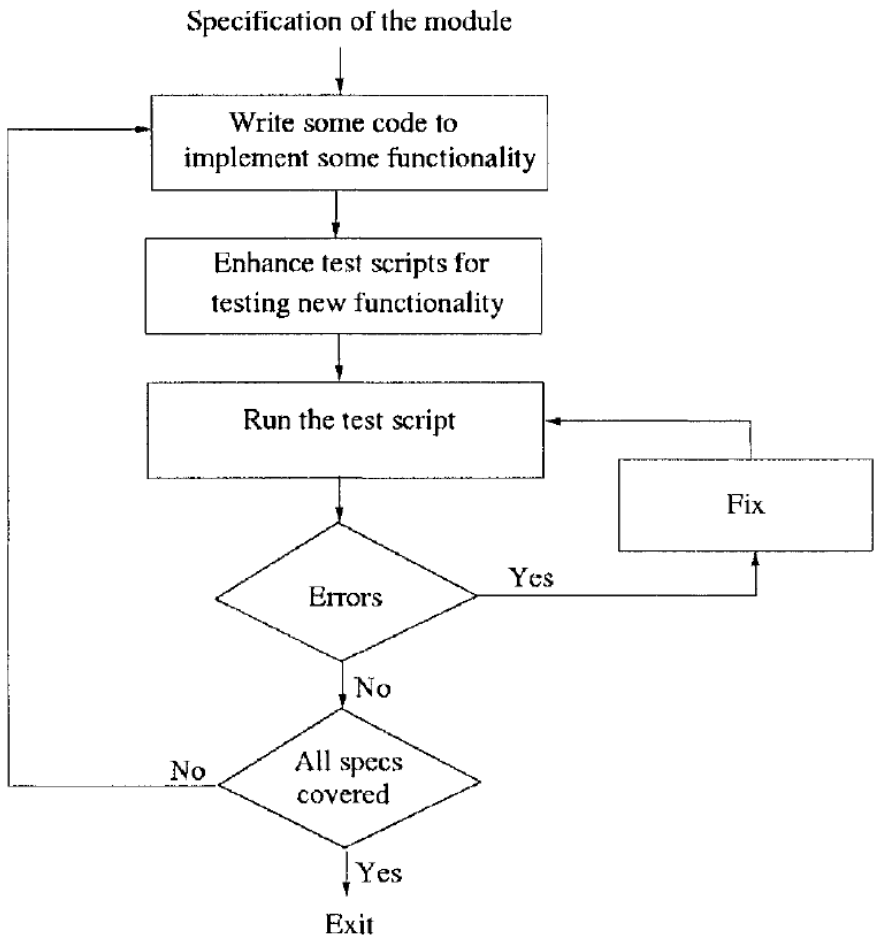
Proceso de codificación

Comienza ni bien está disponible la especificación del diseño de los módulos. Usualmente los módulos se asignan a programadores individuales.

