Ingeniería del Software I

6 – Codificación (Capítulo 9)

Codificación

Objetivo: implementar el diseño de la mejor manera posible

- La codificación afecta al testing y al mantenimiento.

- Como los costos de testing y de mantenimiento son muy altos (en particular, comparado con codificación), el propósito de la codificación es escribir código que reduzca tales costos.

  => el objetivo NO es reducir los costos de implementación,
  sino los de testing y mantenimiento,
  i.e. facilitar el trabajo de quienes testearán y de quienes mantendrán el sistema.

Codificación

Un programa se lee mucho más frecuentemente que el tiempo que demanda su escritura:

- Los programadores leen el código muchas veces para hacer debugging, extenderlo, modificarlo, etcétera.
- Quienes mantienen el código invierten mucho esfuerzo en la lectura y comprensión del código.
- Otros desarrolladores leen el código cuando agregan otras partes al programa.

Entonces: el código debe ser fácil de leer y comprender (¡pero no necesariamente de escribir!).

Afortunadamente el experimento de Weinberg muestra que los programadores responden a los objetivos que se les imponen.

Principios y pautas para la programación

- Objetivo principal del programador: escribir programas simples y fáciles de leer con la menor cantidad de errores (bugs) posibles.
- Por supuesto: también debe hacerlo rápidamente para que su productividad sea alta.
- Existen principios y pautas para la programación que pueden ayudar a escribir código de alta calidad (i.e. fácilmente comprensible), que sea fácil de testear y mantener.

Principios y pautas para la programación

Errores comunes de codificación

- “memory leaks”
- liberar memoria ya liberada
- desreferencias de punteros a NULL
- falta de unicidad en direcciones
- errores de sincronización:
  - deadlocks
  - condiciones de carrera
  - sincronización inconsistente
- índice de arreglo fuera de límites
- excepciones aritméticas
- “justo por uno” (ej.: <= por <)
- uso ilegal de & en lugar de &&
- errores de manipulación de strings
- buffer overflow

Son sólo algunos ejemplos

Principios y pautas para la programación

Programación estructurada

- La programación estructurada se inició en los 70, fundamentalmente contra el uso indiscriminado de constructores de control como los “gotos”.
- Objetivo: simplificar la estructura de los programas de manera que sea fácil razonar sobre ellos.
- Hoy es un paradigma bien establecido y utilizado.

Principios y pautas para la programación

Programación estructurada

- Un programa tiene una estructura estática la cual es el orden de las sentencias en el código (el cual es un orden lineal).
- Un programa tiene una estructura dinámica que es el orden el cual las sentencias se ejecutan.
- Cada estructura define un orden en las sentencias.
- La corrección de un programa debe hablar de la estructura dinámica.

Principios y pautas para la programación

Programación estructurada

- Para mostrar que un programa es correcto debemos mostrar que el comportamiento dinámico es el esperado.

- Pero debemos razonar sobre el código del programa, i.e. la estructura estática.
  i.e. la justificación del comportamiento de un programa se realiza sobre el código estático.

- Esto sería más simple si las estructuras dinámica y estática fueran similares:
  una correspondencia cercana facilitará la comprensión del comportamiento dinámico desde la estructura estática.

- Objetivo de la programación estructurada: escribir programas cuya estructura dinámica es la misma que la estática,
  i.e. las sentencias se ejecutan en el mismo orden que las presenta el código.

Principios y pautas para la programación

Programación estructurada

- Para mostrar que un programa es correcto debemos mostrar que el comportamiento dinámico es el esperado.
- Pero debemos razonar sobre el código del programa, i.e. la estructura estática.
  i.e. la justificación del comportamiento de un programa se realiza sobre el código estático.
- Esto sería más simple si las estructuras diná
  una correspondencia cercana facilitará la co dinámico
  desde la estructura estática.

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

- Objetivo de la programación estructurada: escribir programas cuya estructura dinámica es la misma que la estática,
  i.e. las sentencias se ejecutan en el mismo orden que las presenta el código.

Principios y pautas para la programación

Programación estructurada

- Los constructores de la programación estructurada son de una única entrada y una única salida.
- De esta manera, la ejecución de las sentencias se realizan en el orden en el que aparecen en el código.
  => El orden dinámico y el estático son el mismo orden.
- Los constructores estructurados no pueden ser arbitrarios: ellos deben mostrar un comportamiento claro.
- Se puede mostrar que los constructores “if”, “while”, y la secuencia alcanzan para escribir cualquier tipo de programa.

Principios y pautas para la programación

Programación est

- Los constructores de la programación [illegible] una única salida.
- De esta manera, la ejecución de las sentencias se realiza [illegible] orden en el que aparecen en el código.
  => El orden dinámico y el estático son el mismo orden.
- Los constructores estructurados no pueden ser arbitrarios: ellos deben mostrar un comportamiento claro.
- Se puede mostrar que los constructores “if”, “while”, y la secuencia alcanzan para escribir cualquier tipo de programa.

Entonces: 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

Principios y pautas para la programación

Ocultamiento de la información

- Las soluciones de software siempre contienen estructuras de datos que guardan información.
- Los programas trabajan sobre estas estructuras de datos para realizar ciertas funciones.
- En general sólo ciertas operaciones se realizan sobre la información, i.e., los datos sólo se manipulan de pocas maneras.
- En consecuencia la información debería ocultarse de manera que sólo quede expuesta a esas pocas operaciones.
  i.e. las estructuras de datos son ocultadas tras las funciones de acceso que son las únicas que puede usar el programa.
- El ocultamiento de la información reduce acoplamiento.

Principios y pautas para la programación

Ocultamiento de la información

- Las soluciones de software siempre contienen estructuras que guardan información.
- Los programas trabajan sobre estas estructuras mediante funciones.
- En general sólo ciertas operaciones se realizan sobre la información, i.e., los datos sólo se manipulan de pocas maneras.
- En consecuencia la información debería ocultarse de manera que sólo quede expuesta a esas pocas operaciones.
  i.e. las estructuras de datos son ocultadas tras las funciones de acceso que son las únicas que puede usar el programa.
- El ocultamiento de la información reduce acoplamiento.

