Más Técnicas de Refactorización

Semana 12

Agenda

- Grupos de técnicas de refactorización
  - Simplificando llamadas a métodos
  - Organizando datos

Grupos de técnicas de refactorización

Grupos de técnicas de refactorización

- Clase anterior:
  1. Composing methods
  2. Simplifying Conditional Expressions
  3. Dealing with Generalization
  4. Moving Features between Objects
- Clase actual:
  5. Simplificando llamadas a métodos
  6. Organizando datos

Composing methods

- Extract Method
- Inline Method
- Split Temporary Variable
- Extract Variable
- Inline Temp
- Replace Temp with Query
- Replace Method with Method Object
- Substitute Algorithm

Simplifying Conditional Expressions

- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Introduce Assertion
- Consolidate Conditional Expression
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polymorphism
- Introduce Null Object

Dealing with Generalization

- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Field
- Push Down Method
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

Moving Features between Objects

- **Move Method**
- **Move Field**
- **Extract Class**
- **Inline Class**
- **Hide Delegate**
- **Remove Middle Man**
- **Introduce Foreign Method**
- **Introduce Local Extension**

Simplificando llamadas a métodos

Simplificando llamadas a Métodos

- Estas técnicas pretenden hacer que las llamadas a métodos sean sencillas y fáciles de entender.
- De esta forma se simplifica la interacción entre clases.

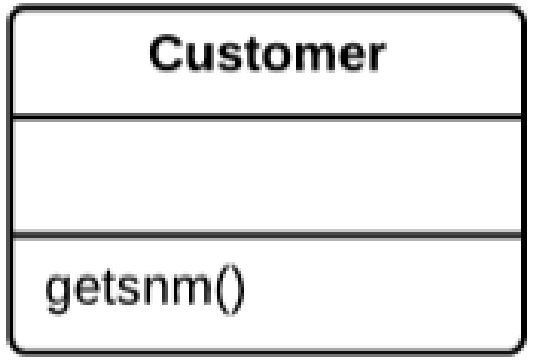
Técnicas

- Renombrar Método
- Remover Parámetro
- Separar Query de Modificador
- Parametrizar Método
- Reemplazar Parámetro con Métodos Explícitos
- Preservar todo el Objeto
- Reemplazar Parámetro con llamada a Método
- Ocultar Método
- Reemplazar Código de error con Excepción
- Reemplazar Excepción con Test

Renombrar Método

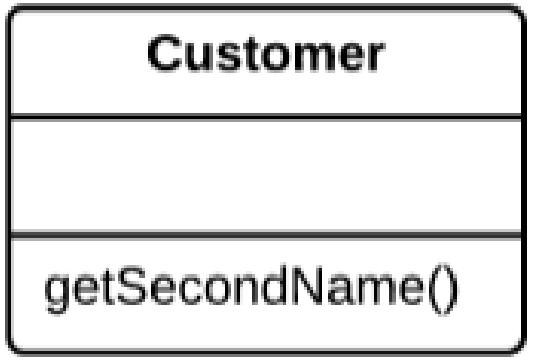
Problema

El nombre de un método no sugiere lo que realmente hace.



Solución

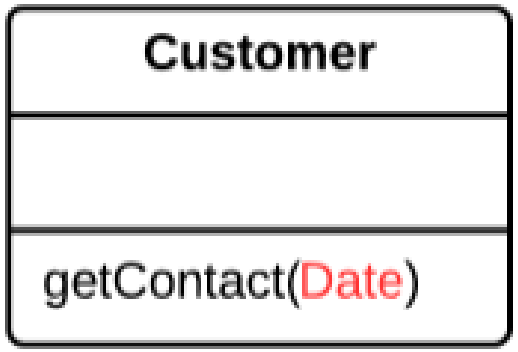
Renombrar el método, teniendo cuidado de reemplazar su nombre en todos los lugares donde se lo llama.