Proceso de codificación incremental

Proceso básico

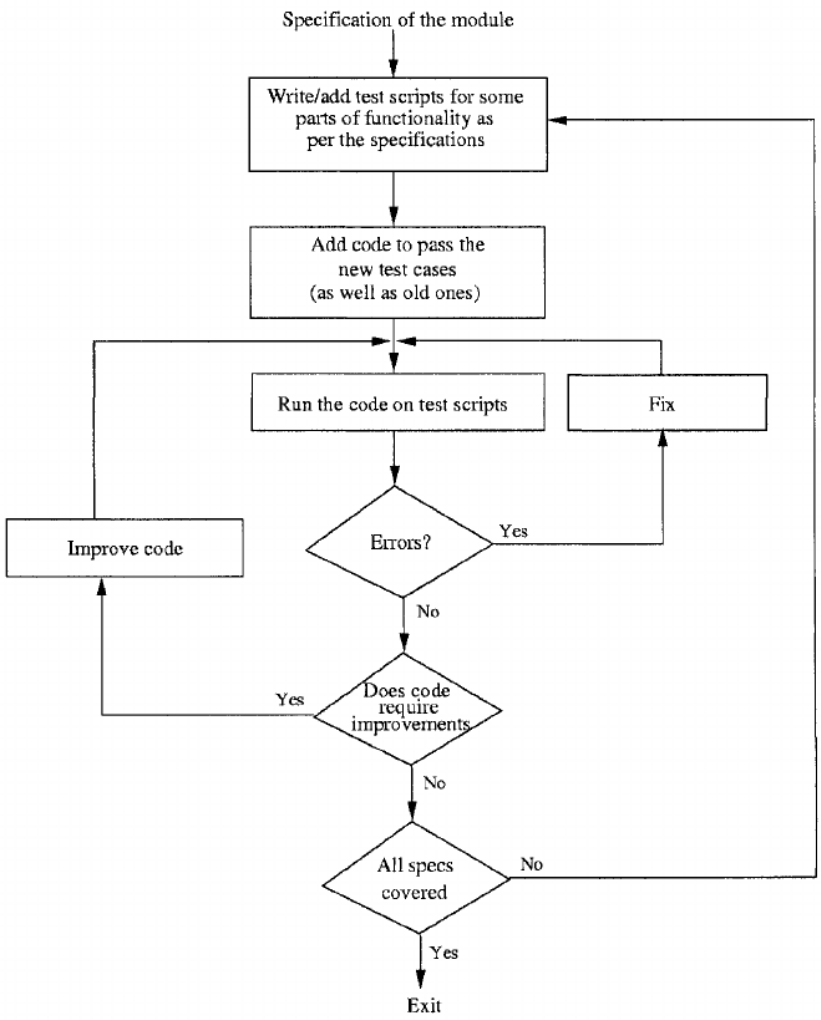
1. Escribir código del módulo
2. Realizar test de unidad
3. Si error: arreglar bugs y repetir tests.



Proceso de codificación incremental.

Desarrollo dirigido por test TDD

TDD: *Test Driven Development*



Proceso de codificación dirigido por test.

Test != script de test

- Se escriben "scripts de test" no "tests" porque estos últimos son solo una tupla de `(entrada, salida esperada)`.

Ventajas	Desventajas
Es completamente testeado.	La completitud del código depende de cuan exhaustivos sean los casos de test.
Ayuda a validar la interfaz del usuario especificada en diseño.	El código necesitará factorización.

- La responsabilidad de asegurar cobertura de toda la funcionalidad radica en el diseño de los casos de test y no en la codificación.
- **Incrementalidad**
- Código sucio: hace falta refactoreo

Programación de a pares

El código se escribe de a 2.
Una tipea otra señala errores. Los roles se alteran periódicamente.

- Hay una revisión continua de código.
- Mejor diseño de algoritmos /estructuras de datos/lógica/..
- Es más difícil que se escapen las condiciones particulares.

La efectividad no está demostrada.

Control de código fuente y construcción

Se usan herramientas como `Git` (repositorio: estructura de directorio controlada y fuente oficial para todos los archivos del código).

Refactorización

¿Qué es?

La **refactorización** es la tarea que permite realizar cambios en un programa con el fin de simplificarlo y mejorar su comprensión (i.e. hacerlo testeable y mantenible), sin cambiar el comportamiento observacional de éste.

No debería cambiar la funcionalidad del código. El objetivo NO es corregir bugs ni agregar características. Se debe hacer sobre código que funciona bien.

La refactorización se realiza durante **codificación** y generalmente está asociada a un **requerimiento de cambio**. Los cambios por refactorización se realizan separadamente de la codificación normal.

La refactorización intenta:

- Reducir acoplamiento.
- Incrementar cohesión.
- Mejorar la respuesta al principio abierto-cerrado.

¿Por qué es necesaria?

Siempre es necesaria porque el código se modifica con el fin de aumentar su funcionalidad, con el tiempo, aún si el diseño inicial era bueno, los cambios en el código deterioran el diseño. Al complicarse el diseño, comienza a hacerse más complicado modificar el código y más susceptible a errores.

La refactorización permite que el diseño del código mejore continuamente en lugar de degradarse con el tiempo.

- El código extra de la refactorización se recupera en la reducción del costo en los cambios.

- No es necesario tener el diseño más general desde el comienzo; se pueden elegir diseños más simples.
- Hace más fácil y menos riesgosa la tarea inicial de diseño.