Esta práctica es fundamental en OO y uso de componentes, y es ampliamente utilizada hoy en día

Principios y pautas para la programación

Algunas prácticas de programación

- Constructores de control: Utilizar algunos pocos constructores estructurados (en lugar de una gran variedad de ellos).
- Gotos: No usar (limitado a caso donde las alternativas son peores).
- Ocultamiento de la información: ¡Usarla!.
- Tipos definidos por el usuario: Utilizarlos para facilitar la lectura de los programas.
- Tamaño de los módulos: No deberían ser muy largos (sino probable baja cohesión).
- Interfaz del módulo: Hacerla simple.
- Robustez: Manipular situaciones excepcionales.
- Efectos secundarios: Evitarlos, documentar (ej.: vars globales).

Principios y pautas para la programación

Algunas prácticas de programación

- Bloque “catch” vacío: Realizar siempre alguna acción por defecto en lugar de nada.
- “if” o “while” vacío: Pésima práctica.
- Switch case: Usar default.
- Valores de retorno en lecturas: leer para lograr robustez.
- “return” en “finally”: No usar.
- Fuentes de datos confiables: Desconfiar (usar psw, hash, etc.).
- Dar importancia a las excepciones: los casos excepcionales son los que tienden a hace que el programa funcione mal.

Principios y pautas para la programación

Estándares de codificación

- Los programadores pasan más tiempo leyendo código que escribiendo código.
- Leen tanto su propio código como el de otros programadores.
- La legibilidad del código aumenta si todos siguen ciertas convenciones de codificación.
- Los estándares de codificación proveen esas pautas para los programadores.
- Cierta dependencia de lenguaje / comunidad / empresa.

Principios y pautas para la programación

Estándares de codificación

Algunas convenciones en Java:

- Convenciones de Nombre:
  - Nombres de paquetes: en minúscula.
  - Nombres de tipos: sustantivos que comienzan con mayúscula.
  - Nombres de variables: sustantivos que comienzan con minúscula.
  - Constantes: todo en mayúscula.
  - Nombres de métodos: verbos que comienzan con minúscula.
  - Variables y métodos buleanos: prefijar con “is”.
- Archivos:
  - Los fuentes tienen la extensión “.java”.
  - Cada archivo contiene solo una clase externa con igual nombre.
  - Longitud de la linea < 80 char; si no continuar debajo y aclarar.

Principios y pautas para la programación

Estándares de codificación

Algunas convenciones en Java (continuación):

- 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”.

Principios y pautas para la programación

Estándares de codificación

Algunas convenciones en Java (continuación):

- 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 deben 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.

El proceso de codificación

- La codificación comienza ni bien está disponible la especificación del diseño de los módulos.
- Usualmente los módulos se asignan a programadores individuales.
- Desarrollo top-down => los módulos de los niveles superiores se desarrollan primero.
- Desarrollo bottom-up => los módulos de los niveles inferiore se desarrollan primero.
- Para la codificación se pueden utilizar distintos procesos.

El proceso de codificación

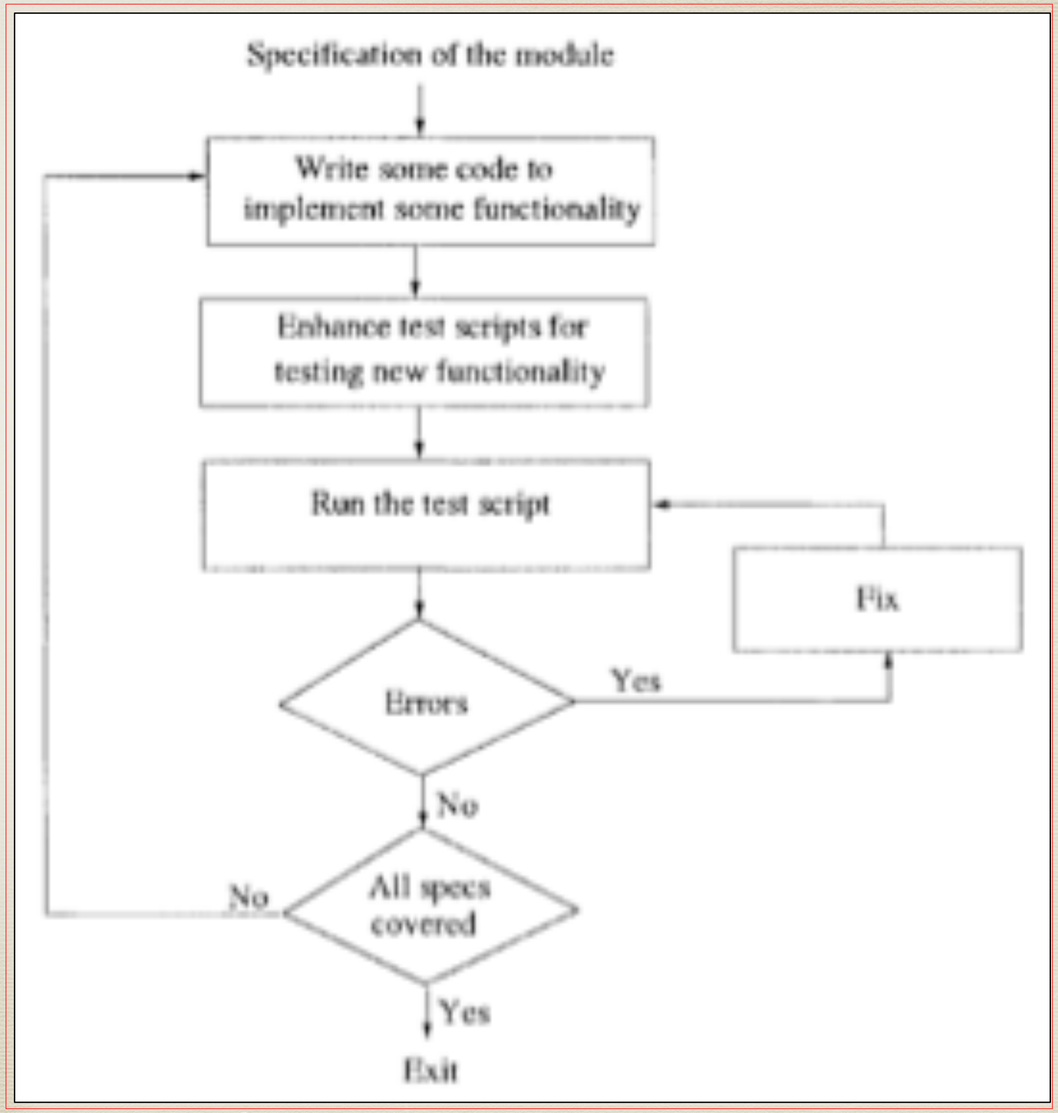
Proceso de codificación incremental

- Proceso básico:
  - Escribir código del módulo.
  - Realizar test de unidad.
  - Si error: arreglar bugs y repetir tests.

