J820

Refactoring

Design no Código

Helder da Rocha (helder@acm.org)

argonavis.com.br

O que é

- Uma técnica de design baseada no código
- Refactoring (ou refatoramento) permite que se altere o design de uma aplicação, grande ou pequena, alterando o seu código diretamente
  - É uma prática altamente disciplinada e previsível. Não é hacking puro e simples.
  - Visa sempre o melhoramento do código
  - Refatoramento contínuo melhora o código e a aplicação inteira continuamente
  - Todo mundo faz (ou já fez), nem sempre de maneira disciplinada, porém.

Quando aplicar

- Sempre que possível
- Em TDD, refactorings são essenciais já que a primeira solução, a mais simples, geralmente não tem o melhor design
- Alterações como "tornar variável private" são um tipo de refatoramento
  - Afeta não apenas a variável em questão, mas todos que dependem dela
  - O refatoramento consiste em fazer todas as mudanças necessárias para que o código volte a funcionar após a alteração.

Testes são fundamentais

- Refatoramento é mexer no código que está funcionando
- É um grande risco se não houver testes para esse código
- Antes de refatorar, escreva testes de unidade para os elementos que serão alterados
  - Execute os testes várias vezes durante o processo
  - Modifique os testes para mantê-los em dia com o código que testam
  - Siga o passo-a-passo recomendado por cada refatoramento. Vários sugerem quando novos testes devem ser escritos e executados

Experiência

- Refatoramentos são simplesmente técnicas para mexer no design do código
  - Geralmente é para melhorar o código em relação às boas práticas OO
  - Em alguns casos pode piorar nesse aspecto quando há outros objetivos no processo (ex: performance)
- Refatoramentos refletem experiências testadas
  - Ganha-se tempo ao se utilizar um padrão de refatoramento em vez de descobrir tudo sozinho

Patterns

- O conhecimento de Padrões de projeto (Design Patterns) não é essencial para utilizar refatoramentos, mas ajuda bastante
  - Muitos refatoramentos visam transformar o código de forma a aplicar um padrão
  - Outros refatoramentos partem de um padrão de projeto ou produzem padrões no processo
- Um conhecimento geral dos padrões GoF traz grandes benefícios para qualquer desenvolvedor

Exemplo interativo de refactoring

- A melhor maneira de entender como funciona o refatoramento é através de um exemplo interativo
- Utilizaremos o exemplo do primeiro capítulo do livro "Refactoring", de Martin Fowler
  - O exemplo mostra um código que "cheira mal" devido a vários problemas que dificultam o seu reuso, manutenção, compreensão e até eficiência
  - Demonstraremos alguns passos que tornarão o programa bem melhor

Exemplo do livro: Locadora de Videos

Movie.java

```
public class Movie {
  public static final int  CHILRENS = 2;
  public static final int  REGULAR = 0;
  public static final int  NEW_RELEASE = 1;

  public String _title;
  private int _priceCode;

  public Movie(String title, int priceCode) {
    _title = title;
    _priceCode = priceCode;
  }

  public int getPriceCode() {
    return _priceCode;
  }

  public void setPriceCode(int arg) {
    _priceCode = arg;
  }

  public String getTitle() {
    return _title;
  }
}
```

Rental.java

```
class Rental {
  private Movie _movie;
  private int _daysRented;

  public Rental(Movie movie, int daysRented) {
    _movie = movie;
    _daysRented = daysRented;
  }

  public int getDaysRented() {
    return _daysRented;
  }

  public Movie getMovie() {
    return _movie;
  }
}
```

Customer.java 1/3

```
class Customer {
  private String _name;
  private Vector _rentals = new Vector;

  public Customer(String name) {
    _name = name;
  }

  public void addRental(Rental arg) {
    _rentals.addElement(arg);
  }

  public String getName() {
    return _name;
  }

....
```

Customer.java 2/3

```
...

public String statement() {

    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      // Determine amounts for each line
      switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
          thisAmount += 2;
          if (each.getDaysRented() > 2)
            thisAmount += (each.getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          thisAmount += each.getDaysRented() * 3;
          break;
        case Movie.CHILDRENS:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented() - 3) * 1.5;
          break;
      }
 ...
```

Customer.java 3/3

```
...

// Add frequent renter points
        frequentRenterPoints++;

      // Add bonus for a two-day, new-release rental
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1)
        frequentRenterPoints++;

// Show figures for this rental
      result += "\t" + each.getMovie().getTitle() + "\t" +
        Sting.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
    }

    // Add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
      " frequent renter points";
    return result;
  }
}
```

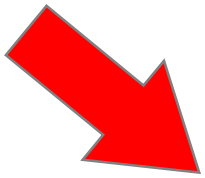
Catálogo de refactorings

- Assim como padrões de projeto, técnicas de refatoramento refletem a experiência adquirida por programadores experientes que pode ser reutilizada por todos para alcançar resultados desejados mais rapidamente e de maneira testada
- Assim como padrões de projeto, refatoramentos têm nome. Além disso, têm uma forma reversível de aplicação, geralmente em passos pequenos
  - Deve-se sempre executar testes de unidade em cada passo, para garantir a reversibilidade
  - Catálogos de refatoramento geralmente descrevem como implementar cada alteração detalhadamente

Extract Method

- ***Problema***: *Você tem um fragmento de método que pode ser agrupado em método independente*
- ***Solução***: *transforme o fragmento em um método cujo nome explique a sua finalidade.*

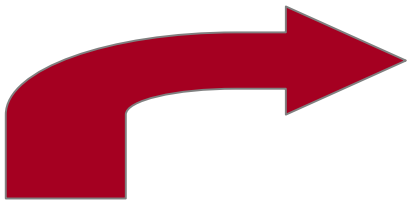
```
void printOwing(int amount) {
   printBanner();
   System.out.println ("name: " + _name);
   System.out.println ("amount " + amount);
}
```



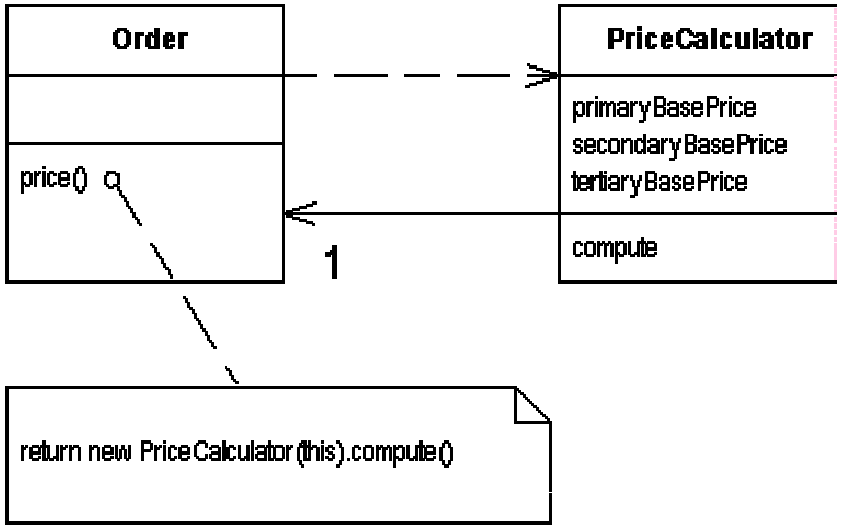
```
void printOwing() {
   printBanner();
   printDetails(amount);
}
void printDetails(double amount) {
   System.out.println ("name: " + _name);
   System.out.println ("amount " + amount);
}
```

Replace Method With Method Object

- ***Problema***: *Você tem um método longo que usa variáveis locais de tal maneira que não é possível aplicar Extract Method*
- ***Solução***: *Transforme o método em um objeto para que todas as variáveis locais virem atributos*