Remover Parámetro

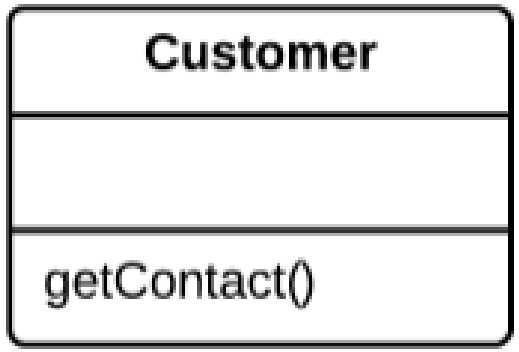
Problema

Un parámetro no es usado en la implementación del método.



Solución

Remover el parámetro que no se usa.



Separar Query de Modificador

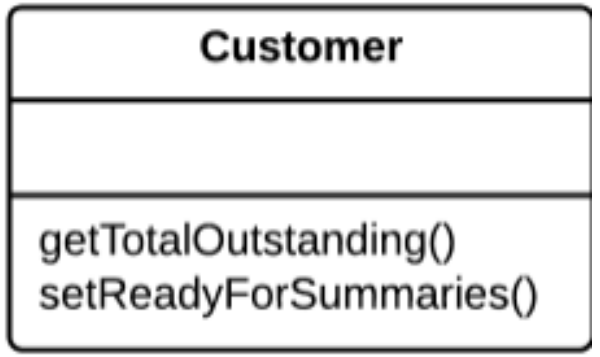
Problema

Se tiene un método que retorna un valor, pero también realiza la actualización de un valor.

```
public int setWins() {
    int winTotal = wins++;
    return winTotal;
}
```

Solución

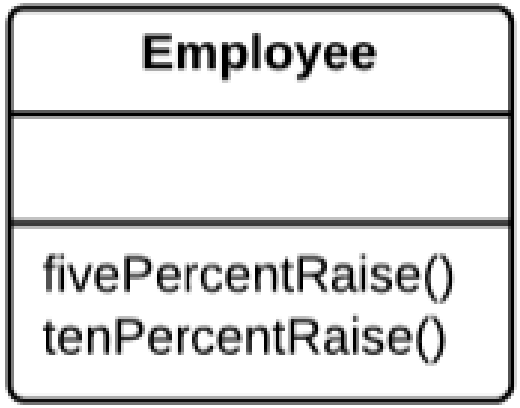
Dividir el método en dos.



Parametrizar Método

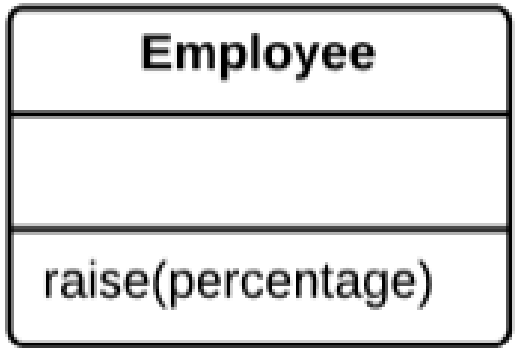
Problema

Múltiples métodos realizan distintas acciones que solo difieren en sus valores internos.



Solución

Combinar esos métodos y usar un parámetro que envíe a la función el valor especial.



Reemplazar Parámetro con Métodos Explícitos

Problema

Un método está dividido en varias partes, cada una funciona dependiendo del parámetro.

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

Solución

Extraer las partes individuales del método. Llamarlos en las partes adecuadas.

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

Preservar todo el Objeto

Problema

Se extraen varios valores de un objetos y luego se los envía como argumentos a un método.

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

Solución

Enviar todo el objeto y acceder a los valores necesarios dentro del método.