El proceso de codificación

Proceso de codificación incremental

Es mejor realizar el mismo proceso incrementalmente:



El proceso de codificación

Desarrollo dirigido por test

TDD: Test Driven Development.

- Este proceso de codificación cambia el orden de las actividades en la codificación.
- En TDD el programador primero escribe los scripts para los tests y luego el código para que estos pasen los casos de tests en el script.
- Se realiza incrementalmente.
- Nueva técnica, parte de Extreme programming (pero se puede usar independientemente).

El proceso de codificación

Desarrollo dirigido por test

- En TDD se escribe el código suficiente para pasar el test
  i.e. el código está en sincronía con los tests y es testeado por estos casos de test.
  No es lo mismo en el modelo anterior donde los casos de test podrían testear sólo parte de la funcionalidad.

- La responsabilidad de asegurar cobertura de toda la funcionalidad radica en el diseño de los casos de test y no en la codificación.

- Ayuda a asegurar que todo el código es testeable.

- Se enfoca en cómo será usado el código a desarrollar dado que los tests se escriben primero.
  Ayuda a validar la interfaz del usuario especificada en diseño.
  (Primeros tests se enfocan en funcionalidades principales).

El proceso de codificación

Desarrollo dirigido por test

- En TDD se escribe el código suficiente para pasar el test
  i.e. el código está en sincronía con los tests y es testeado por estos casos de test.
  No es lo mismo en el modelo anterior donde los c testear sólo parte de la funcionalidad.

- La responsabilidad de asegurar cobertur iseño de los casos de test y no en la codificación.

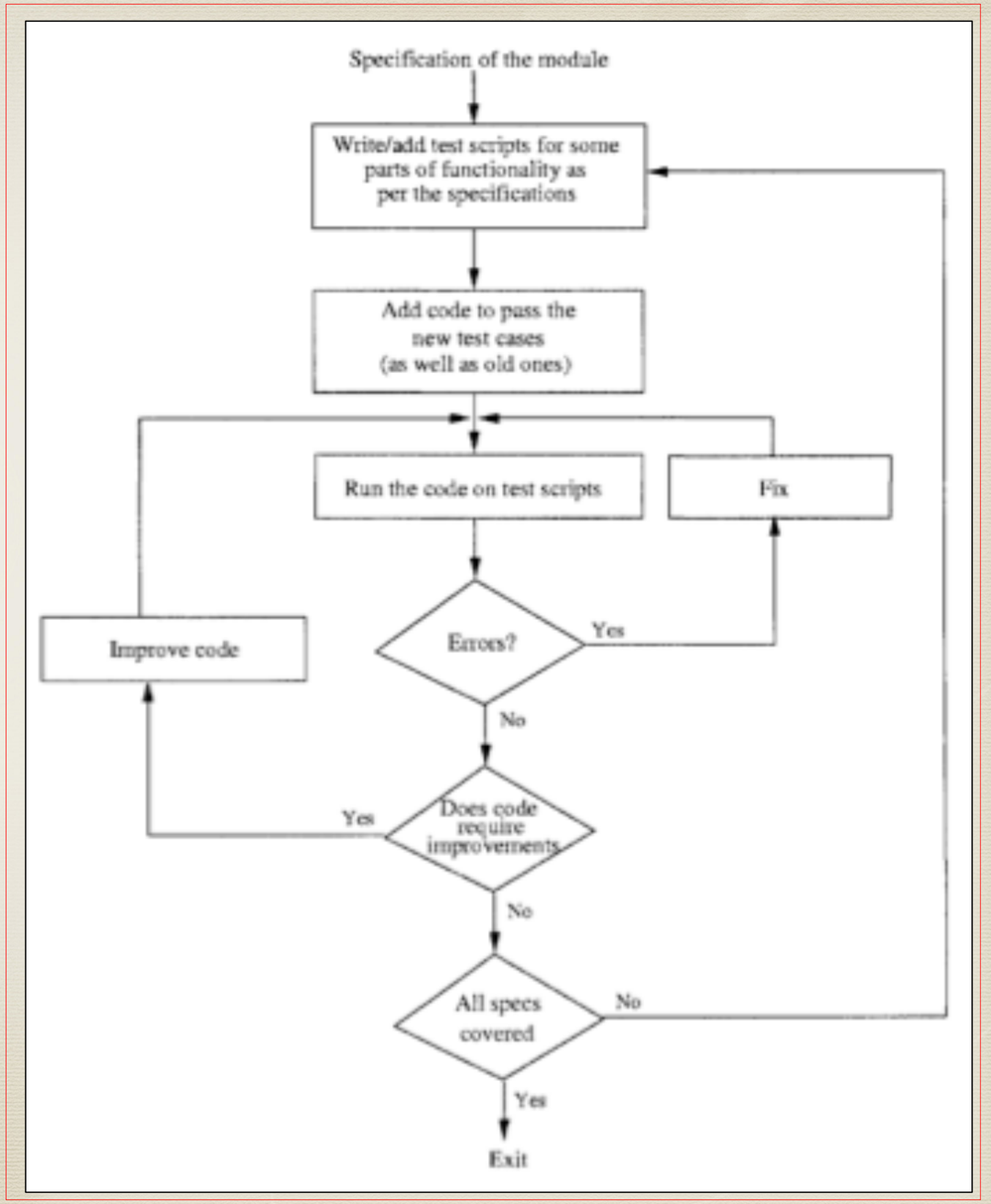
- Ayuda a asegurar que todo el código es testeable.

- Se enfoca en cómo será usado el código a desarrollar dado que los tests se escriben primero.
  Ayuda a validar la interfaz del usuario especificada en diseño.
  (Primeros tests se enfocan en funcionalidades principales).