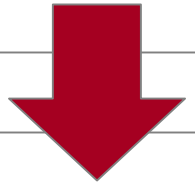
```
class Order... double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // long computation; ...
}
```



Substitute Algorithm

- ***Problema***: *Algoritmo de método pode ser melhorado.*
- ***Solução***: *substitua o conteúdo do método com um algoritmo que faça a mesma coisa mas seja melhor.*

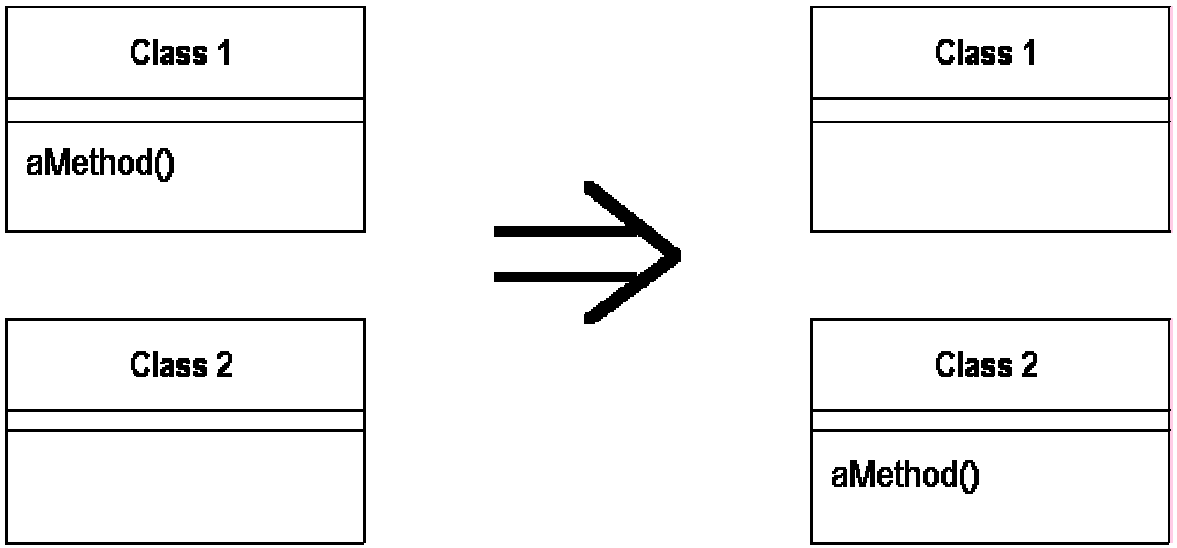
```
String foundPerson(String[] people){
  for (int i = 0; i < people.length; i++) {
    if (people[i].equals ("Don")){ return "Don"; }
    if (people[i].equals ("John")){ return "John"; }
    if (people[i].equals ("Kent")){ return "Kent"; }
  } return "";
}
```



```
String foundPerson(String[] people){
  List candidates =
    Arrays.asList(new String[] {"Don", "John", "Kent"});
    for (int i=0; i<people.length; i++)
      if (candidates.contains(people[i]))
        return people[i]; return "";
}
```

Move Method

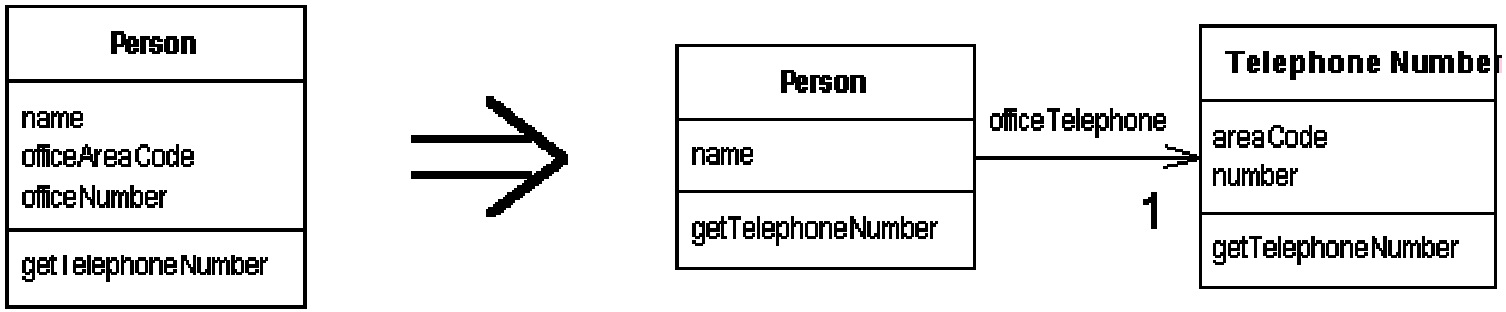
- ***Problema***: *Um método é usado por mais recursos de outra classe que na classe em que é definido.*
- ***Solução***: *crie novo método com corpo similar na classe em que é mais usado. Transforme o antigo em simples delegação ou elimine-o.*



Veja também:
Move Field

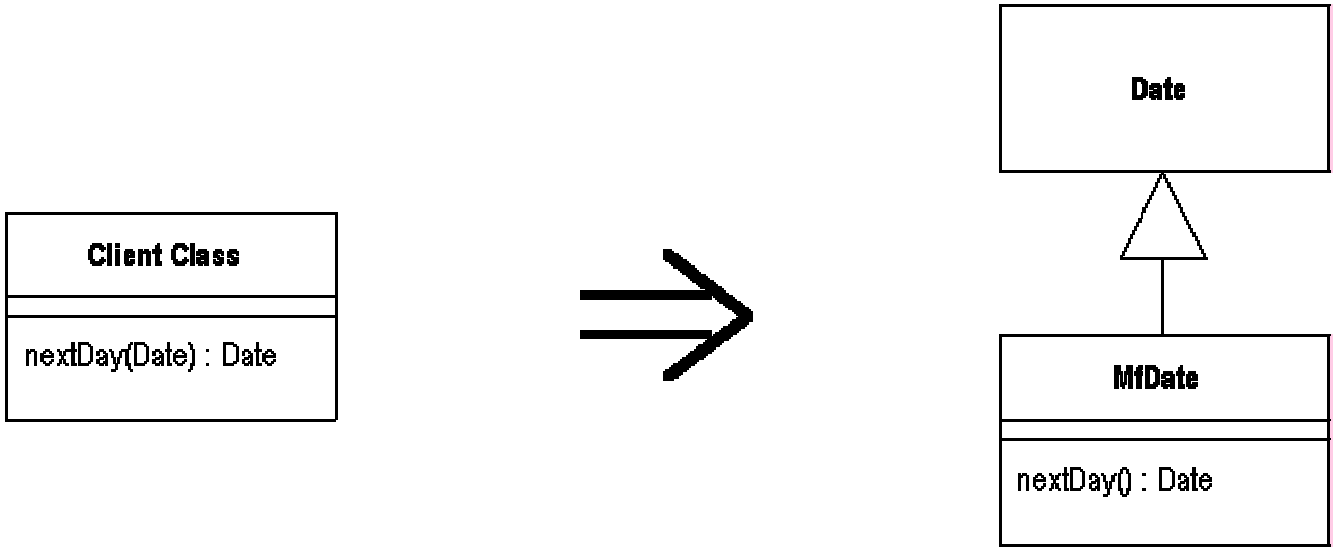
Extract Class

- ***Problema***: *Você tem uma classe fazendo trabalho que deveria ser feito por duas classes.*
- ***Solução***: *Crie uma nova classe e transfira métodos e atributos relevantes para ela.*



