Code Smells i Refactoring

Code Smells i Refactoring

- Definition
- Code Smells in Code
- Refactoring Techniques
- Common Refactorings
- Eclipse Refactorings
- Applying Refactorings to Code Smells
- Refactoring to Patterns
- Testing with Refactoring
- Laboratory Use Case
- References

Definition

- **Refactoring** is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

- Performing refactoring step-by-step and running tests after each change are key elements of refactoring that make it predictable and safe.

- Refactoring requires:
  - Detecting bad smells in code
  - Determining and applying refactorings techniques or steps.

Code Smells in Code

- A **code smell** (or bad smell in code) is a surface indication that usually corresponds to a deeper problem in the system.

- A smell is by definition something that's quick to spot (for instance, a long method). Just looking at the code we can see if there are more than a dozen lines of Java.

- A smells don't *always* indicate a problem (for instance, some long methods are just fine).

Code Smells in Code

Taxonomy of Bad Smells in Code

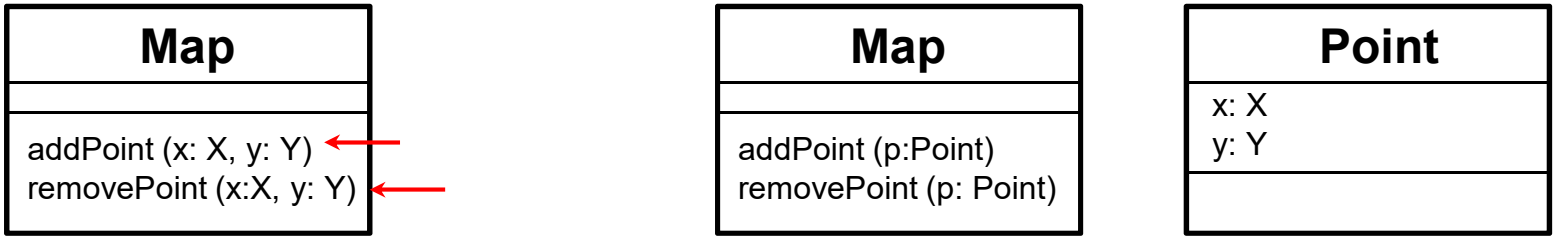
Group Name	Description	Code Smell Name
Bloaters	Methods and classes have increased to such proportions that they are hard to work with.	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps
Object-Orientation Abusers	All these smells are incomplete or incorrect application of object-oriented programming principles.	Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces
Change Preventers	Changing something in one place in your code imply many changes in other places too.	Divergent Change, Shotgun Surgery, Parallel Inheritance Hierarchies
Dispensables	Something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.	Comments, Duplicate Code, Lazy Class, Data Class, Speculative Generality
Couplers	All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.	Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man, Incomplete Library Class

Code Smells in Code

BLOATERS

Data Clumps

Different parts of the code contain identical groups of variables.



OO ABUSERS

Switch Statement

You have a complex switch operator or sequence of if statements.

CHANGE PREVENTERS

Shotgun Surgery

Making any modifications requires that you make many small changes to many different classes.

Code Smells in Code

DISPENSABLES

Comments

A method is filled with explanatory comments.

```
….
/* Convert dollars to euros*/
 e = d * r;
```

```
….
amountInEuros = AmountInDollars * exchangeRate;
```

COUPLERS

Feature Envy

A method accesses the data of another object more than its own data.

```
public class Customer {
  private Address currentAddress = null;
public String MailingAddress() {
   String mailingAddress = currentAddress.getCity() + " " + currentAddress.getCountry(); } }
```

```
public class Customer {
  private Address currentAddress = null;
  public String MailingAddress() {
    String mailingAddress = currentAddress.
                getMailingAddress(); } }
```

```
public class Address {
  private String city;
  private String country;
  public String MailingAddress() {
    String mailingAddress = this.getCity() +
                " " + this.getCountry(); }}
```

Refactoring Techniques

- **Refactoring techniques** describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