Cuidado:
(1) La completitud del código depende de cuan exhaustivo sean los casos de test.
(2) El código necesitará refactorización para mejorar el código posiblemente confuso

El proceso de codificación

Desarrollo dirigido por test



El proceso de codificación

Programación de a pares

- También propuesto como práctica en XP(extreme programming).
- El código se escribe por dos programadores en lugar de uno solo:
  - Conjuntamente, ambos programadores diseñan los algoritmos, estructuras de datos, estrategias, etcétera.
  - Una persona tipea el código, la otra revisa activamente el código que se tipea.
  - Se señalan los errores y conjuntamente formulan soluciones.
  - Los roles se alternan periodicamente.
- La revisión de código en este modelo es continua.
- Mejor diseño de algoritmos/estructuras de datos/lógica/...
- Es más difícil que se escapen las condiciones particulares.
- La efectividad de este método no es aún bien sabida (perdida de productividad?)

El proceso de codificación

Control del código fuente y construcción (“built”)

- El control del código fuente es un paso escencial que los programadores deben realizar.
- Generalmente se usan herramientas como SVN, CVS, VVS, etc.
- La herramienta consiste en un repositorio, el cual es una estructura de directorio controlada.
- El repositorio es la fuente oficial para todos los archivos del código.
- La construcción del sistema se realiza sólo con los archivos en el repositorio.
- Las herramientas proveen diversos comandos y varían en su funcionalidad.

El proceso de codificación

Control del código fuente y construcción (“built”)

- Operaciones básicas:
  - Check out
  - Commit (o Check in)
  - Update
- Las herramientas mantienen la historia completa de los cambios y todas las versiones más viejas pueden recuperarse.
- El control del código fuente es una herramienta esencial para la coordinación y el desarrollo de grandes proyectos.

Refactorización

- Usualmente los códigos se modifican 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.
  i.e. la calidad y la productividad en la realización de cambios comienza a disminuir.
- La refactorización es una técnica para mejorar el diseño del código existente.
- Se realiza durante la codificación, pero el propósito no es agregar nuevas características sino mejorar el diseño.

Refactorización

- Usualmente los códigos se modifican 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 [illegible] el código y más susceptible a errores.
  i.e. la calidad y la productividad en la [illegible] comienza a disminuir.
- La refactorización es una técnica para mejorar [illegible]nte.
- Se realiza durante la codificación, pero el propósito no es agregar nuevas características sino mejorar el diseño.

El objetivo de la refactorización
no es corregir bugs.
Se aplica a código que ya está funcionando

Particularmente importante en XP y TDD

Refactorización

- 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.
  - La estructura interna del software cambia.
  - El comportamiento externo permanece igual.
- El objetivo básico es el de mejorar el diseño plasmado en el código (no es lo mismo que mejorar el diseño durante el proceso de diseño).

Refactorización

Conceptos básicos

- Dado que el fin es mejorar el diseño, la refactorización intenta lograr una o más de las siguientes cosas:
  reducir acoplamiento,
  incrementar cohesión,
  mejorar respuesta al principio abierto-cerrado.
- Cualquier modificación al código con el fin de llevar acabo lo anterior no debe cambiar la funcionalidad.
- La refactorización se realiza durante codificación y generalmente está asociada a un requerimiento de cambio.
- Sin embargo: no mezclar codificación normal con refactorización.
  Los cambios por refactorización se realizan separadamente de la codificación normal.

Refactorización

Conceptos básicos

- El principal riesgo de realizar una mejora al código, es que se puede "romper" la funcionalidad existente.
- Para disminuir esta posibilidad:
  - Refactorizar en pequeños pasos.
  - Disponer de scripts para tests automatizados para testear la funcionalidad existente.

Permite identificar los errores más fácilmente e identificarlos

Facilita la verificación de que si el comportamiento externo se preserva o no

- 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.