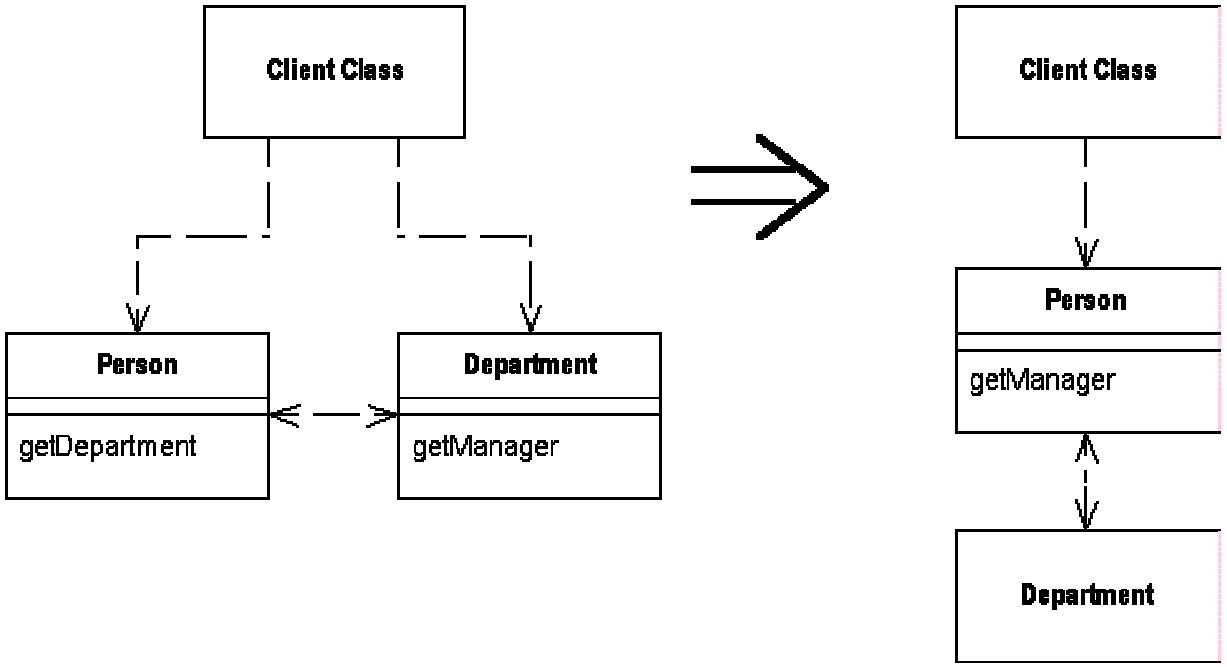
Introduce Local Extension

- ***Problema***: *Uma classe servidora que você está usando requer vários métodos adicionais, mas você não pode modificar a classe.*
- ***Solução***: *crie uma nova classe que contém esses métodos extras. Faça essa extensão uma subclasse ou um wrapper da original.*



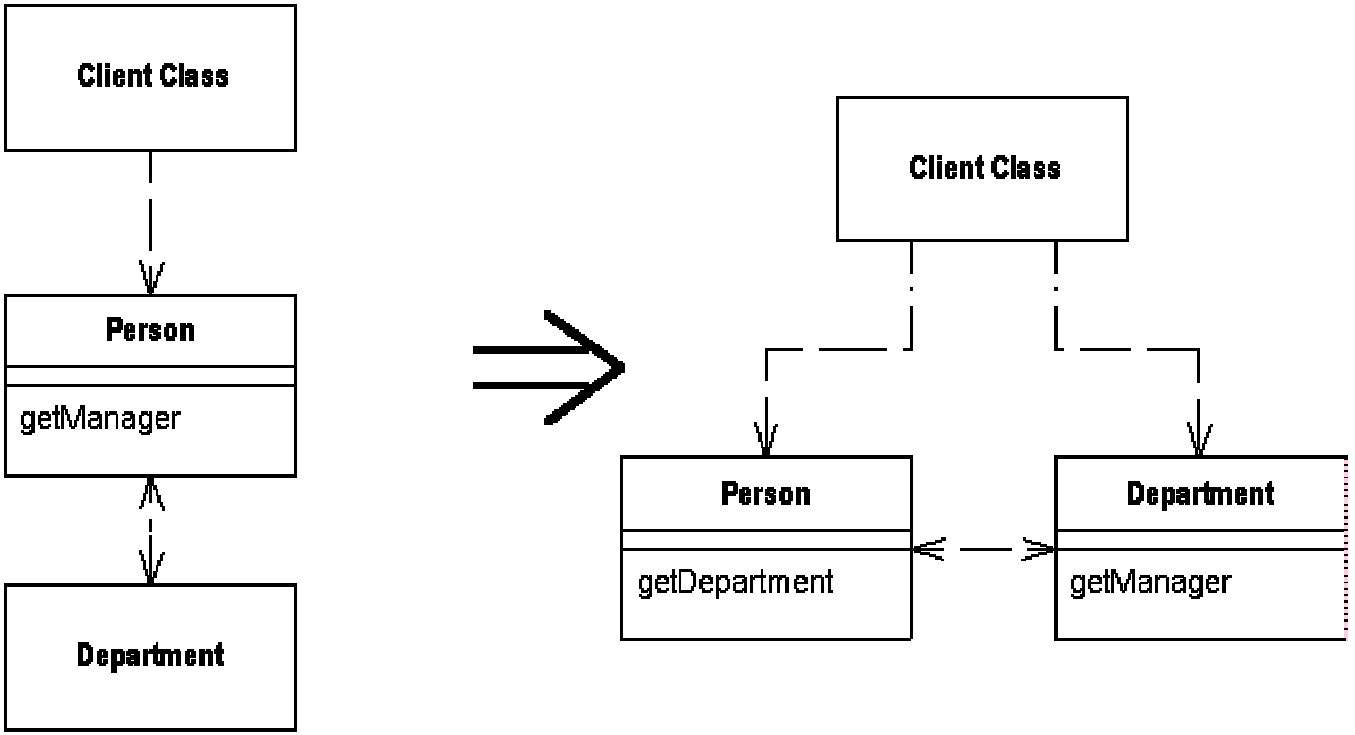
Hide Delegate

- ***Problema***: *Um cliente tem acesso e está chamando uma classe que é delegada pelo objeto que utiliza.*
- ***Solução***: *Crie métodos no servidor para ocultar o objeto do acesso pelo cliente.*



