CS 683 Emerging Technologies: Embracing Change
Spring Semester, 2001
Doc 23 Some Refactorings with Patterns

References

Refactoring to Patterns, Joshua Kerievsky, 2001, http://industriallogic.com/xp/refactoring/

Refactoring: Improving the Design of Existing Code, Fowler, 1999

Some Refactorings with Patterns

But first some common refactorings

Replace Nested Conditional with Guard Clauses (250)[1]

You have a method with nested conditionals

The method is hard to read

The method does not make clear the normal path of execution

So

Use guard clauses for all the special cases

[1] Fowler 99, pp. 250-254

Example

```
payAmount() {
   double result;
   if (_isDead )
      result = deadAmount();
   else {
      if (_isSeparated )
         result = separatedAmount();
      else {
         if (_isRetired )
            result = retiredAmount();
         else
            result = self normalPayAmount();
      }
   }
   return result;
}
```

becomes

```
payAmount() {
   if (_isDead )
      return deadAmount();
   if (_isSeparated )
      return separatedAmount();
   if (_isRetired )
      return retiredAmount();
   return normalPayAmount();
}
```

Replace Conditional with Polymorphism (255)[2]

You have case statements or ifs

The cases depend on types

Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract

[2] Fowler 99 pp. 255-259

Example

```
class Employee {
   public double payAmount()
      if (_type = Employee.ENGINEER)
         return _monthlySalary;
      if (_type = Employee.MANAGER)
         return _monthlySalary * 2;
      if (_type = Employee.ENGINEER)
         return _monthlySalary/2;
}
```

So

- Create an EmployeeType class
- Create Engineer, Manager & Instructor subclasses of EmployeeType

We get the code on the next page

```
class Employee {
   public double payAmount() {
      return _type.payAmount(this);
   }
}

abstract class EmployeeType {
   pubic abstract double payAmount(Employee x);
}

class Engineer extends EmployeeType {
   pubic double payAmount(Employee anEmployee) {
      return anEmployee.monthlySalary();
   }
}

class Manager extends EmployeeType {
   pubic double payAmount(Employee anEmployee) {
      return anEmployee.monthlySalary() * 2;
   }
}

class Instructor extends EmployeeType {
   pubic double payAmount(Employee anEmployee) {
      return anEmployee.monthlySalary() / 2;
   }
}
```

Replace Type Code with State/Strategy (227)[3]

You have a type code that affects the behavior of a class, but you can not use subclassing

So

Replace the type code with a state object

Example

See example for Replace Conditional with Polymorphism

[3] Fowler 99, pp. 227-231

Form Template Method (345)[4]

We have two methods in subclasss that perform similar steps in the same order

So:

- Extract the steps into methods with the same signatures.
- The original methods in each class become the same.
- Pull the original method up to the parent class

[4] Fowler 99

Example

We start with:

```
class ResidentialSite extends Site {
   public double getBillableAmount() {
      double base = _units * _rate;
      double tax = base * Site.TAX_RATE;
      return base * tax;
   }
}

class LifelineSite extends Site {
   public double getBillableAmount() {
      double base = _units * _rate * 0.5;
      double tax = base * Site.TAX_RATE * 0.3;
      return base * tax;
   }
}
```

We refactor to:

```
abstract class Site {
   public double getBillableAmount() {
      return getBaseAmount() + getTaxAmount();
   }
   public abstract double getBaseAmount();
   public abstract double getTaxAmount();
}

class ResidentialSite extends Site {
   public double getBaseAmount() {
      return  _units * _rate;
   }
   public double getTaxAmount() {
      return  getBaseAmount() * Site.TAX_RATE;
   }
}

class LifelineSite extends Site {
   public double getBaseAmount() {
      return  _units * _rate * 0.5;
   }
   public double getTaxAmount() {
      return  getBaseAmount() * Site.TAX_RATE * 0.3;
   }
```

Replace Constructor with Factory Method (304)[5]

You want to do more than simple construction when you create an object

So

Replace the constructor with a factory method

One Motivation

You want to create an object based on a type code

Different type codes require different types of objects from subclasses

A constructor can only return one type

So use a factory method

The factory method

- Returns the correct type of object for the type code
- Return type is declared to be the base type

```
static Money currency(String currencyType&Amount) {
    blah
}
```

[5] Fowler 99

Encapsulate Constructors with Factory Methods[6]

You have a class with lots of constructors

Constructors do not communicate intent well

Many constructors on a class make it hard to decide which constructor to use

So

Encapsulate the constructors with intention-revealing factory methods

[6] Kerievsky 2000, pp. 4-8

Mechanics

Identify the class that has many constructors

Identify the catchall constructor

Make this constructor protected

For every type of object that can be created, write an intention-revealing factory method.

Replace all calls to constructors with calls to the proper factory method

Remove all unneeded constructors

Often these constructors/Factory methods will have many arguments

Consider using Introduce Parameter Object (295) of Fowler 99

Example

Loan Constructors

public Loan(notional, outstanding, customerRating, expire)
public Loan(notional, outstanding, customerRating, expire, maturity)
public Loan(capitalStategy, outstanding, customerRating, expire, maturity)
public Loan(type, capitalStategy, outstanding, customerRating, expire)
public Loan(type, capitalStategy, outstanding, customerRating, expire, maturity)