¿Cómo refactorizar?

Para disminuir la posibilidad de "romper" la funcionalidad existente, el código debe refactorizarse en **pasos pequeños**, y así saber dónde se cometen errores. Y además se debe disponer de scripts de **tests automatizados** para testear la funcionalidad existente.

¿Cuándo se necesita?

Si se detecta alguno de los siguientes signos en el código, es posible la necesidad de refactorización.

1. **Código duplicado:** La misma funcionalidad aparece en lugares distintos.
2. **Métodos largos:** Podría estar haciendo demasiadas cosas.
3. **Clases grandes:** Puede haber baja cohesión, encapsulando muchos conceptos.
4. **Lista larga de parámetros**
5. **Sentencia `switch`:** Podría no estar usando herencia, si es así `switch`s similares se repetirán en otros lugares (violación principio abierto-cerrado).
6. **Generalidad especulativa:** La subclase es la misma que la superclase, no hay razón aparente para esta jerarquía.
7. **Demasiada comunicación entre objetos**
8. **Encadenamiento de mensajes:** un método llama a otro que llama a otro...; posible acoplamiento innecesario.

Refactorizaciones más comunes

Mejoras de métodos:

Extracción de métodos

- Se realiza si el método es demasiado largo.
- Objetivo: Separar en métodos cortos cuya signatura indique lo que el método hace.
- También se realiza si un método retorna un valor y a la vez cambia el estado del objeto (dividir en dos métodos).

Agregar/eliminar parámetros

- Para simplificar las interfaces.
- Agregar solo si los parámetros existentes no proveen la información que se necesita.
- Eliminar los que no se usan.

Mejoras de clases

Desplazamiento de métodos

Se realiza cuando el método interactúa demasiado con los objetos de otra clase. Inicialmente puede ser conveniente dejar un método en la clase inicial que delegue al nuevo (debería tender a desaparecer).

Desplazamiento de atributos

Si un atributo se usa más en otra clase, moverlo. Así se mejora cohesión y acoplamiento.

Extracción de clases

Si una clase agrupa múltiples conceptos, separar cada concepto en una clase distinta.

Reemplazar valores de datos por objetos

Una colección de atributos se puede transformar en una entidad lógica. En ese caso, separarlos como una clase y definir objetos para accederlos.

Mejoras de jerarquías

Reemplazar condicionales con polimorfismos

Si el comportamiento depende de algún indicador de tipo, no se está explotando el poder de la OO.

Reemplazar tal análisis de casos a través de una jerarquía de clases apropiada.

Subir métodos/atributos en la jerarquía de herencia

Los elementos comunes deben pertenecer a la superclase.
Si la funcionalidad o atributo esta duplicado en las subclases pueden subirse a la superclase.

Verificación

Verificación del código por parte del mismo programador.

Técnicas

Inspección de código

- Proceso de revisión, luego de que el código esté compilado, testeado algunas veces, y chequeado con herramientas de análisis estático.

- Consiste en encontrar defectos y bugs en el código. Se utilizan listas de control para enfocar la atención.
- Es caro, efectivo y ampliamente usado en la industria.

Test de unidad

- Es testing enfocado en el **módulo** escrito por un programador. Y lo realiza este mismo.
- Requiere casos de test para el módulo; y la escritura de "drivers" que ejecuten el módulo con los casos de test.
- Si se realiza codificación incremental, entonces el TU completo necesita automatizarse
- Los tests pueden dar como resultado: pass/fail/inconclusive.

Análisis estático

- Son herramientas para analizar los programas fuentes y verificar la existencia de problemas.
- No pueden encontrar todos los bugs y en ocasiones dan **falsos positivos**.

Falso Negativo	Falso Positivo
Un test pasó, pero había un bug.	Dice que hay error, pero no.

Son efectivas para encontrar bugs como: memory leaks, código muerto, punteros colgando, etc.

Métodos formales

- Apuntan a demostrar la corrección de los programas. Es decir, a demostrar que el programa implementa la especificación dada.
- Requiere especificaciones formales.
- No son escalables.
- Utilizado en software crítico.