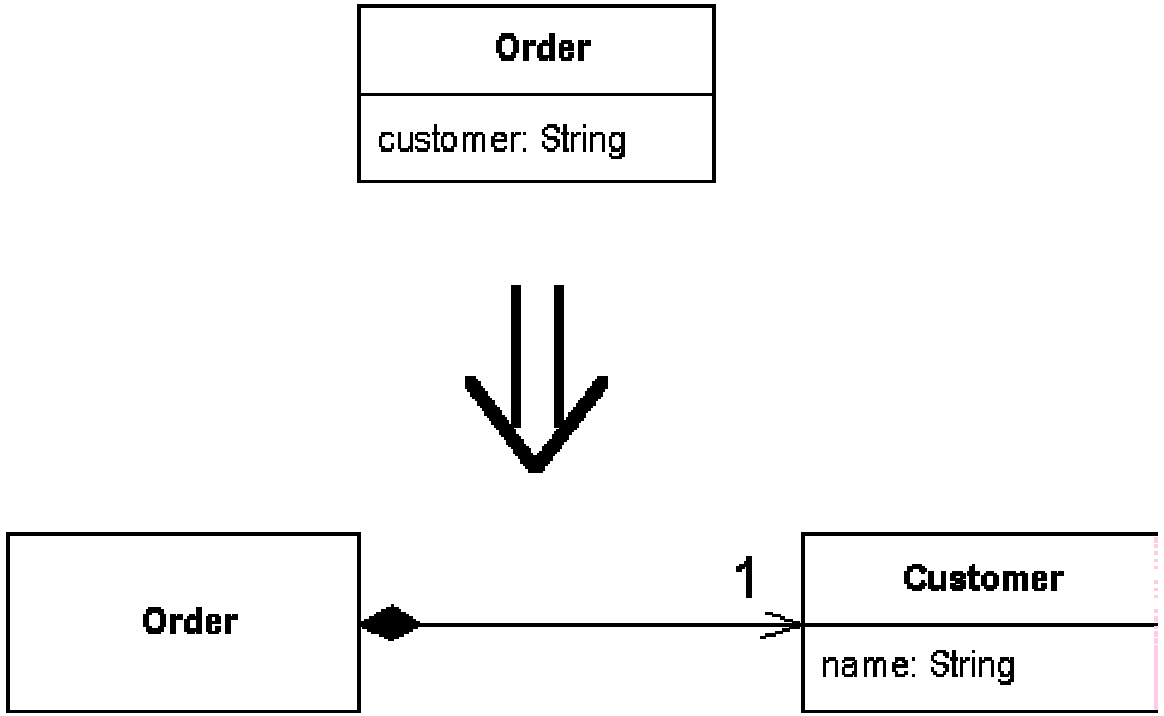
Remove Middle-Man

- ***Problema***: *Uma classe está fazendo delegações simples em excesso.*
- ***Solução***: *Faça o cliente delegar diretamente*.



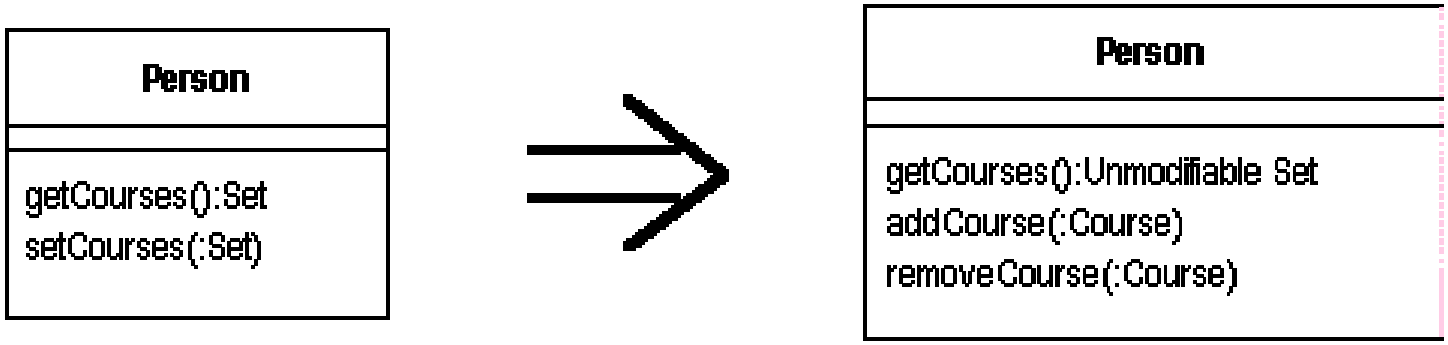
Replace Data Value With Object

- ***Problema***: *Você tem um item de dados que requer dados adicionais ou comportamento.*
- ***Solução***: *transforme o item de dados em um objeto.*



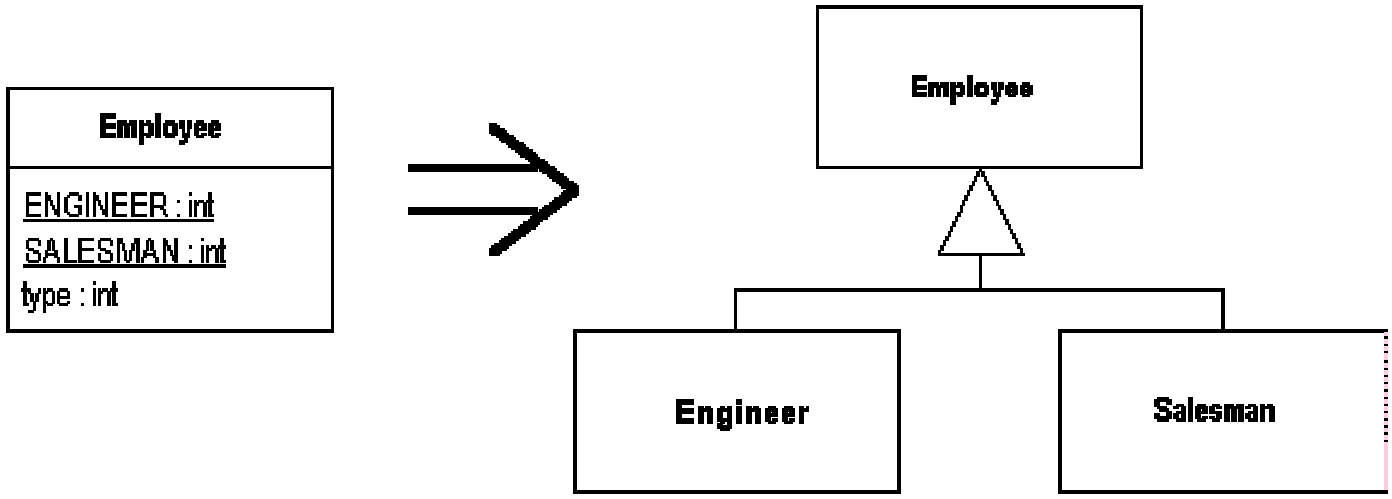
Encapsulate Collection

- ***Problema***: *Um método retorna uma Coleção (List, Set, HashMap) - acesso total e ausência de controle sobre tipos de dados (tudo é Object)*
- ***Solução***: *Faça-o retornar uma visão read-only dos dados e forneça métodos de adição e remoção.*



