Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

Departamento de Computación
Diseño y Arquitectura de Software
Profesora Marlene O. Sánchez Escobar

“Cheat Sheet - Malos Olores”

Ángel Heredia Vázquez
A01650574

Samuel Kareem Cueto González
A01656120

Carlos Andrés Conde Besil
A01650549

Javier Arturo Flores Zavala
A01651678

José Carlos Acosta García
A01650306

Cheat Sheet - Malos Olores

Código duplicado
Si se encuentra la misma estructura, métodos y variables en distintas partes del código se debe realizar una extracción de métodos.

Método largo
Entre más largo sea un procedimiento, más difícil de entender es. Si se nombra correctamente un método, no se tiene que ver su contenido. La regla es, si sentimos la necesidad de escribir un comentario, es mejor crear otro método. De esta forma el nombre del método describe su acción. Nuevamente, lo que debe hacerse es la extracción de método, descomposición de condicionales, etc.

Large class
Cuando una clase realiza muchas funciones, cuenta con muchas variables y no utiliza todas. Suele provocar código duplicado. La solución para esto es agrupar los métodos y variables semánticamente y extraer una clase.

Lista larga de parámetros
Antes se solía enviar una larga lista de parámetros. Sin embargo, la programación orientada a objetos facilita este proceso pues se suele pasar solamente el objeto, por lo que basta con un par de consultas para darle a los métodos los datos que requieren para su funcionamiento. Para solucionarlo se utilizan las técnicas: reemplazar parámetro por métodos y preservar el objeto completo.

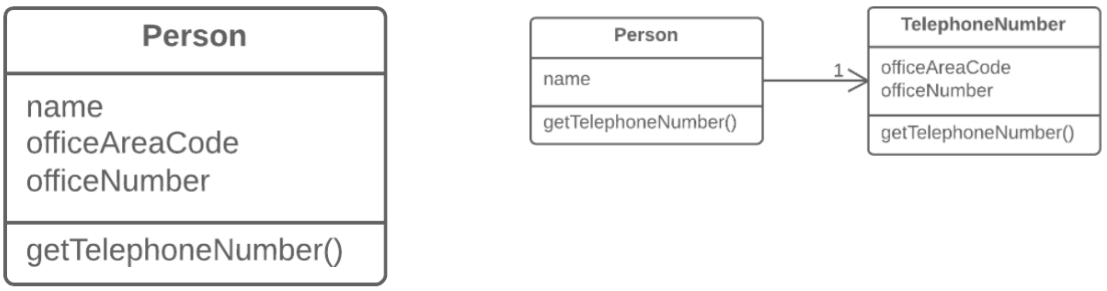
Cambio divergente
Este tipo de cambio ocurre por ejemplo cuando se tienen que cambiar 3 cosas distintas para poner otra base de datos, o agregar un instrumento financiero. Cualquier cambio o variación debe hacerse sobre una misma clase y esa clase debe reflejar todo lo introducido. Para lograrlo, se utiliza el método de extracción de clase.

Obsesión primitiva
Es preferible contar con objetos personalizados para campos como la dirección, que incluye calle, colonia, código postal. En lugar de tener en la misma clase cada atributo con datos primitivos. Se resuelve al reemplazar valores y datos con un objeto.

Lazy Class
Cada clase creada cuesta dinero mantenerla. Es por eso que una clase que no tenga una función relevante definida, debe ser eliminada.

Comentarios
Los comentarios suelen encubrir algún mal olor. Cada vez que haya un comentario, puede ser necesario refactorizar y utilizar un método que resulta autodescriptivo.

Técnica: Extracción de métodos, campos en clases.
¿Qué problema resuelve? Resuelve el problema de clases largas que hacen el trabajo de varias clases, la obsesión primitiva.
¿Cómo se lleva a cabo? Se crea una nueva clase donde se colocan los métodos y campos correspondientes.
Resultado: Clases sencillas, apegadas al principio de responsabilidad única, fáciles de entender.
Ejemplo:



Se tiene una clase de persona que contiene la lada y el número. Es decir, dos campos que están relacionados entre sí. Este método consiste en crear una nueva clase de número telefónico donde se colocan estos datos con sus correspondientes getters y setters.

Otro ejemplo es si la persona tiene su cumpleaños, está compuesto por día, mes y año. En lugar de guardar la información en un string con el formato dd/mm/yyyy, se extrae en una nueva clase. En términos generales se recomienda tener objetos personalizados, en lugar de contar con un exceso de datos primitivos.

Técnica: Simplificación de condicionales, decompose conditional (extraer a métodos)
¿Qué problema resuelve? Condicionales muy complejos, difíciles de entender, métodos largos.
¿Cómo se lleva a cabo? Descomponer las partes complicadas en métodos, de forma que se haga más legible. Implementar guard clauses (if sin else).
Resultado: Código condicional extraído a métodos bien nombrados, fáciles de entender,
Ejemplo:

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

Se tiene que calcular el cargo a realizar. Sin embargo, en lugar de validar muchas condiciones dentro de los ifs y hacer cálculos complejos, se colocan métodos con nombre específico que permiten entender fácilmente lo que el código está haciendo. Otro aspecto es que se suelen colocar en el orden en el que más frecuente sea su aparición.

