

Refactoring

```
public abstract class AbstractCollection implements Collection {  
    public void addAll(AbstractCollection c) {  
        if (c instanceof Set) {  
            Set s = (Set)c;  
            for (int i=0; i < s.size(); i++) {  
                if (!contains(s.getElementAt(i))) {  
                    add(s.getElementAt(i));  
                }  
            }  
        } else if (c instanceof List) {  
            List l = (List)c;  
            for (int i=0; i < l.size(); i++) {  
                if (!contains(l.get(i))) {  
                    add(l.get(i));  
                }  
            }  
        } else if (c instanceof Map) {  
            Map m = (Map)c;  
            for (int i=0; i<m.size(); i++)  
                add(m.keys[i], m.values[i]);  
        }  
    }  
}
```

The diagram illustrates several refactoring opportunities in the provided Java code snippet:

- Duplicated Code** (Blue label): Points to the `if (!contains(s.getElementAt(i)))` condition in the Set branch and the `if (!contains(l.get(i)))` condition in the List branch, which perform identical logic.
- Duplicated Code** (Purple label): Points to the `add(s.getElementAt(i))` statement in the Set branch and the `add(l.get(i))` statement in the List branch.
- Alternative Classes with Different Interfaces** (Green label): Points to the `if (!contains(l.get(i)))` condition in the List branch, highlighting the use of `l.get(i)` instead of `s.getElementAt(i)`.
- Switch Statement** (Red label): Points to the `if` statements in the `addAll` method, suggesting a `switch` statement for better readability.
- Inappropriate Intimacy** (Red label): Points to the `m.keys[i]` and `m.values[i]` access in the Map branch, indicating that the Map interface does not provide indexed access to keys and values.
- Long Method** (Black label): Points to the `addAll` method, suggesting it might be too long and could be refactored into smaller methods.



Refactorización (Refactoring)

- Técnica de Ing de Software para optimizar un código escrito.
- Se hacen cambios internos que no deben afectar al funcionamiento externo.
- Se mejora el código sin cambiar funcionalidad
 - NO se añade funcionalidad: Eso es desarrollar
 - NO se arreglan errores: Eso es debug
- El objetivo es código limpio y modularizado

```
if (isArray(model, val)) {  
  var val = self.getValue(  
    if (el.checked) {  
      if (indexOf(model, val)  
        model.push(val);  
    }  
  } else {  
    model.$remove(val);  
  }
```

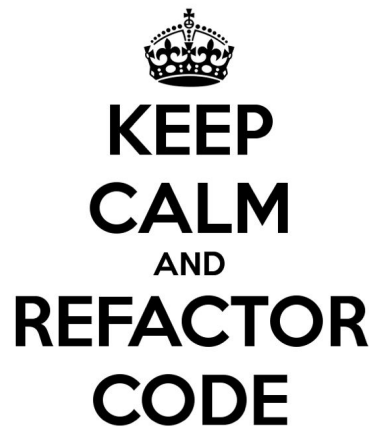
Refactorizar



- Se mejora el código para facilitar nuevos desarrollos futuros simplificando el código que ya existe permitiendo que otras personas lo comprendan con más facilidad
- En la fase de refactorización es donde realmente aplicamos todos los aspectos vistos de código limpio: aumentar cohesión, dividir clases, modularizar aspectos, elegir nombres adecuados...
- Es muy difícil hacer un código limpio a la primera, así que es necesario refactorizar después de que el código funcione, y es necesario hacerlo antes de pasar a la siguiente tarea.

Ventajas de Refactorizar

- Revisar diseño del sistema
- Plantearse cambios en el futuro
- Incremento de facilidad de lectura del código
- Mejor comprensión del código
- Detección y solución temprana de fallos
- Mayor velocidad de desarrollo
- No todo se puede refactorizar:
 - Bases de datos: No sería refactorizar, sino migrar estructura y datos



¿Cuándo refactorizar?: Malos olores (Bad Smells)

- Código duplicado (duplicated code):
 - si el mismo código está en más de un sitio, se mete en una función que se llama desde dos sitios: Aplicable a todo desde CSS en adelante
- Métodos muy largos (Long method):
 - Cuanto más largo es un método, más difícil es de entender.
 - Seguramente se pueda subdividir
 - Cuanto más corto es, más posibilidades tiene de reutilización.
- Lista de parámetros muy larga (Long parameter list):
 - Más de dos o tres parámetros son muchos
 - Se debe encapsular en clases toda esa información
 - Se reconoce porque hay parámetros que siempre van juntos => tienen entidad común



¿Cuándo refactorizar?: Malos olores (Bad Smells)

- Clases muy grandes (Large class):
 - Una clase con demasiados métodos o atributos
 - Tendrá demasiadas funcionalidades
 - Subdividir en clases, herencia...
- Cambio divergente (Divergent change):
 - Una clase que empezó siendo de una manera y ahora no la reconoce ni su madre
 - La clase no cumple su propósito original o incluso ningún propósito: obsoleta, desfasada
 - La clase se debe subdividir o eliminar
- Cirugía con escopeta (Shotgun surgery):
 - Si al modificar una clase hay que cambiar código en otros sitios
 - Mala granularización



¿Cuándo refactorizar?: Malos olores (Bad Smells)

- Envidia de funcionalidad (Feature envy):
 - Un método de una clase que utiliza muchos datos de otras clases
 - El método se debe mover a la clase que tanto usa y pasar la clase en la que está declarado como parámetro.
- Clase de sólo datos (Data class):
 - Una clase que sólo tiene atributos y métodos de acceso get/set
 - No tienen comportamiento
 - Se puede estudiar mover el contenido a otra clase
 - No siempre
- Herencia rechazada (Refused bequest):
 - Clases hija que no utilizan casi nada la clase padre
 - La herencia está mal diseñada