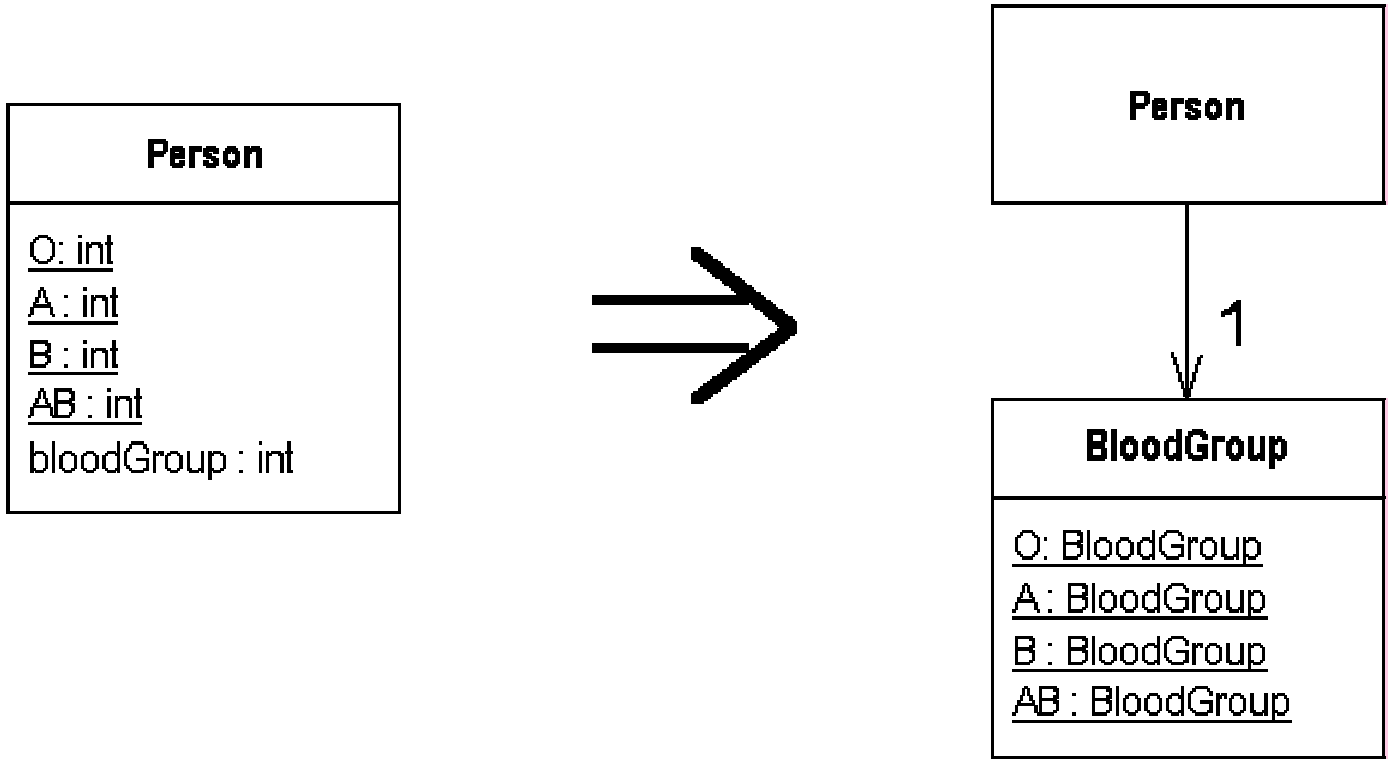
Replace Type Code With Subclass

- ***Problema***: *Você tem um código de tipo imutável que afeta o comportamento de uma classe.*
- ***Solução***: *Substitua o código de tipo com uma subclasse.*



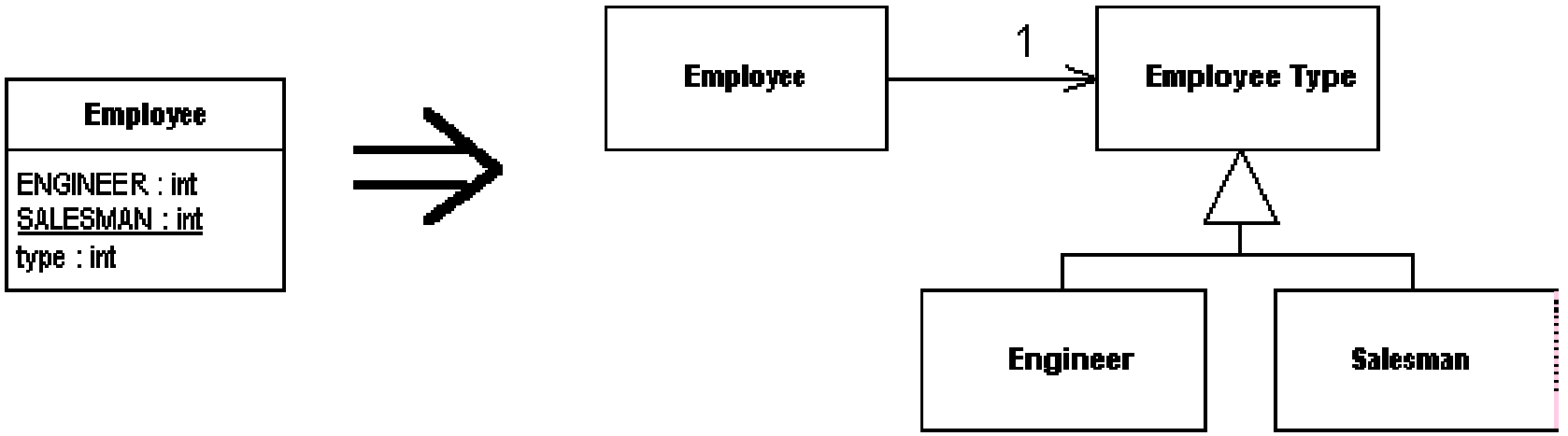
Replace Type Code With Class

- ***Problema***: *Uma classe tem um código numérico de tipo que não afeta seu comportamento*
- ***Solução***: *Substitua o número com uma nova classe.*



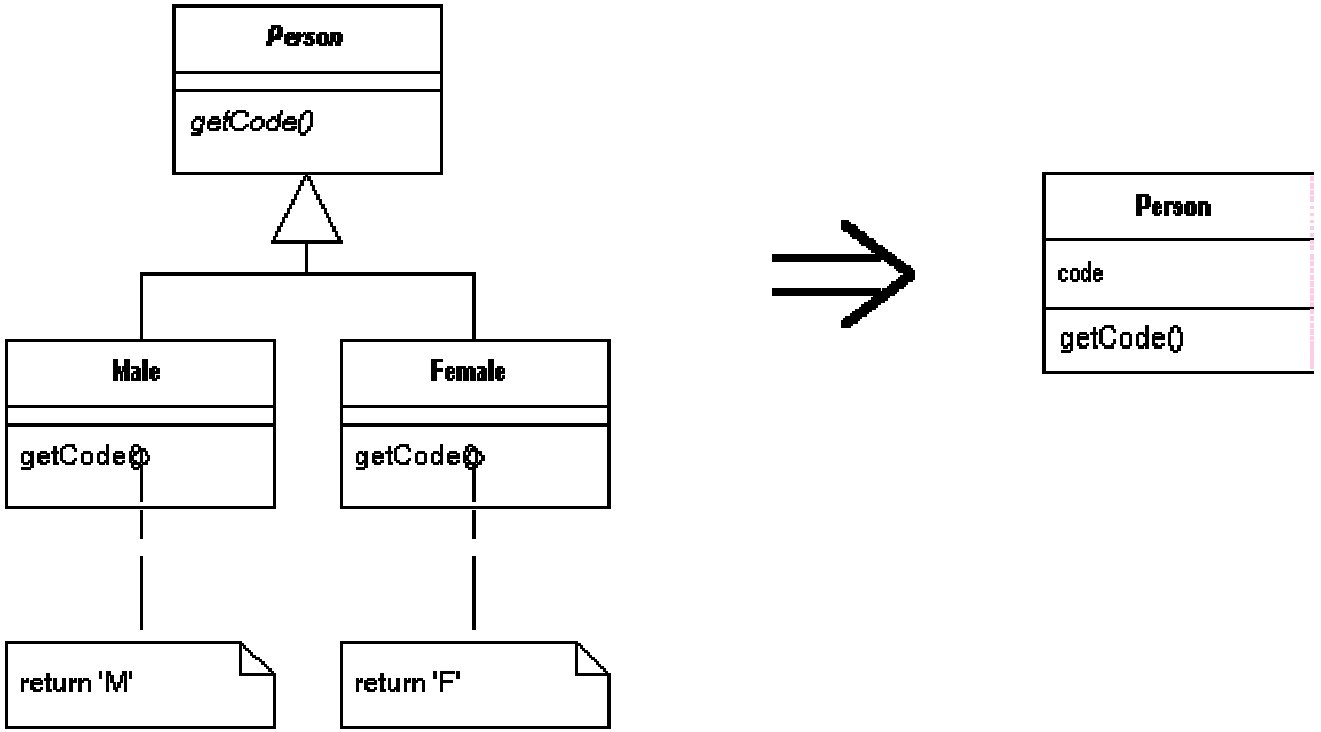
Replace Type Code With State/Strategy

- ***Problema***: *Você tem código que representa um tipo e que afeta o comportamento de uma classe mas não pode usar subclasses*
- ***Solução***: *Represente o tipo com um objeto de estado.*