Refactoring Techniques

- There are several refactoring catalogs such as:

http://refactoring.com/catalog/

https://industriallogic.com/xp/refactoring/catalog.html

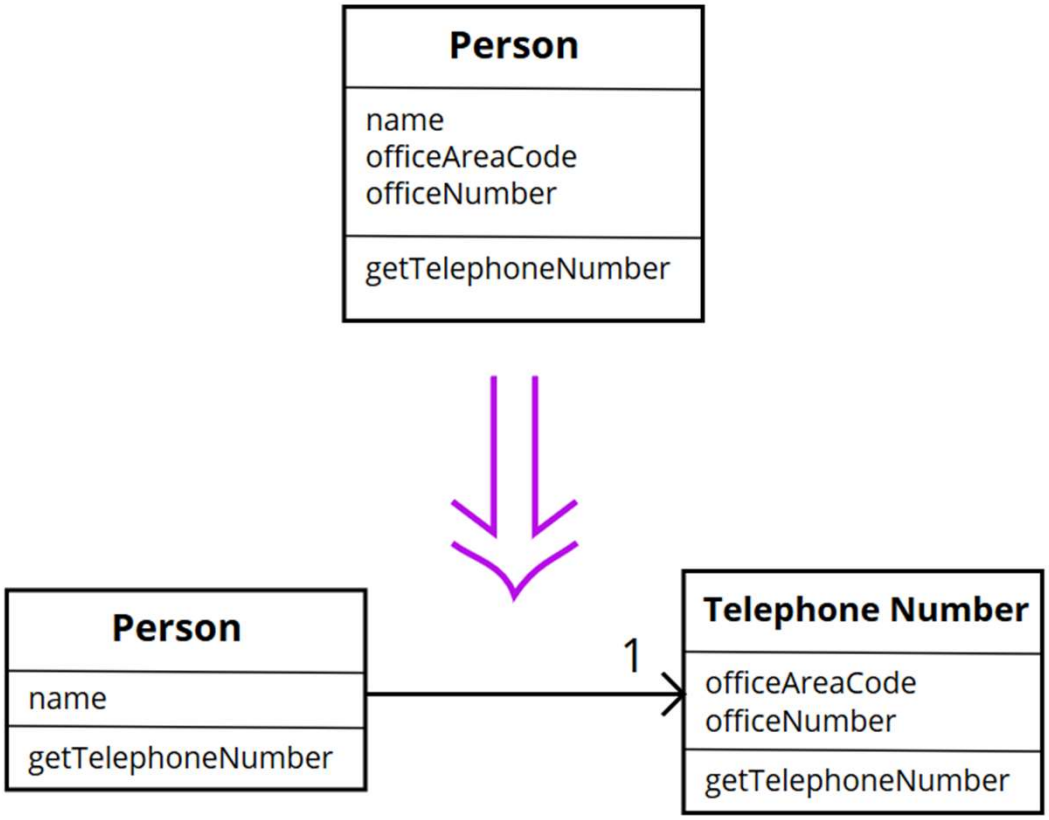
- There exists a relationship between code smells and refactorings:

https://www.industriallogic.com/img/blog/2005/09/smellstorefactorings.pdf

Common Refactorings

Extract Class

A class doing work that should be done by two



Common Refactorings

Extract Method

A code fragment that can be grouped together

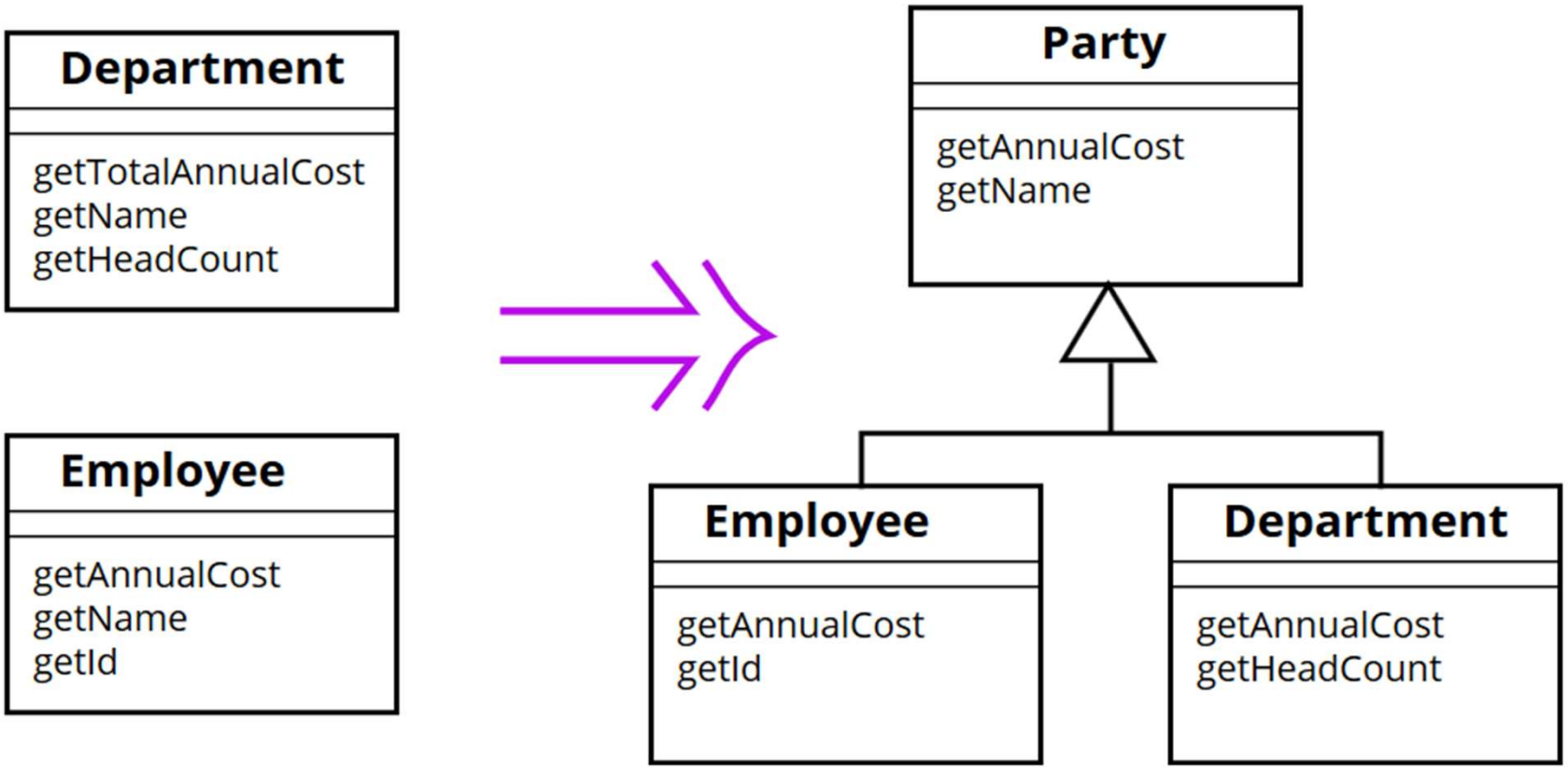
```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:   " + _name);
  System.out.println ("amount " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

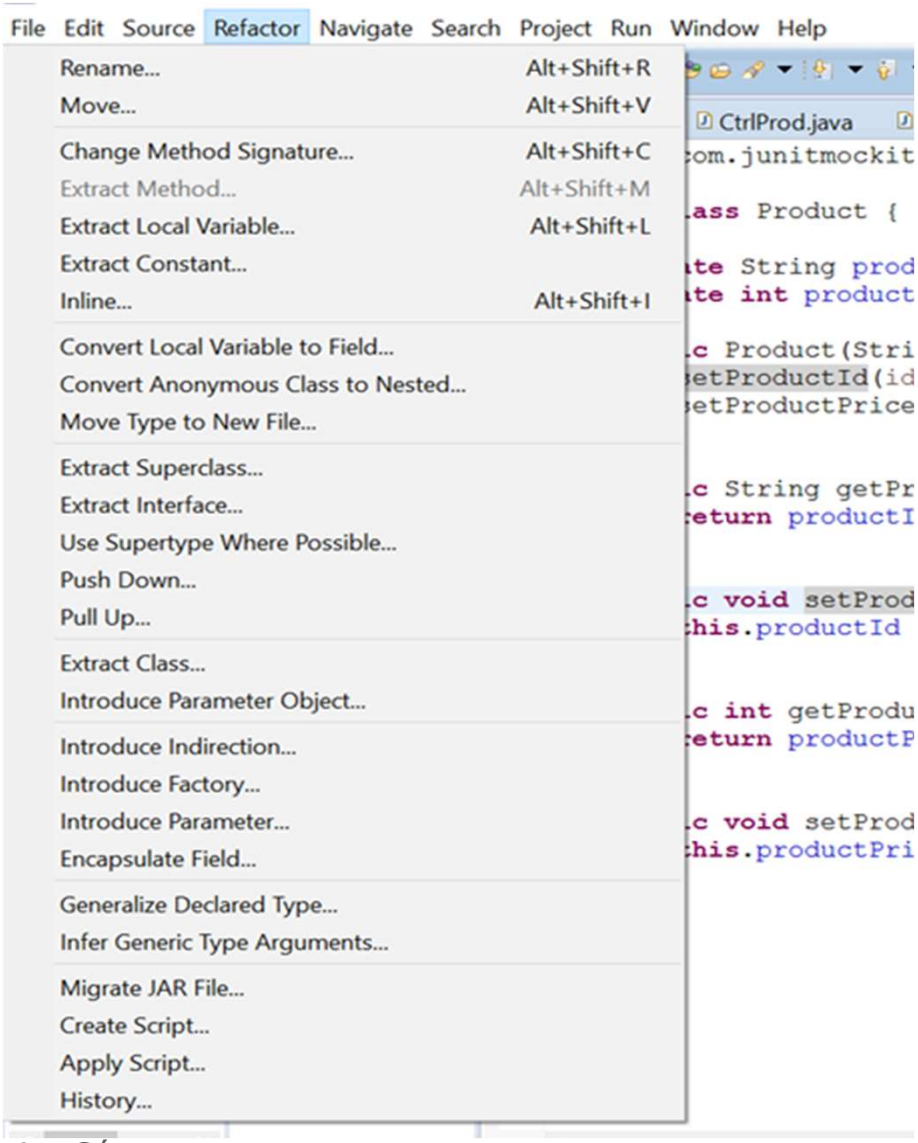
Common Refactorings

Extract Superclass

Two classes with similar features



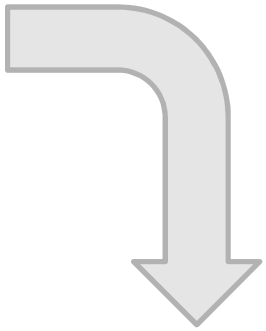
Eclipse Refactorings



Applying Refactorings to Code Smells

Long Method
A method contains too many lines of code (generally longer than ten).

