Código de Calidad

- No es objetivo, depende del cliente y la aplicación (criticidad, longevidad, etc)
- Principio nº 1: No repetir. Extraer métodos (privados), extraer clases, utilizar patrones
- Principio nº 2: Estandarizar formatos, comentarios, nombres, etc.
- Principio nº 3: No sorprender. Evitar lucirse y no explicar lo obvio.
- Usar espacios, sangrías, paréntesis y todo lo que pueda aclarar el código.

Nombres

- El código se escribe una vez, pero se lee muchas. Escribir en código para humanos,
- Escribir nombres descriptivos, en términos del problema, y no de la solución (evitar usar palabras como registro, vector, etc),

Métodos

- Evitar significados ocultos, como que un método devuelva -1 por alguna razón.
- Los nombres de los métodos describen qué hace, y no cómo lo hace.
- Ojo con el *case-sensitive*.
- Tratar de que los métodos no dependan de un orden de ejecución.
- Evitar la recursividad, salvo que ayude a la legibilidad.
- Que sólo afecte a la clase en la que está declarado.

Parámetros

- Limitar la cantidad de parámetros, y que estén ordenados por un criterio general.
- Siempre chequear los parámetros a la entrada.

Variables Locales

- Inicialización explícita.
- Evitar el doble uso.
- Que su vida sea lo más corta posible.
- Usar variables o métodos booleanos.
- Evitar *hardcodeo*.

Comentarios

- Son buenos, pero si necesitan explicar mucho, posiblemente se pueda mejorar el código.
- No deberían abundar, no explicar cosas obvias.
- Fáciles de entender, usar javadoc o NDoc si es posible.
- Comentar previo a un método qué es lo que hace.
- Aclarar sobre fuentes, efectos laterales, restricciones, invariantes.

Condiciones, ciclos

- Evitar la anidación profunda.
- Si se necesitan muchos if, usar switch.
- Los case deben estar ordenados de manera lógica (alfabética o numéricamente por ejemplo)
- Usar cada ciclo como corresponde.
- Cortar con break, return o una excepción si eso lo hace más claro.

Mejora de desempeño

- Resistir la tentación, no siempre da ganancias equivalentes a lo que cuesta.

- Evitar el uso de memoria externa.
- Realizar la menor cantidad de operaciones posibles dentro de un ciclo
- Ordenar los if de manera que primero se prueben las cosas más simples. Las más frecuentes y menos costosas.
- Usar enteros en vez de punto flotante.
- Luego de mejorar, probar que realmente mejoró (a veces hacemos cosas que el IDE o el compilador ya hacían automáticamente).
- Asignar null al finalizar de usar una referencia, y hacerlo al comienzo si no tiene valor.
- Hay herramientas para algunas de estas mejoras. Muchas vienen con los IDE.

Buenas prácticas

- Utilizar TDD, hacer pruebas unitarias y funcionales todo el tiempo.
- Hacer pruebas de integración.
- Codificar pensando en el cambio.
- Recordar: el cliente pide (requerimientos), no necesariamente lo que quiere (expectativas), ni mucho menos lo que necesita (necesidad).
- Es importante tener comunicación frecuente con el cliente.
- Simplicidad. YAGNI (*You ain't gonna need it*)

Mandamientos del alumno de Algoritmos III

- No repetirás código.
- No dejarás a tu prójimo lo que no quieres que te dejen a tí.
- Escribirás código legible.
- Usarás la barra espaciadora y el salto de línea en tu teclado.
- *"Premature optimization is the root of all evil"*.

Diseño MVC

- El diseño es el "cómo" del desarrollo. Es concebir una solución para un problema.
- Diseñamos: Despliegue en hardware. Subsistemas o componentes en software y sus interacciones. Integración con otros sistemas. Interfaz de usuario. Estructuras de datos. Algoritmos. Definiciones tecnológicas (plataforma, lenguaje, base de datos, etc)
- Principio número 1: Estar preparado para el cambio.

Patrones

- Solución exitosa de un problema que se presenta con frecuencia.
- Patrones de diseño
- Patrones de programación

MVC

- Separa incumbencias.
- Facilita cambios.
- Es bueno en aplicaciones con mucha UI (muchas vistas para un mismo modelo)
- Módulos de alta cohesión (Intrínseca, respecto al mismo módulo)
- Módulos de bajo acomplamiento (con otros módulos). Acota los efectos de las modificaciones.