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

Reemplazar Parámetro con llamada a Métodos

Problema

Se invoca a un método pasando como argumentos los resultados de otros métodos.

```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice =
    discountedPrice(basePrice, seasonDiscount, fees);
```

Solución

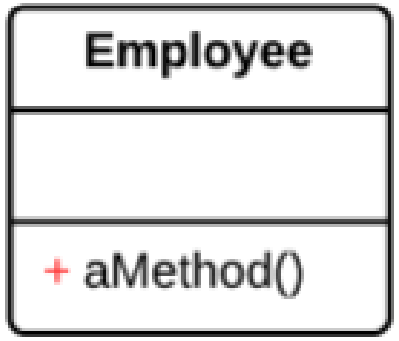
Que el método haga las llamadas a los getters directamente.

```
int basePrice = quantity * itemPrice;
double finalPrice =
        discountedPrice(basePrice);
```

Ocultar Método

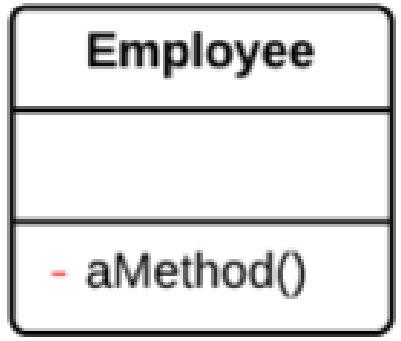
Problema

Un método no es usado por otras clases.



Solución

Cambiar la visibilidad del método a privado o protegido.



Reemplazar Código de error con Excepción

Problema

Un método retorna un valor especial que indica que sucedió un error.

```
int withdraw(int amount) {
  if (amount > _balance) {
    return -1;
  }
  else {
    balance -= amount;
    return 0;
  }
}
```

Solución

Arrojar una excepción.

```
void withdraw(int amount)
    throws BalanceException {
  if (amount > _balance) {
    throw new BalanceException();
  }
  balance -= amount;
}
```

Reemplazar Excepción con Test

Problema

Se lanza una excepción en un lugar donde ejecutar una evaluación es suficiente.

```
double getValueForPeriod
        (int periodNumber) {
  try {
    return values[periodNumber];
  } catch (ArrayIndexOutOfBoundsException e) {
    return 0;
  }
}
```

Solución

Reemplazar la excepción con la condición.

```
double getValueForPeriod
    (int periodNumber) {
  if (periodNumber >= values.length) {
    return 0;
  }
  return values[periodNumber];
}
```

Organizando Datos

Mover características entre Objetos

- Estas técnicas de refactorización buscan mejorar la estructura del código, reemplazando tipos primitivos con clases.
- Un resultado importante, es minimizar el “tangling” que pudiese resultar de asociaciones inadecuadas entre las clases.

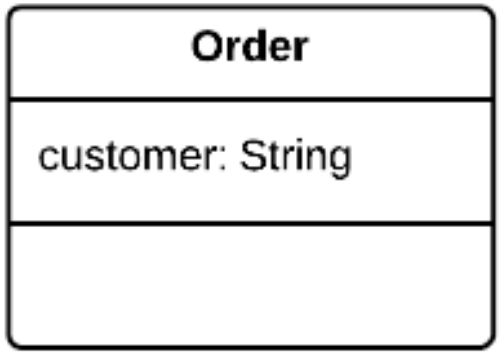
Técnicas

- Reemplazar Atributo Primitivo con Objeto
- Reemplazar un Array con un Objecto
- Duplicar Data Observada
- Cambiar Asociaciones Unidireccionales a Bidireccionales
- Cambiar Asociaciones Bidireccionales a Unidireccionales
- Reemplazar ‘Número Mágico’ con una constante
- Encapsular Colección
- Reemplazar ‘Type Code’ con Clases y Subclases
- Reemplazar Subclases con Campos

Reemplazar Atributo Primitivo con Objeto

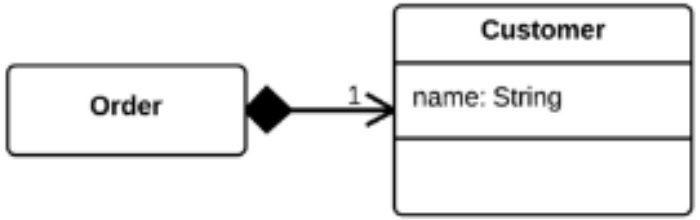
Problema

Una clase o grupo de clases contiene un campo de datos. El campo tiene su propio comportamiento y datos asociados.