Técnica: Simplificación de condicionales con polimorfismo
¿Qué problema resuelve? Cuando se tiene un condicional que ejecuta acciones dependiendo del tipo de objeto o propiedades, switch.
¿Cómo se lleva a cabo? Se elimina el switch y se crean clases que coinciden con las ramas de los condicionales. Por lo que se sustituye el condicional llamando a un método con polimorfismo.
Resultado: El resultado es un código más limpio sin código duplicado, que sigue el principio Open/closed; ya que si se requiere añadir una nueva variante, basta con agregar otra clase sin tocar lo existente.
Ejemplo:

```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCocc
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}

// Somewhere in client code
speed = bird.getSpeed();
```

Se tiene un switch donde se calcula la velocidad del pájaro basado en el tipo del que se trate. Se sustituye el switch, colocando una clase por cada tipo de pájaro, que extiende de la clase genérica. De modo que al llamar el método de getSpeed, cada clase sabe cómo calcularlo y entregar el valor correcto.

Técnica: Reemplazar condicionales anidados por guard clauses.
¿Qué problema resuelve? Condicionales complejos, anidados. Dificultad para determinar el flujo de ejecución del código.
¿Cómo se lleva a cabo? Aislar cada uno de los casos y colocarlos idealmente como una lista plana de condicionales, uno tras otro.
Resultado: Estructura de condicional plana. Código fácil de entender.
Ejemplo:

```
public double getPayAmount() {
  double result;
  if (isDead){
    result = deadAmount();
  }
  else {
    if (isSeparated){
      result = separatedAmount();
    }
    else {
      if (isRetired){
        result = retiredAmount();
      }
      else{
        result = normalPayAmount();
      }
    }
  }
  return result;
}
```

```
public double getPayAmount() {
  if (isDead){
    return deadAmount();
  }
  if (isSeparated){
    return separatedAmount();
  }
  if (isRetired){
    return retiredAmount();
  }
  return normalPayAmount();
}
```

Se cuenta con un sistema de pagos que calcula el monto a pagar de acuerdo con ciertas condiciones como si la persona está retirada, separada, entre otros. Se sustituye los if anidados, es decir la estructura compleja e indentada, por una estructura plana con if en forma de lista.

Técnica: Reemplazar condicionales con strategy pattern
¿Qué problema resuelve? Tener que realizar cambios sobre las clases, pues es mala práctica. Condicionales complejos
¿Cómo se lleva a cabo? El strategy pattern consiste en cambiar dinámicamente los algoritmos que usan objetos en el tiempo de ejecución. Elimina condicionales. Se crean subclases por cada algoritmo.
Resultado: Un programa que permite cambiar y agregar funcionalidad sin modificar las clases principales. Poco desacoplado.
Ejemplo:

```
class NoBonus implements Pay{
    public double getPay(double salary) {
        return salary;
    }
}
class Bonus20Per implements Pay{
    public double getPay(double salary) {
        return salary + (salary * .20);
    }
}
```

Tenemos un sistema de pago para empleados donde cada uno de los empleados tiene su salario y un bonus que corresponde a un porcentaje sobre el mismo. Al aplicar este método se

reduce la complejidad de los condicionales al no tener que comprobar tanto el salario, como el bonus de cada uno de los empleados para determinar el monto del pago.

Técnica: Reemplazar parámetro por método explícito
¿Qué problema resuelve? Un método está dividido en partes donde cada uno se ejecuta dependiendo del valor de un parámetro.
¿Cómo se lleva a cabo? Extraer las partes individuales del método y llamarlas en lugar del método original.
Resultado: Se logra reducir los métodos dependientes de parámetros que han crecido mucho. Código que no es trivial se corre en ramas. Es más fácil leer y entender el código.
Ejemplo:

```
void setValue(String name, int value) {
  if (name.equals("height")) {
    height = value;
    return;
  }
  if (name.equals("width")) {
    width = value;
    return;
  }
  Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
  height = arg;
}
void setWidth(int arg) {
  width = arg;
}
```

Es más fácil leer y entender startEngine(), en lugar de setValue(“engineEnabled”, true).

Técnica: Preservar el objeto completo
¿Qué problema resuelve? Obtener varios valores de un objeto y pasarlos como parámetros a un método.
¿Cómo se lleva a cabo? En lugar de separar los valores y pasar cada uno individualmente a un método, se pasa el objeto completo.
Resultado: Todos los valores que se necesitan están agrupados en el mismo lugar. En lugar de una serie de valores, se ve más fácil un objeto con nombre claro que contiene los valores necesarios. Si un método requiere datos de un objeto no se tiene que reescribir todos los lugares donde el método se utiliza.
Ejemplo:

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```
boolean withinPlan = plan.withinRange(daysTempRange);
```

Referencias

Banas, D. (2013). Code Refactoring 4. Recuperado el 2 de mayo de 2022 de: https://www.youtube.com/watch?v=BsJlP-X0WVg

Banas, D. (2013). Code Refactoring 6. Recuperado el 2 de mayo de 2022 de: https://www.youtube.com/watch?v=_7PI2LYAx-g

Banas, D. (2013). Code Refactoring 7. Recuperado el 2 de mayo de 2022 de: https://www.youtube.com/watch?v=owss5KuioFs

Refactoring Guru. (2020). Extract class. Recuperado el 2 de mayo de 2022 de: https://refactoring.guru/extract-class

Refactoring Guru. (2020). Decompose conditional. Recuperado el 2 de mayo de 2022 de: https://refactoring.guru/decompose-conditional

Refactoring Guru. (2020). Replace nested conditionals with guard clauses. Recuperado el 3 de mayo de 2022 de: https://refactoring.guru/replace-nested-conditional-with-guard-clauses

Refactoring Guru. (2020). Replace parameters with explicit methods. Recuperado el 3 de mayo de 2022 de: https://refactoring.guru/replace-parameter-with-explicit-methods

Refactoring Guru. (2020). Preserve whole object. Recuperado el 3 de mayo de 2022 de: https://refactoring.guru/preserve-whole-object