Cohesión y acoplamiento en clases

- La clase debe representar a una sola entidad (un sustantivo la debe poder describir) (alta cohesión)
- Debe tener pocas dependencias con otras clases. Acota las posibles modificaciones. (bajo acoplamiento)
- Orden de dependencia entre clases, de mayor a menor: Herencia, composición, asociación simple, dependencia débil.

Cohesión y acoplamiento en paquetes

- Los paquetes deben describir una visión global del sistema. Agrupar clases relacionadas entre sí para lograr alta choesión.
- Deben tener pocas dependencias con otros paquetes. Para lograr bajo acoplamiento.

Cohesión y acoplamiento en métodos

- Cada método se describe con una oración con sólo un verbo (alta cohesión)
- Debe tener pocos parámetros y pocas llamadas a otros métodos (bajo acoplamiento)

Razones para crear clases

- Modelar entidad del dominio.
- Modelar entidad abstracta (clases abstractas o interfaces). Una clase que reunen las cosas que otras clases tienen en común.
- Reducir complejidad.
- Agrupar operaciones relacionadas (ejemplo: Math, en Java)
- Aislar complejidad y detalles de implementación (ListaClientes vs ArrayList<Clientes>)
- Aislar código que se espera que cambie.
- Si una clase descendiente no añade o redefine un método, no es necesaria.

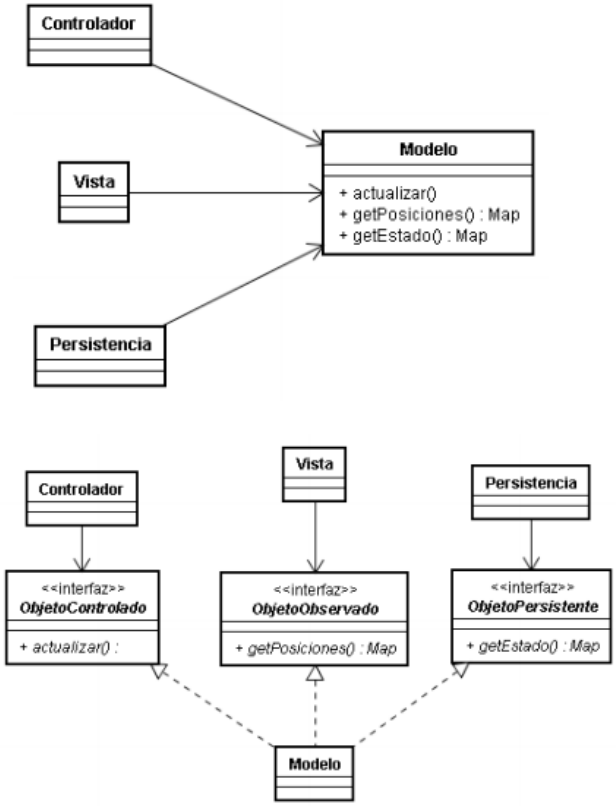
Razones para crear métodos

- Refactorización (Reducir complejidad, hacer legible el código, simplificar tests booleanos, evitar duplicación de código, mejorar choesión, encapsulamiento).
- Si no son necesarios para el cliente de la clase, son privados.
- Comentarios que indican que un método se podría separar: If anidados, ciclos, comentarios que separan cosas.
- Evitar efectos laterales.
- Actuar solamente sobre el objeto con el que es llamado.
- Las funciones deben tener nombres que describan el valor que devuelven.
- Los procedimientos deben tener nombres que describan qué hacen (un verbo)

Principios

- Única responsabilidad: Relacionado con la alta cohesión. Cada clase debe tener una única razón para poder cambiar. Encapsular lo que varía.
- Abierto-Cerrado: Las clases deben estar cerradas para modificación, pero abiertas para reutilización. No modificar las clases existentes. Extenderlas o adaptarlas por herencia o delegación.
- Delegación sobre herencia: La delegación otorga más flexibilidad.
- Sustitución: Los subtipos deben ser sustituíbles en todas partes por sus tipos base. Las clases base no deben tener comportamientos que dependan de las clases derivadas. (Ej: Círculo/Elipse).
- Inversión de dependencia: No depender de clases concretas volátiles, hacerlo de clases abstractas o interfaces. “Programe contra interfaces, no implementaciones”.
- Polimorfismo: Evitar condicionales.