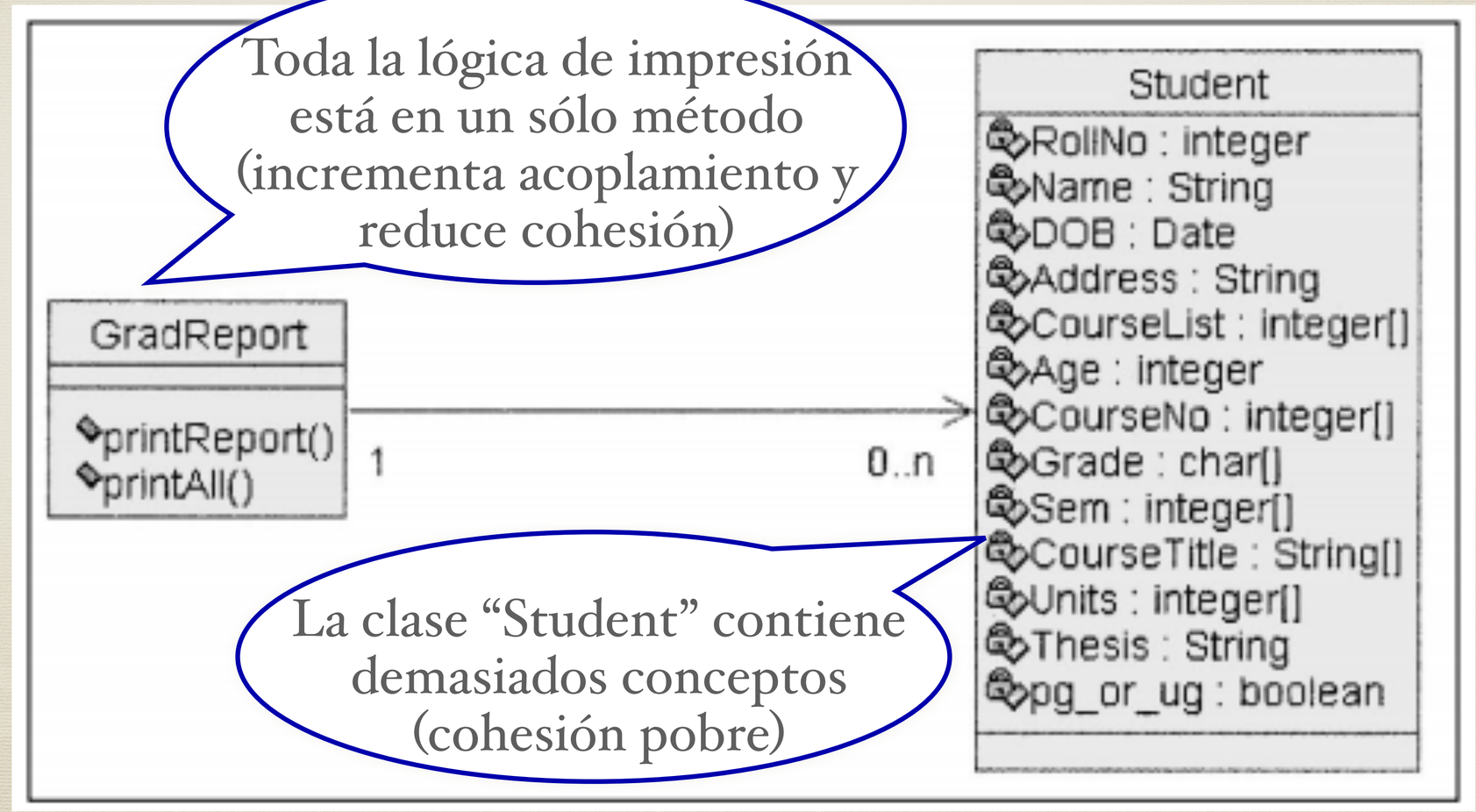
Refactorización

Un ejemplo

- Sistema para producir el certificado analítico de un alumno.
- El estudiante toma cursos y escribe la tesis.
- El sistema puede verificar si el alumno completó los requerimientos de graduación e imprime el resultado junto con las calificaciones.
- Un alumno puede ser de grado o de posgrado.
- Los requerimientos de graduación difieren para cada caso (en cantidad de cursos y requisitos de tesis).