Solución

Crear una nueva clase, colocar el capo y su comportamiento en en la clase. Asociar las clases.



Reemplazar un Arreglo con un Objeto

Problema

Existe un arreglo que contiene varios tipos de datos.

```
String[] row = new String[2];
row[0] = "Liverpool";
row[1] = "15";
```

Solución

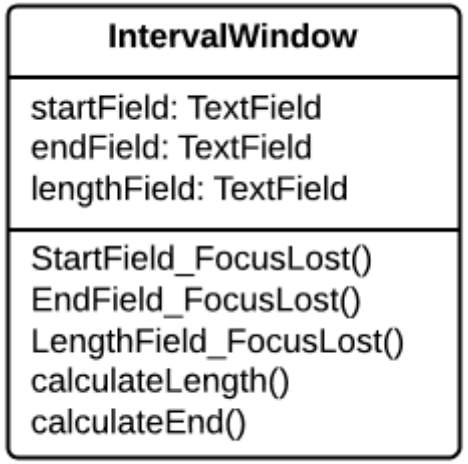
Reemplazar el arreglo con un objeto que permita manejar los datos.

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Duplicar Data Observada

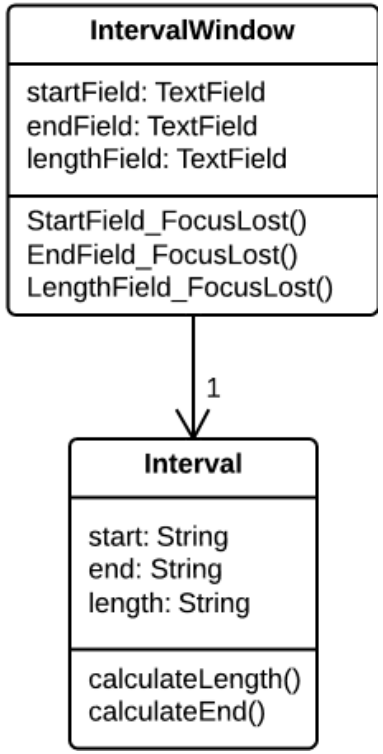
Problema

Se están manejando datos en clases responsables de la GUI.



Solución

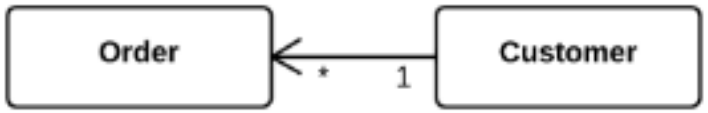
Separar el código que maneja los datos y ponerlos en una clase, asegurando conexión y sincronización.



Cambiar Asociaciones Unidireccionales a Bidireccionales

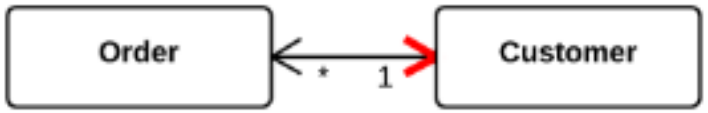
Problema

Se tienen dos clases donde ambas que usan las funcionalidades de la otra, pero la asociación es unidireccional.



Solución

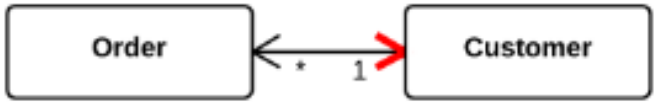
Agregar la asociación a la clase que lo necesita.



Cambiar Asociaciones Bidireccionales a Unidireccionales

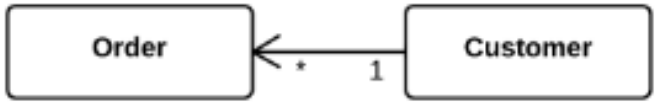
Problema

Existen relaciones bidireccionales entre dos clases, pero una no hace uso de las funciones de la otra.



Solución

Remover la asociación que no se requiere.