- Segregación de la interfaz: Si una clase tiene una referencia a, o hereda de, otra clase, de la cual sólo tiene sentido que utilice algunos de sus métodos, pero no todos, lo mejor sería separar la clase en cuestión en más de una.



Visibilidad

- Todo debe ser lo más privado posible. Garantiza poder modificar código que no afecte al cliente de la clase.
- Atributos privados.

Entropía y mejora del diseño

- Todo código empeora con el tiempo
- Refactorizar: Mejorar código haciéndolo más comprensible, sin cambiar funcionalidad.
- Se debe realizar de maner constante. Antes de modificar código, tras incorporar funcionalidad,antes de optimizar. Asegurarse que tanto antes, como después, el código funcione de la misma manera (una forma, pruebas).
- Modificar sólo un paso a la vez. Probando en el medio.

Mandamientos del alumno de Algoritmos III

- Mantendrás bajo el acoplamiento.
- Privilegiarás la delegación sobre la herencia.
- Preferirás interfaces a clases concretas.
- Darás la menor visibilidad posible a los clientes de tus clases.
- Mantendrás la entropía bajo control.

Patrones

State

- El compotamiento del objeto cambia dependiendo del estado del mismo.

Singleton

- Consiste en restringir la creación de objetos pertenecientes a una clase. Su objetivo es que una clase tenga sólo una instancia y proporcionar un punto de acceso global a ella.
- Una de las claves es crear el constructor de la clase, como privado. Evitando así que se utilice el constructor por defecto.

Composite

- Quiero tratar a un objeto y a una colección de objetos, con la misma interfaz. Pensar en el ejemplo de una ventana. Puede tener objetos, y uno de ellos, ser una colección de objetos.

Double Dispatch

- El ejemplo del cuerpo celeste.

RTTI y reflexión. Modelo de datos.

RTTI (Run-Time Type Information)

- Cuando se crea un objeto, se guarda en un atributo de Object, una referencia a un objeto Class. Ese objeto se crea al compilar la clase.
- Clases: Method, Contructor, Field
- Métodos: getMethods(), getConstructors(), getFields(), getInterfaces(), getSuperclass().
- En clase Field: get(), set()
- En clase Method: invoke()
- Crear un objeto cuya clase se sabe en tiempo de ejecución, ejemplo:

```
Serializable f = x.crearObjeto();// Serializable es una interfaz
```

El método sería:

```
public Serializable crearObjeto() {
      String nombreClase = leerArchivoConfiguracion();
      Class claseInstanciar = Class.forName(nombreClase);
      Object nuevo = claseInstanciar.newInstance();
      return (Serializable)nuevo;
}
```

- Ya usamos reflexión en Junit. Usa polimorfimos en setUp() y tearDown(), y reflexión en los métodos public void testXxx().

Autoboxing

- `int x = 4; Integer i = x; // equivale a Integer I = new Integer(x);`
- `inty = (int) i;// equivale a int y = i.intValue( );`

Modelo de memoria en Java

- Comparaciones:
  - == compara referencias.
  - `a.equals(b)` compara contenido. Está definido en object, implementado en colecciones, no hace comparaciones profundas.
- Asignaciones:
  - `Object a = b; // asigna referencias`
  - `Object o,b; o.m(b); // b pasa como referencia`
  - `int x = y; // asigna valores.`

- `int y; Object o; o.m(y); // y pasa como valor.`
- `A = b.clone();` Hay que redefinir clone como público y que llame a super.clone(), si quiero copia profunda hay que implementarla, y se debe implementar Cloneable.

Recolección de basura

- Asegura que no me voy a quedar sin meoria mientras haya objetos sin referencias. Y que no va a liberar ningún objeto que esté siendo referenciado desde un objeto referenciado.
- Es cómoda y evita errores difícil de encontrar.

Finalize()

- No es un destructor, se ejecuta cuando pasa el recolector de basura, que no sabemos cuádo es, y puede que nunca (si no es necesario).

Claves

- Todo objeto conoce su tipo, y es permanente: RTTI.
- Se le puede preguntar de todo a un objeto: reflexión.
- Usar RTTI y reflexión con medida.
- La clonación y el equals() solucionan problemas de usar referencias.