Refactorización

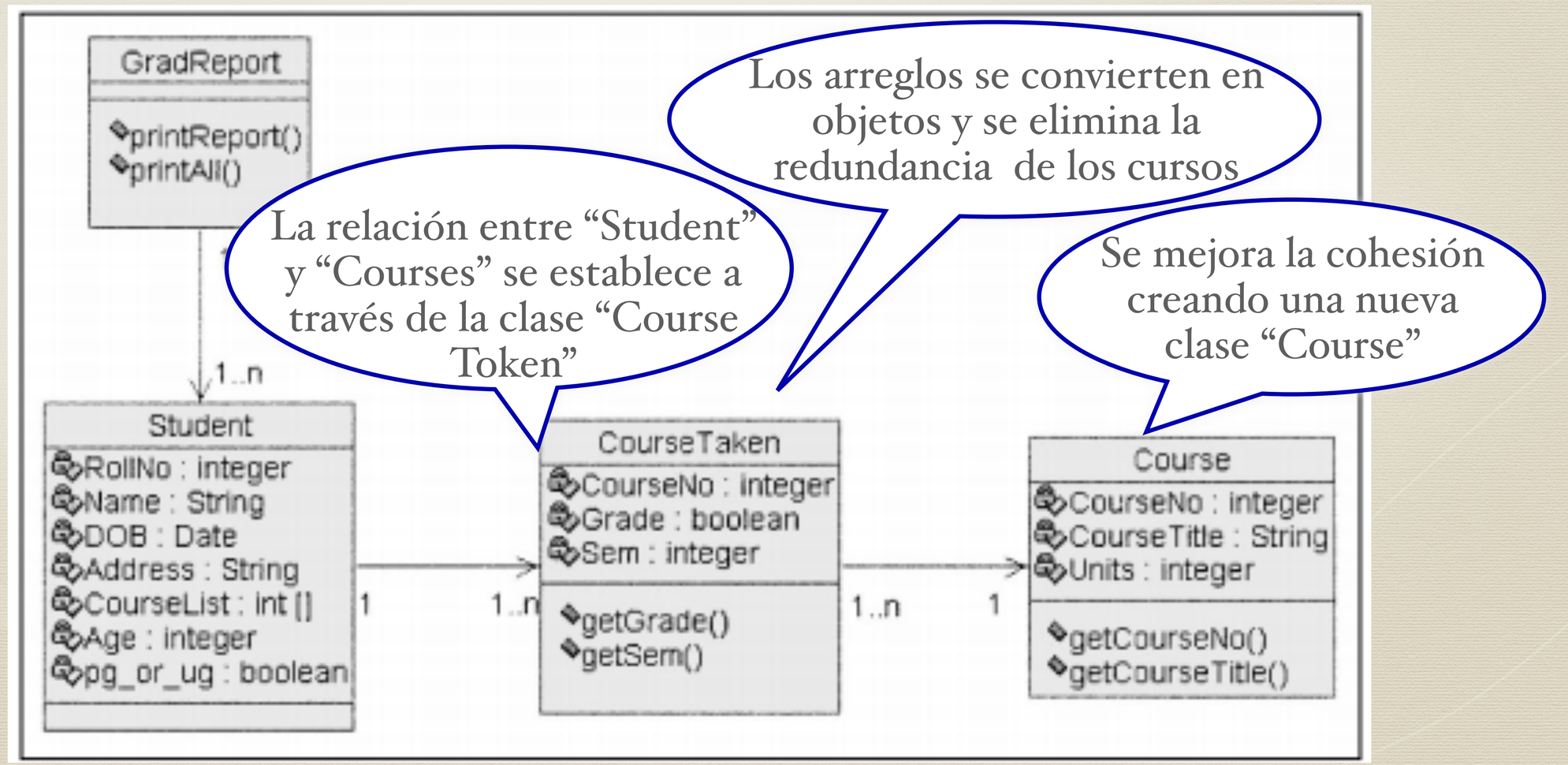
Un ejemplo



Refactorización

Un ejemplo

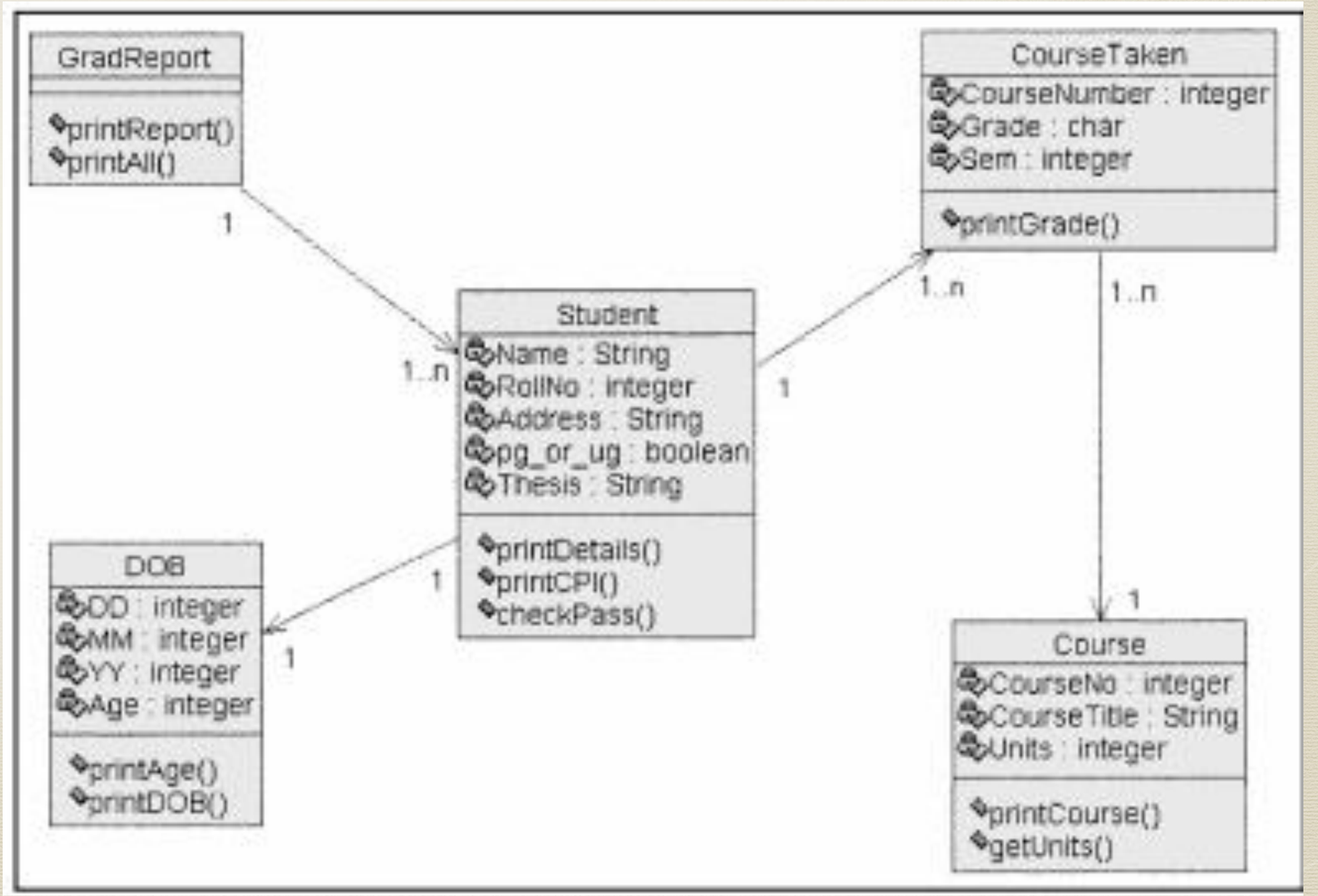
Primera refactorización



Refactorización

Un ejemplo

Segunda refactorización



Refactorización

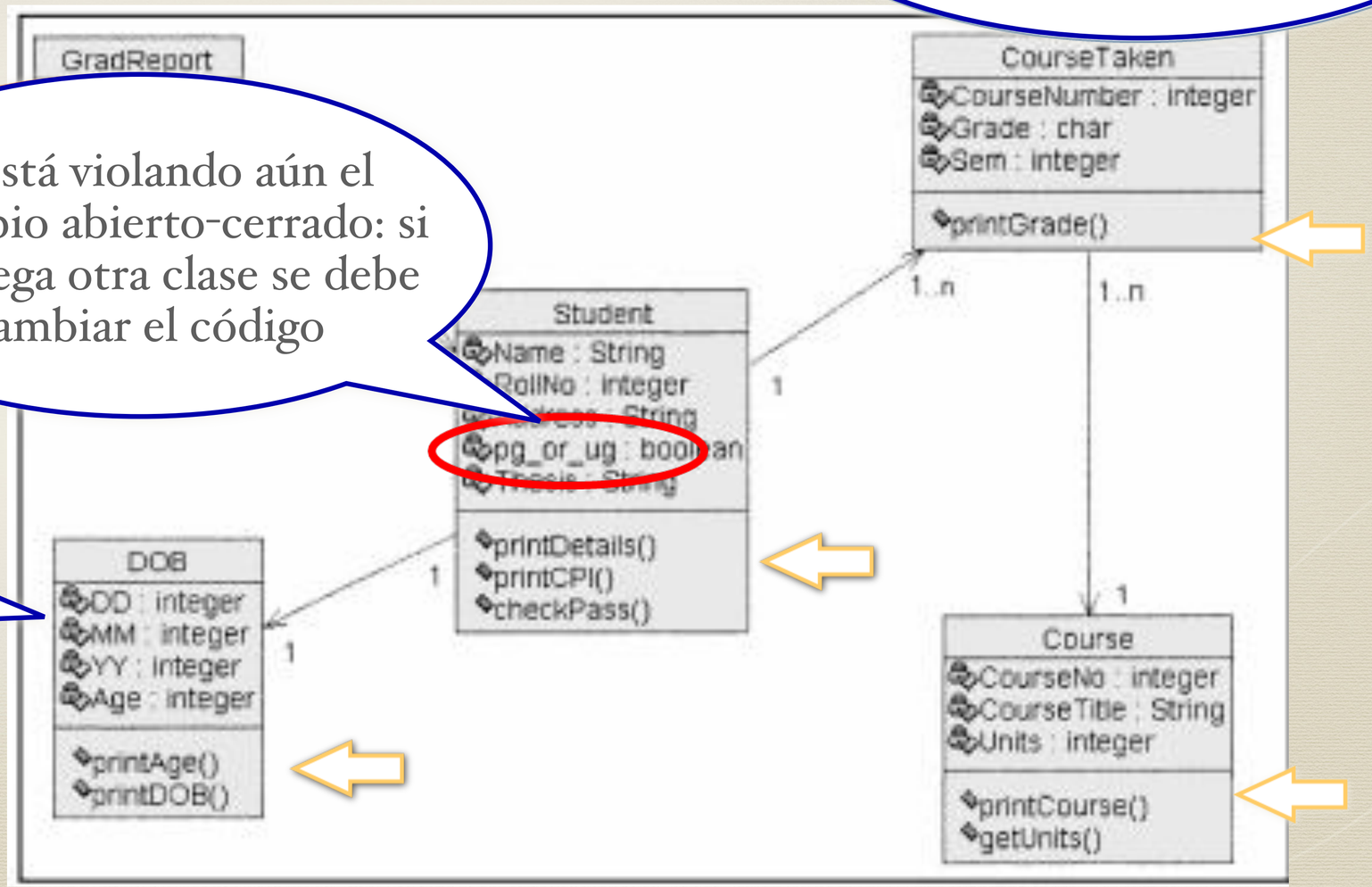
Un ejemplo

Segunda refactorización

La responsabilidad de la impresión se distribuyó a donde reside la información (reduce acoplamiento)

Se está violando aún el principio abierto-cerrado: si se agrega otra clase se debe cambiar el código

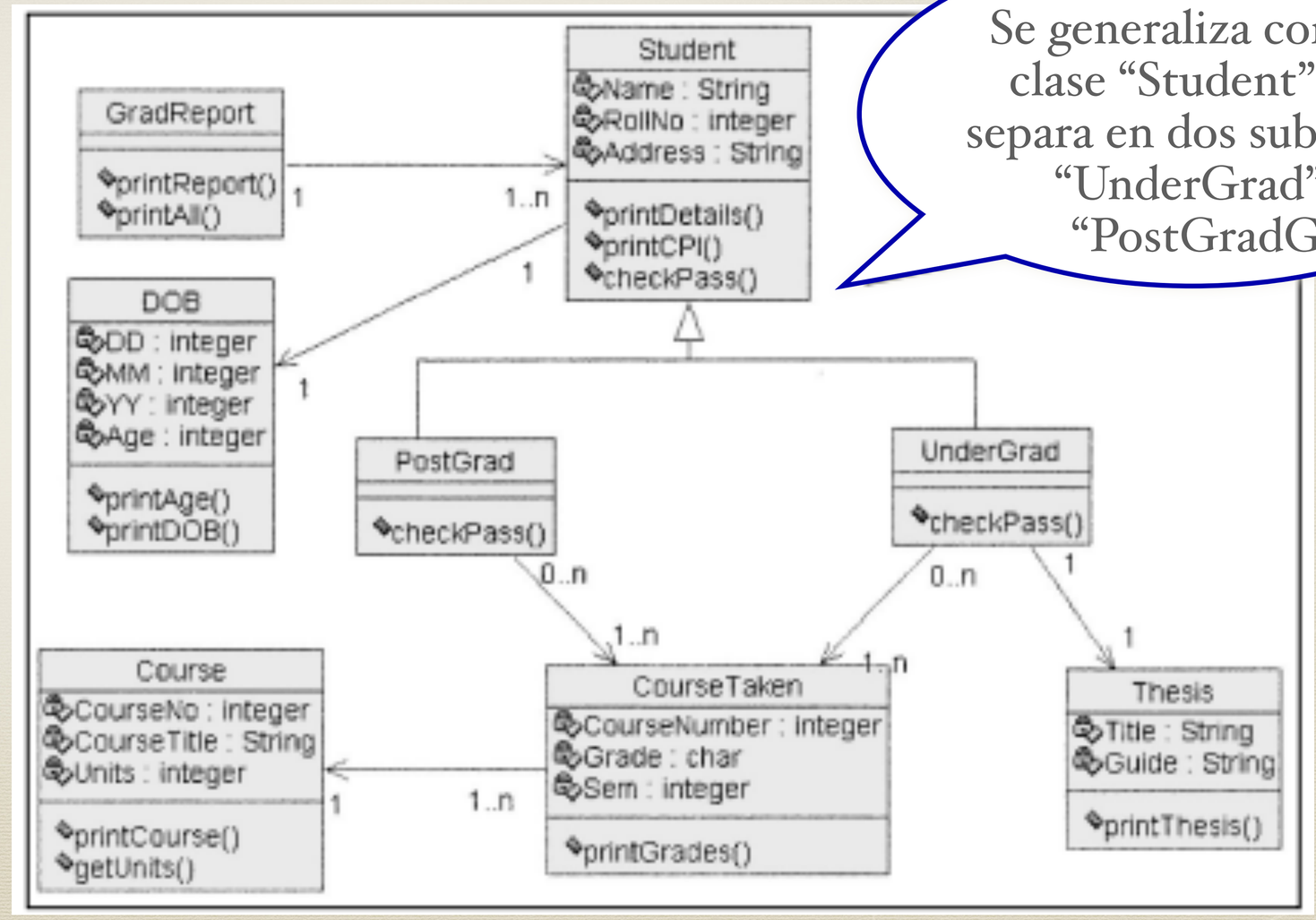
La clase “DOB” mejora cohesión y flexibilidad dado que puede reusarse



Refactorización

Un ejemplo

Refactorización final



Refactorización

Malos olores

Los “malos olores” son signos fáciles de localizar en el código que indican la posible necesidad de refactorización.

- No garantiza que sea realmente necesario: son sólo malos olores.
- Se necesita hacer análisis caso por caso.
- Posibles malos olores:
  - Código duplicado: es muy común - la misma funcionalidad aparece en lugares distintos; el cambio de esta funcionalidad será complicado.
  - Método largo: podría estar tratando de hacer demasiadas cosas.
  - Clase grande: Puede estar encapsulando muchos conceptos; puede no ser cohesiva.