Reemplazar 'Número Mágico' con una constante

Problema

El programa hace uso de un número varias veces y que representa una regla de negocio o característica.

```
double potentialEnergy
        (double mass, double height) {
  return mass * height * 9.81;
}
```

Solución

Reemplazar ese número con una constante. Colocar un nombre representativo.

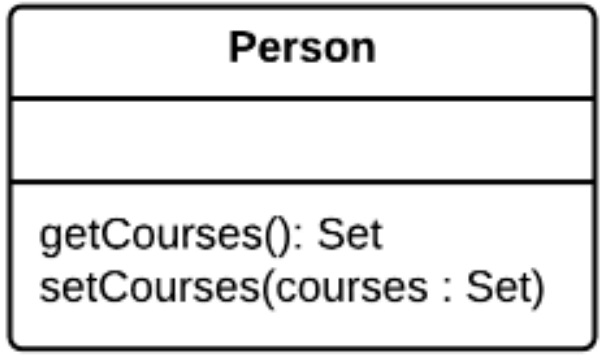
```
static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
  return mass * height * GRAVITATIONAL_CONSTANT;
}
```

Encapsular Colección

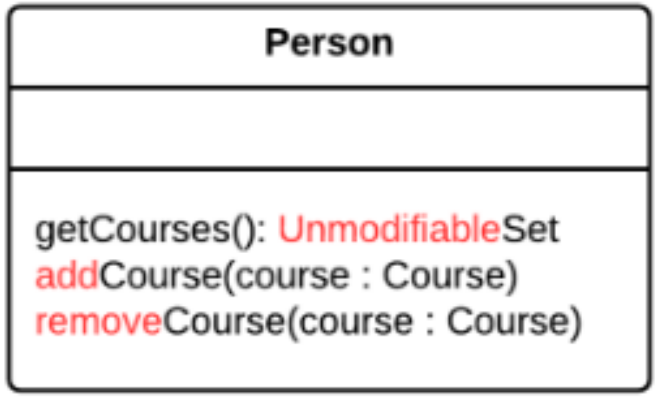
Problema

Una clase contiene una colección y un simple getter y setter para trabajar con ella.



Solución

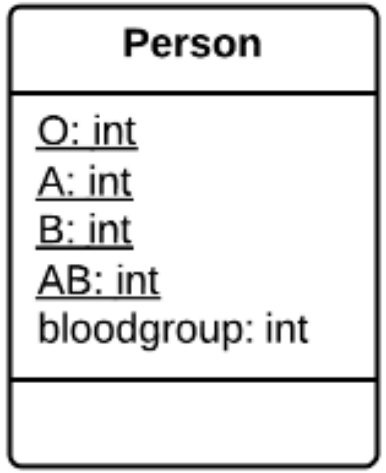
Hacer que el getter retorne valores de solo lectura. Crear métodos para agregar y remover elementos.



Reemplazar ‘Type Code’ con Clases

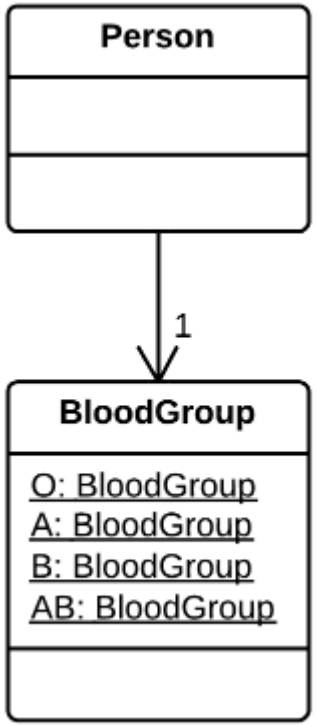
Problema

Una clase contiene ‘type code’.