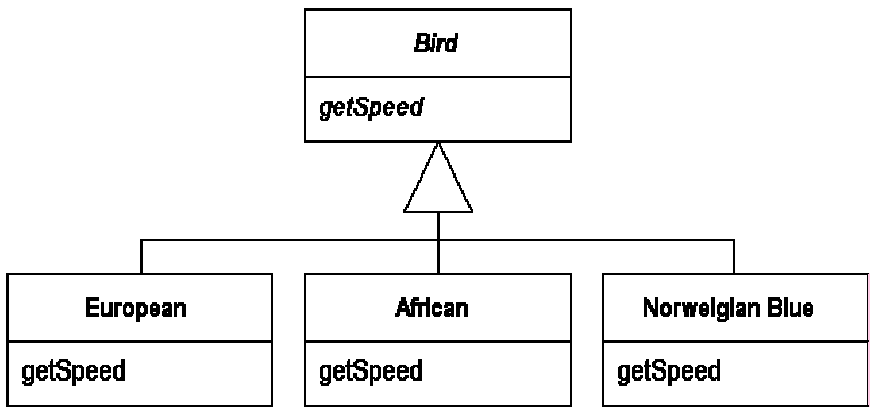
Replace Subclass with Fields

- ***Problema***: *Você tem subclasses que só variam em métodos que retornam dados constantes*
- ***Solução***: *mova os métodos para atributos de superclasses e elimine as subclasses*



Replace Conditional With Polymorphism

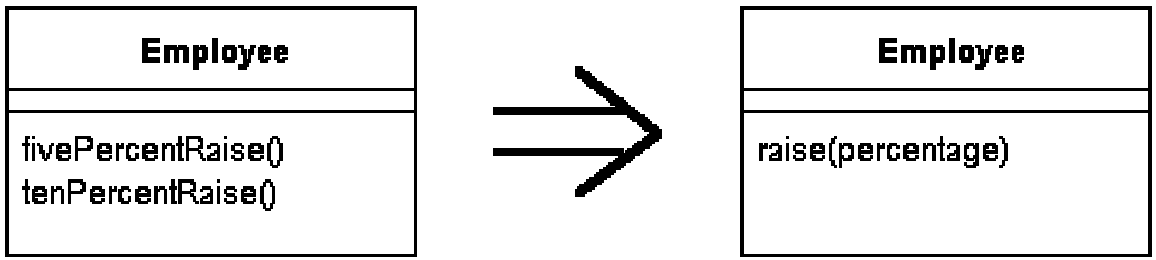
```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
         return getBaseSpeed();
    case AFRICAN:
         return getBaseSpeed() -
              getLoadFactor() * _coconuts;
    case NORWEIGIAN_BLUE:
         return (_isNailed)
         ? 0
         : getBaseSpeed(_voltage);
  } throw new RuntimeException
         ("Should be unreachable");
}
```



- ***Problema***: *Você tem um tipo condicional que age diferentemente dependendo do tipo de um objeto.*
- ***Solução***: *Mova cada bloco para método em uma subclasse e faça o método original abstrato.*

Parameterize Method

- ***Problema***: *Vários métodos fazem coisas similares mas com diferentes valores contidos no corpo do objeto*
- ***Solução***: *crie um método que usa um parâmetro para os valores diferentes.*



Preserve Whole Object

- ***Problema***: *Você está recebendo diversos valores de um objeto e passando-os como parâmetros em uma chamada*
- ***Solução***: *Mande o objeto inteiro*

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

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

Replace Constructor With Factory Method

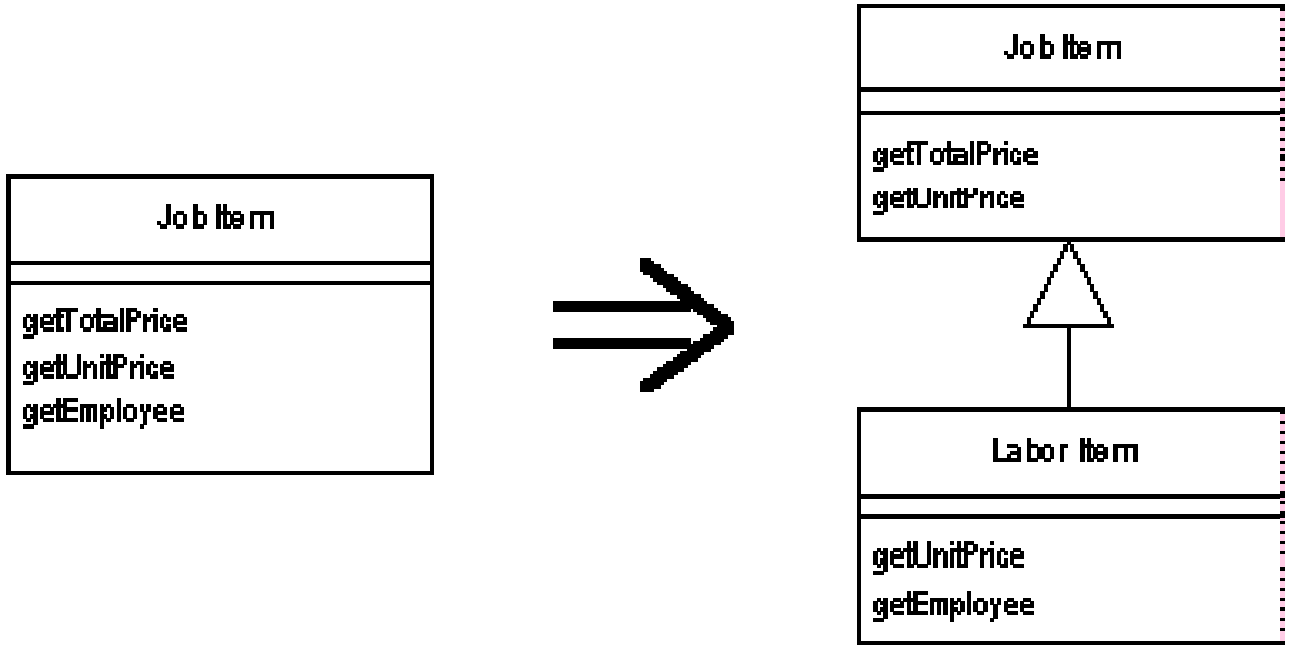
- ***Problema***: *Você quer poder realizar mais que simples construção quando cria um objeto (ou ter mais controle e flexibilidade ao criar objetos)*
- ***Solução***: *Substitua construtor com um factory method*

```
Employee (int type) {
      _type = type;
}
```

```
static Employee create(int type) {
      return new Employee(type);
}
```