Refactorización

Malos olores

- Posibles malos olores (continuación):
  - Lista larga de parámetros: las interfaces complejas son indeseables: reducir si es posible.
  - Sentencia “switch”: podría no estar usando herencia - si es así switch similares se repetirán en otros lugares. (Violación principio abierto-cerrado)
  - Generalidad especulativa: la subclase es la misma que la superclase; no hay razón aparente para esta jerarquía.
  - Demasiada comunicación entre objetos: las clases y/o los métodos pueden no ser cohesivos.
  - Encadenamiento de mensajes: un método llama a otro que llama a otro; posible acoplamiento innecesario.

Refactorización

Refactorizaciones más comunes

- Muchas formas de mejorar el diseño de programas.
- Existen catálogos que se extienden continuamente.
- Para mejorar el diseño se enfocan en
  - métodos,
  - clases,
  - jerarquía de clases.
- Siempre con el objetivo de mejorar acoplamiento, cohesión, y el principio de abierto-cerrado.

Refactorización

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.
  - Partes de código se extraen como nuevos métodos.
  - Variables referenciadas en esta partes se transforman en parámetros.
  - Variables declaradas en esta parte pero utilizadas en otras partes deben definirse en el método original.
  - También se realiza si un método retorna un valor y también cambia el estado del objeto. (Dividir en dos métodos).

Refactorización

Refactorizaciones más comunes - Mejoras de métodos

- Agregar/eliminar parámetros:
  - Para simplificar las interfaces donde sea posible.
  - Agregar sólo si los parámetros existentes no proveen la información que se necesita.
  - Eliminar si los parámetros se agregaron originalmente “por las dudas” pero no se utilizan.

Refactorización

Refactorizaciones más comunes - Mejoras de clases

- Desplazamiento de métodos:
  - Mover un método de una clase a otra.
  - Se realiza cuando el método actúa demasiado con los objetos de la 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 a esta clase.
  - Mejora cohesión y acoplamiento.

Refactorización

Refactorizaciones más comunes - Mejoras de clases

- Extracción de clases:
  - Si una clase agrupa múltiples conceptos, separa cada concepto en una clase distinta.
  - Mejora cohesión.
- Remplazar valores de datos por objetos:
  - Algunas veces, una colección de atributos se transforma en una entidad lógica.
  - Separarlos como una clase y definir objetos para accederlos.

Refactorización

Refactorizaciones más comunes - Mejoras de jerarquías

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

Verificación

- El código necesita verificarse antes de que sea utilizado por otros.
- En esta parte hablaremos sólo de la verificación del código escrito por un programador (la verificación del sistema queda para la unidad sobre testing).
- Distintas técnicas:
  - Inspección de código
  - Test de unidad
  - Verificación de programa

Verificación

Inspección de código

- El proceso de inspección es un proceso de revisión como cualquier otro.
- Se aplica luego de que el código fue compilado, testeado algunas veces, y chequeado con herramientas de análisis estático.
- El equipo de revisión se enfoca en encontrar defectos y bugs en el código.
- Son muy efectivos y ampliamente usados en la industria.
- También es caro: para código no crítico se puede usar sólo una persona en la inspección.
- Se utilizan listas de control para enfocar la atención.

Verificación

Inspección de código

Algunos ítems de la lista de control:

- ¿Todos los punteros apuntan a algún lado?
- ¿Se inicializaron todas las variables y punteros?
- ¿Los índices de los arreglos están dentro de sus cotas?
- ¿Terminan todos los loops?
- ¿Hay defectos de seguridad?
- ¿Se verificaron los datos de entrada?
- ¿Se satisfacen los estándares de codificación?

Verificación

Testing de unidad

- Es testing, sólo que se enfoca en el módulo escrito por un programador.
- Usualmente, el test de unidad (TU) lo realiza el mismo programador.
- El TU requiere casos de test para el módulo (la obtención de casos de test se explicará en la próxima unidad).
- El TU también requiere la escritura de “drivers” que ejecuten el módulo con los casos de test.
- Si se realiza codificación incremental, entonces el TU completo necesita automatizarse. (Si no sería demasiado tedioso la ejecución repetida de los TU).

Verificación
Testing de unidad

- Existen herramientas que auxilian al test:
  - Ellas proveen los drivers.
  - Se programan los casos de test, incluyendo la verificación del resultado, i.e. el TU es un script que retorna “pass” o “fail”.
  - Ej.: Junit

Verificación

Análisis estático

- Son herramientas para analizar los programas fuentes y verificar la existencia de problemas.
- Los analizadores estáticos no pueden encontrar todos los bugs y en ocasiones dan “falsos positivos”.
- Hay muchas herramientas disponibles que usan distintas técnicas.
- Son efectivas para encontrar bugs como: memory leaks, código muerto, punteros colgando, etc.
- Hay muchas herramientas.
- Otras herramientas relacionadas incluyen:
  - Model checkers (Banderas, JavaPathfinder, SLAM, Terminator, etc.).
  - Herramientas de análisis dinámico.

Verificación

Métodos formales

Lógica de Hoare, WP, técnicas de derivación, etc.

- Estos enfoques apuntan a demostrar la corrección de los programas.
- Es decir, a demostrar que el programa implementa la especificación dada.
- Requiere especificaciones formales para el programa, así como reglas para derivar las pruebas.
- Ha sido un área de investigación muy activa.
- La escalabilidad es el cuello de botella.
- Utilizado en software bajo situaciones críticas (ej.: software de seguridad crítica, misión crítica, etcétera).

Codificación

El buen código es invisible.

Codificación

Lectura complementaria:

Capítulo 9 Jalote

Clean Code by Robert C. Martin

Estándares:

GNU:http://www.gnu.org/prep/standards/

Linux Kernel:http://lxr.linux.no/source/Documentation/CodingStyle

Mozilla:http://www.mozilla.org/hacking/mozilla-style-guide.html

C++:https://en.wikibooks.org/wiki/C%2B%2B_Programming/Programming_Languages/C%2B%2B/Code/Style_Conventions

Java:http://java.sun.com/docs/codeconv/

Más en Wikipedia: http://en.wikipedia.org/wiki/Programming_style

Control del código fuente:

http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

Refactorización: http://www.refactoring.com/

Análisis estático: http://samate.nist.gov/index.php/Source_Code_Security_Analyzers