After refactoring:

protected Loan(type, capitalStategy, outstanding, customerRating, expire, maturity)

static Loan newTermLoan(notional, outstanding, customerRating, expire)
static Loan newTermLoanWithStategy(capitalStategy, notional, outstanding, customerRating, expire)

static Loan newRevolver(notional, outstanding, customerRating, expire)
static Loan newRevolverWithStategy(capitalStategy, notional, outstanding, customerRating, expire)

static Loan newRCTL(notional, outstanding, customerRating, expire, maturity)
static Loan newRCTLWithStategy(capitalStategy, notional, outstanding, customerRating, expire, maturity)

Encapsulate Subclass Constructors with Superclass Factory Methods[7]

You have a number of subclasses that implement a common interface

You want client code to interact with subclass instances through the common interface

Subclass instances are created from subclass constructors

This allows clients to know the subclass type

This leads programmers to add new features to subclasses, not the common interface

So

Encapsulate subclass constructors with intention-revealing factory methods in the parent class

[7] Kerievsky 2000, pp. 9-12

Mechanics

Make the subclass constructor protected

Create an intention-revealing Factory Method in the superclass

- The declared return type is the superclass

Replace all calls to the subclass constructor with calls to the superclass Factory Method

Compile and test after each replacement

Extract Special-Case Logic into Decorators[8]

Your classes/methods have lots of special case logic

Most of the time the normal case is used

There are lots of special cases

The special cases make the code hard to read & maintain

So

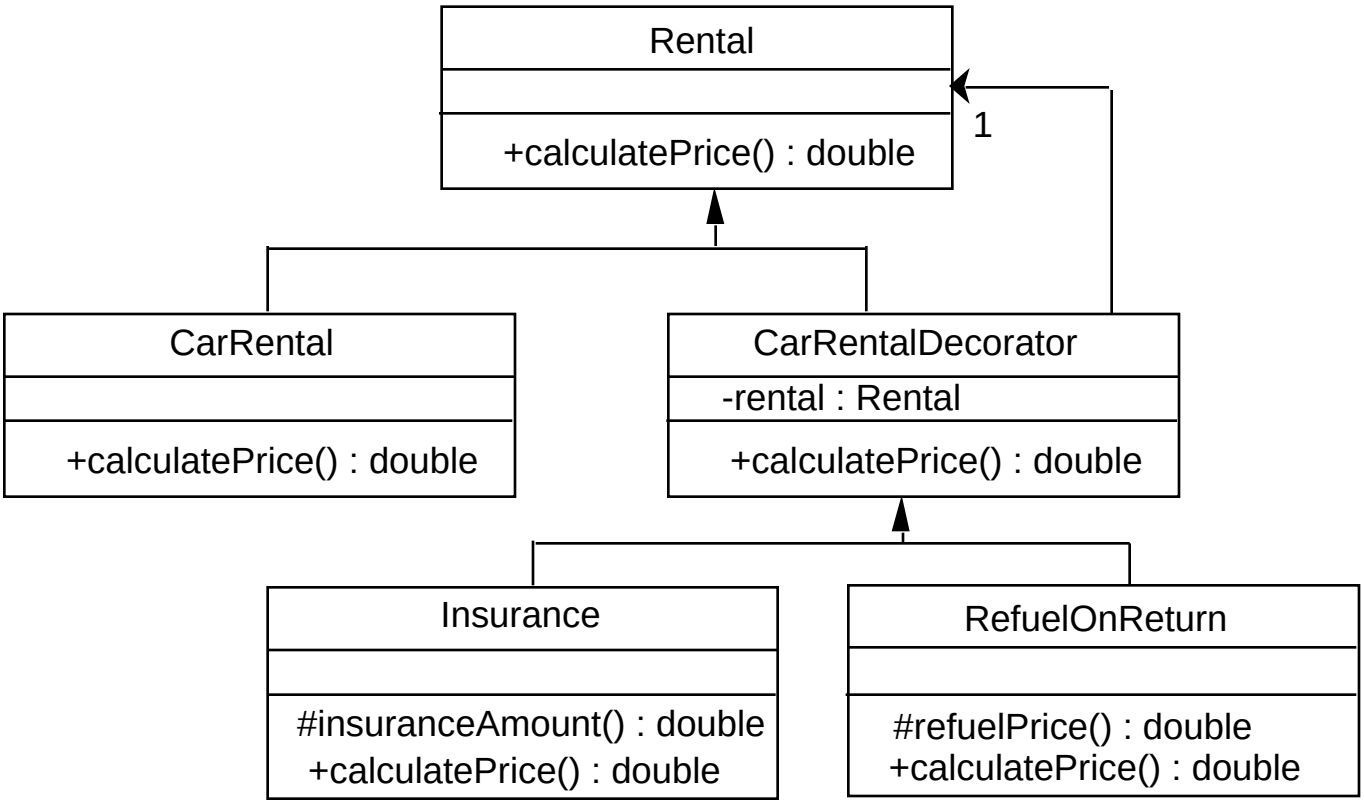
Keep only normal case logic in the class and extract the special case logic into decorators

Example

```
Class CarRental {
   public double calculatePrice() {
      double price = _model.price() * _days;
      if (_hasInsurance)
         price += insuranceAmount();
      if (_neededRefuelOnReturn)
         price += refuelPrice();
      return price;
   }
}
```

[8] Kerievsky 2000, pp. 21-29

Refactored Classes



Decorator Code

```
class Insurance extends CarRentalDecorator {
   public double calculatePrice() {
      return rental.calculatePrice() + insuranceAmount();
   }

   private double insuranceAmount() {
      blah
   }
}

class RefuelOnReturn extends CarRentalDecorator {
   public double calculatePrice() {
      return rental.calculatePrice() + refuelPrice();
   }

   private double refuelPrice() {
      blah
   }
}
```