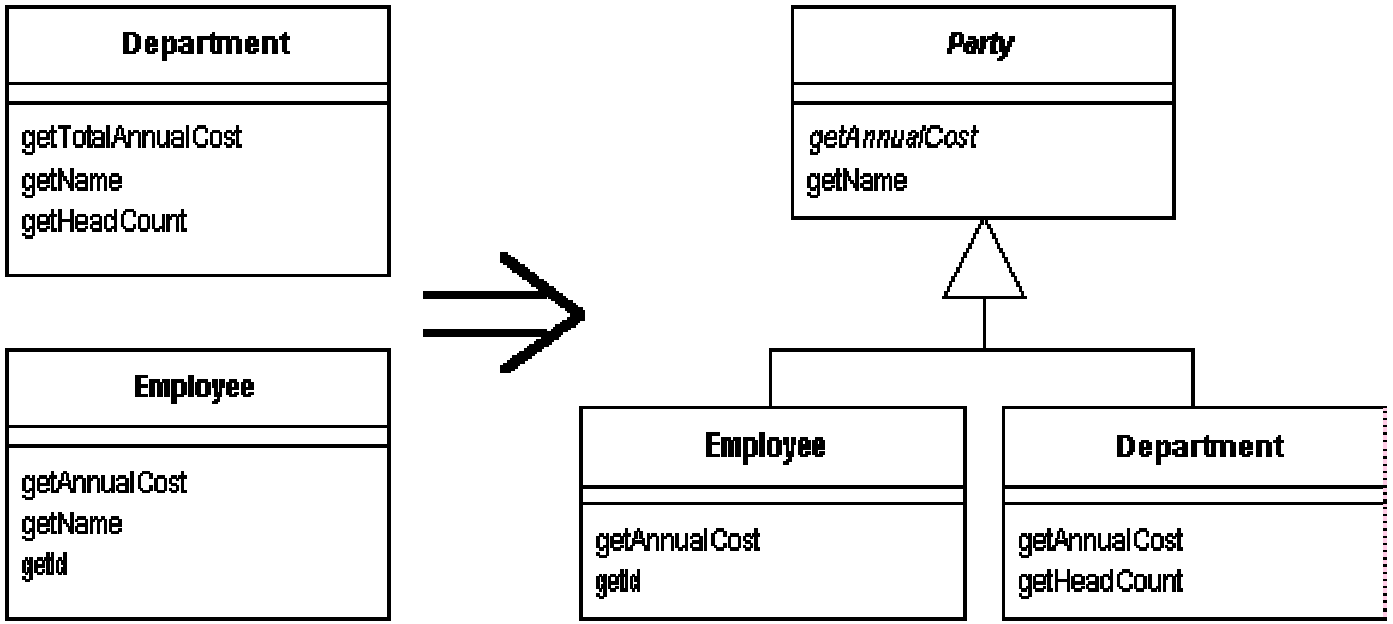
Extract Subclass

- ***Problema***: *Uma classe tem recursos que só são utilizados em algumas instâncias*
- ***Solução***: *crie uma subclasse para esse conjunto de recursos.*



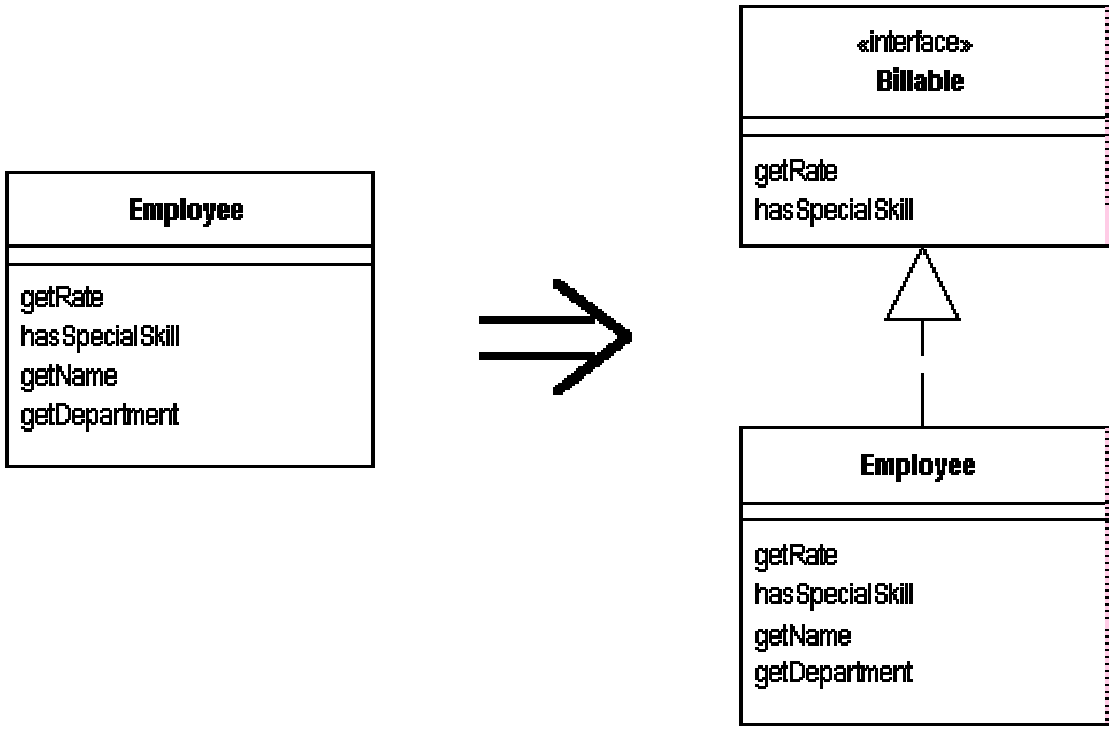
Extract Superclass

- ***Problema***: *Você tem duas classes com recursos similares.*
- ***Solução***: *crie uma subclasse e mova os recursos comuns à superclasse.*



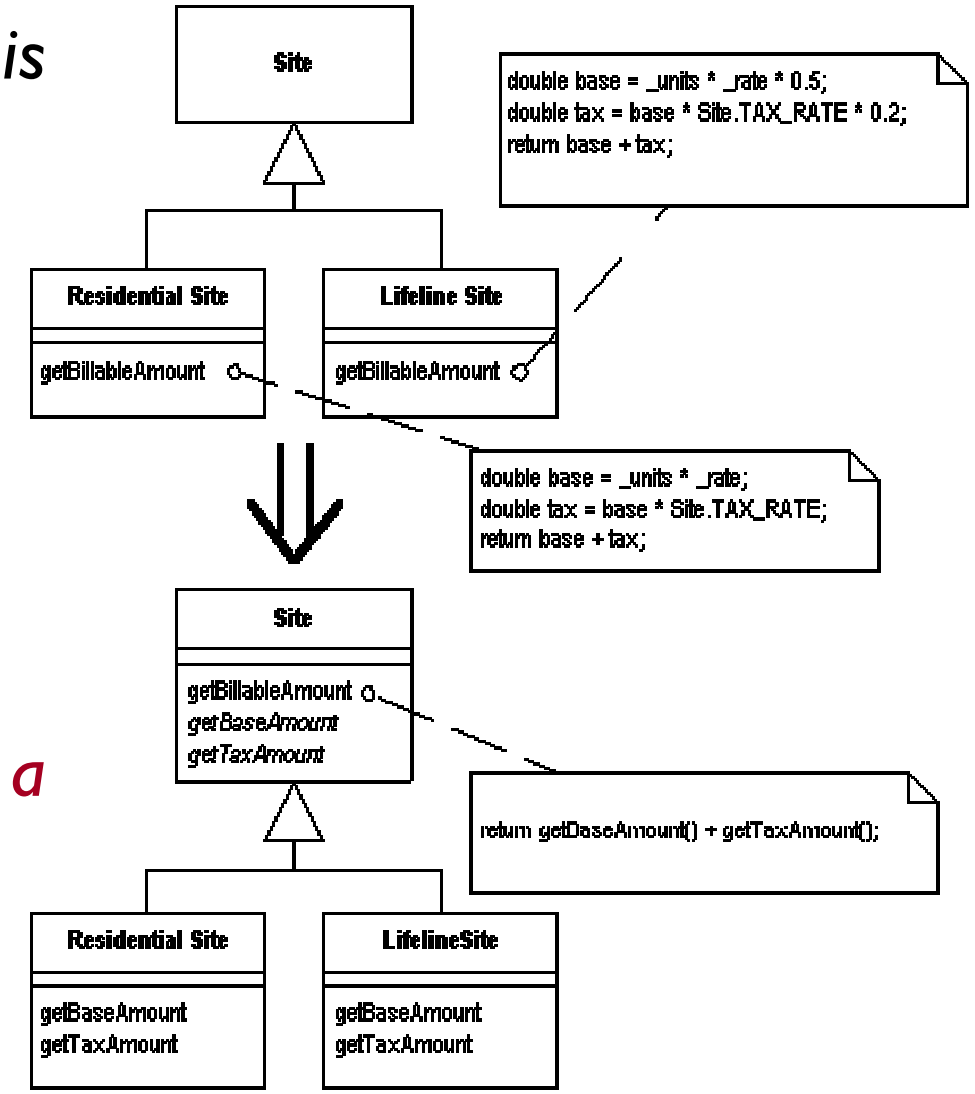
Extract Interface

- ***Problema***: *Vários clientes usam o mesmo subconjunto da interface de uma classe, ou duas classes têm parte de sua interface em comum*
- ***Solução***: *extraia o subconjunto em uma interface*