Solución

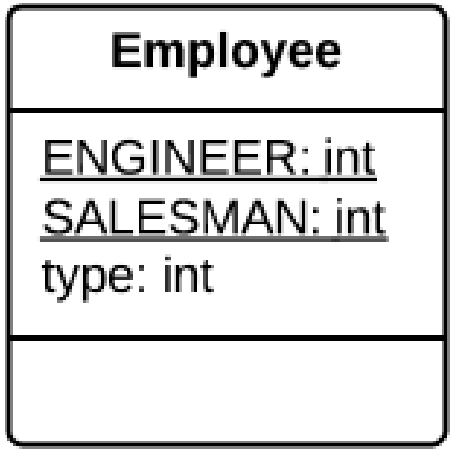
Crear una clase y usar sus objetos.



Reemplazar 'Type Code' con SubClases

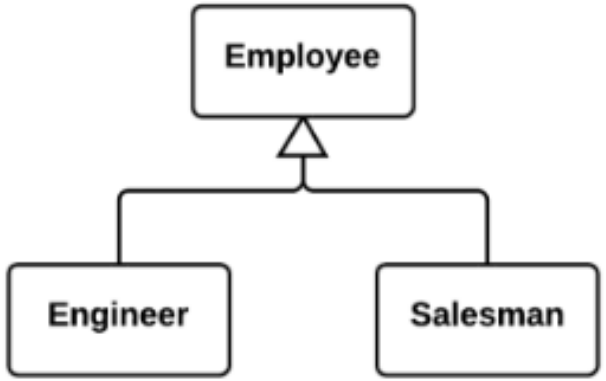
Problema

Una clase contiene 'type code' y su tipo afecta el comportamiento del programa.



Solución

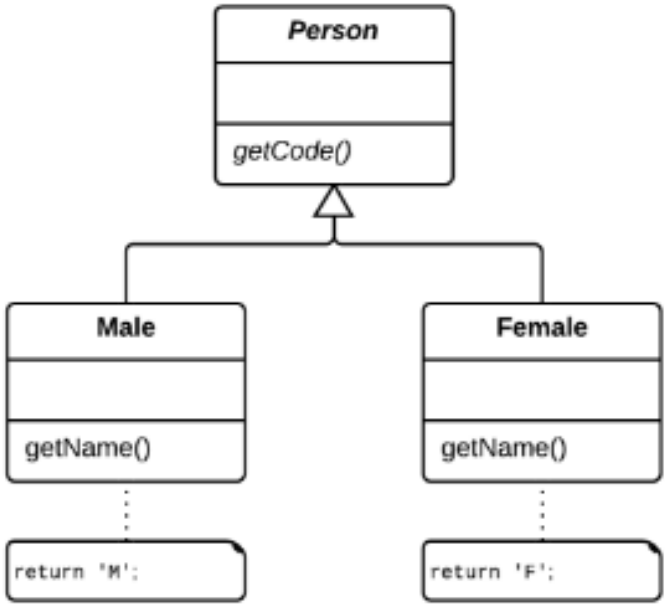
Crear una subclase por cada valor. Agregar el comportamiento.



Reemplazar Subclases con Campos

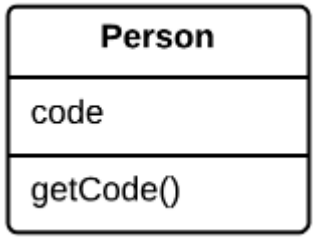
Problema

Existen clases que tienen un método que difieren solo en el valor de retorno.



Solución

Reemplazar los métodos con campos en la clase padre y eliminar las subclases.



Antes de finalizar

Puntos para recordar

- Recordar un ejemplo de cada técnica de refactorización correspondientes a cada categoría:
  - Simplificando llamadas a métodos
  - Organizando datos
  - Composición de Métodos
  - Simplificación de Sentencias Condicionales
  - Manejo de Generalización
  - Mover características entre Objetos
- Asociar malos olores con potenciales técnicas de refactorización.

Lectura adicional

- Martin Fowler, “Refactoring: Improving the Design of Existing Code”, first or second edition
- Source Making:
  - https://sourcemaking.com/refactoring
- Refactoring Guru:
  - https://refactoring.guru/

Próxima sesión

- Introducción a las pruebas de software y elaboración de casos de prueba