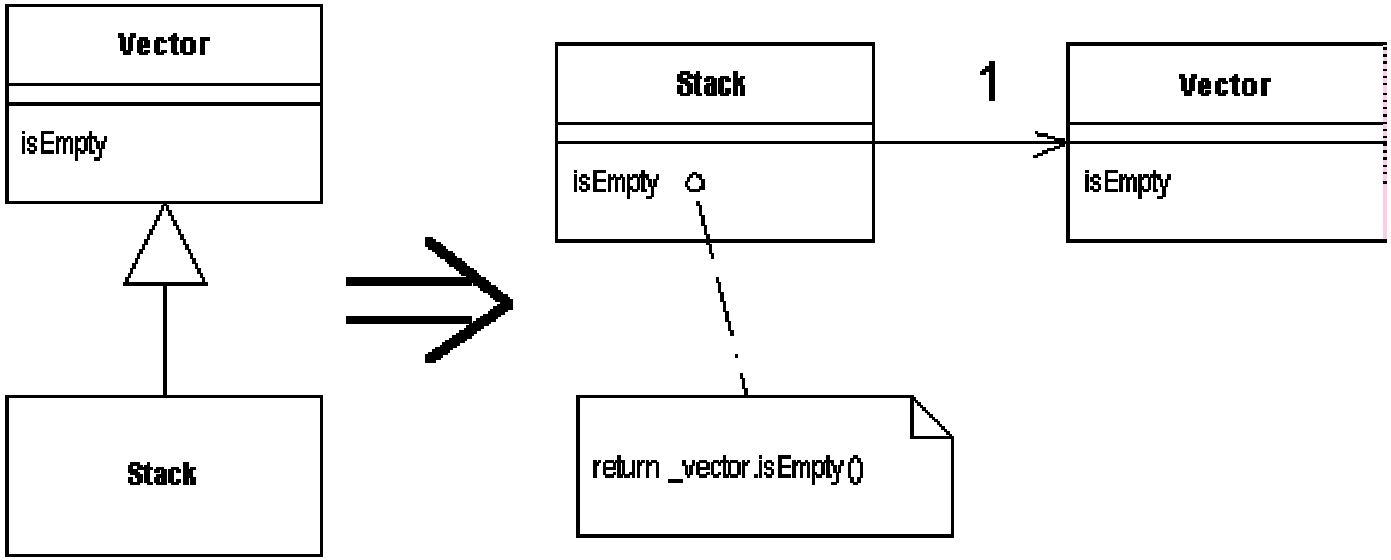
Form Template Method

- ***Problema***: *Você tem dois métodos em subclasses que realizam passos similares na mesma ordem, porém os passos são implementados de forma diferente*
- ***Solução***: *Coloque os passos em métodos com a mesma assinatura e depois puxe-os acima na hierarquia.*



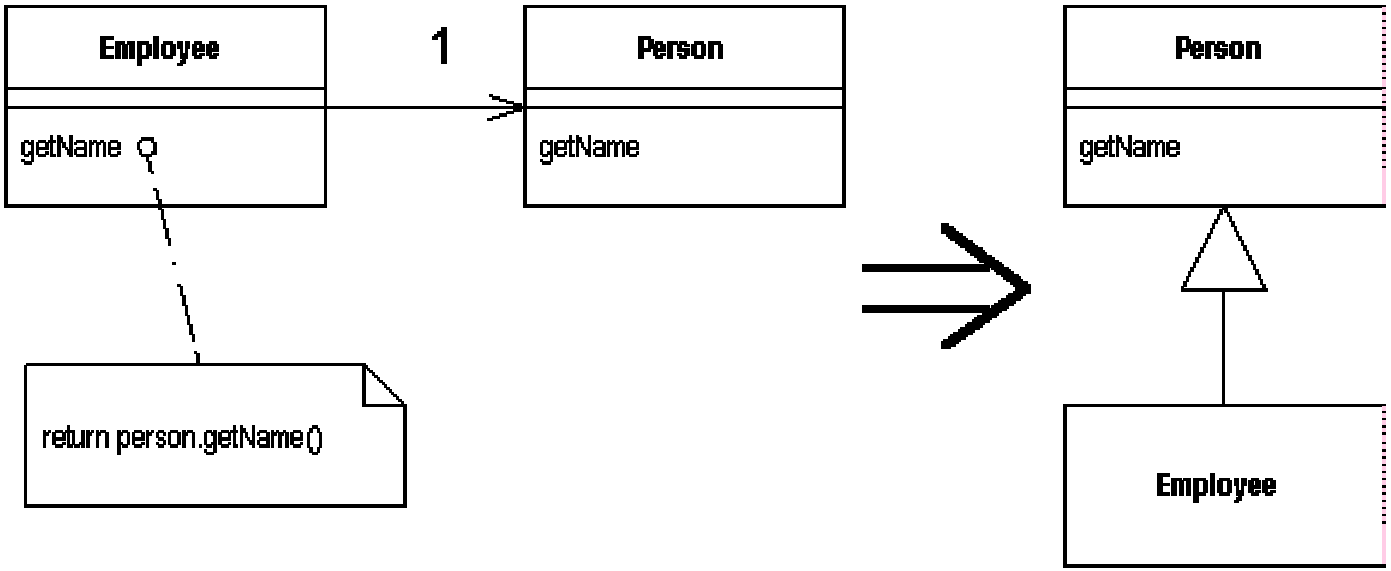
Replace Inheritance With Delegation

- ***Problema***: *Uma subclasse usa apenas uma parte da interface da superclasse ou não deseja herdar dados*
- ***Solução***: *Crie um atributo de dados para a superclasse, ajuste os métodos para delegar à superclasse e remova a estrutura de subclasse*



Replace Delegation With Inheritance

- ***Problema***: *Você usa delegação e está frequentemente escrevendo pequenas delegações para toda a interface*
- ***Solução***: *Faça a classe que delega uma subclasse da delegada*



Alguns "cheiros ruins" e suas soluções

- Código duplicado
  - Extract Method, Extract Class, Pull Up Method, Form Template Method
- Intimidade inapropriada
  - Mome Method, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation
- Classe gigante
  - Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
- Método gigante
  - Extract Method, Replace Method with Method Object, Decompose Conditional

Mais Refatoramentos

- Veja o catálogo do Martin Fowler (do livro Refactoring) em www.refactoring.com/catalog
- Procure descobrir refatoramentos específicos para a área no quel você está pesquisando
  - J2EE
  - Aplicações gráficas
  - Concorrência e programação paralela
- Refatoramento, Padrões de Projeto, estratégias (aplicações de patterns / idioms) e Melhores Práticas estão sempre relacionados
  - Todos fornecem experiências valiosas que melhoram a produtividade e utilização dos recursos de uma linguagem na solução de problemas de design.

Fontes

[1] Martin Fowler, "Refactoring: improving the design of existing code". Addison-Wesley Object Technology Series, 2000

[2] Martin Fowler. "On-line Refactoring Catalog". www.refactoring.com/catalog/. Fonte dos diagramas e imagens.

Curso J820
Produtividade e Qualidade em Java: Ferramentas e Metodologias

Revisão 1.1

(helder@acm.org)

argonavis.com.br