```
public void debit (float amount) {
  // Deduct amount from balance
  balance -=amount;
  // Record transaction
  transactions.add(new Transaction(true, amount));
  // Update last debit date
  Calendar calendar = Calendar.getInstance();
  lastDebitDate = calendar.get(calendar.DATE) + " / " +
                  calendar.get(calendar.MONTH) + " / " +
                  calendar.get(calendar.YEAR);  }
```



Extract Method

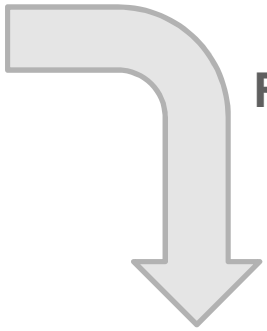
```
public void debit (float amount) {
  deductAmountFromBalance(amount);
  recordTransaction(true, amount);
  updateLastDebitDate(); }
public void deductAmountFromBalance(amount) {…}
public void recordTransaction(isDebit, amount) {…}
public void updateLastDebitDate() {…}
```

Applying Refactorings to Code Smells

Switch Statement

You have a complex switch operator or sequence of if statements.

```
class Bird {
  //...
  double getSpeed()  {
     switch (type) {
          case EUROPEAN:
               return getBaseSpeed();
          case AFRICAN:
               return getSpeed() * weight;
          case NORWEGIAN_BLUE:
               return 50; } } }
```



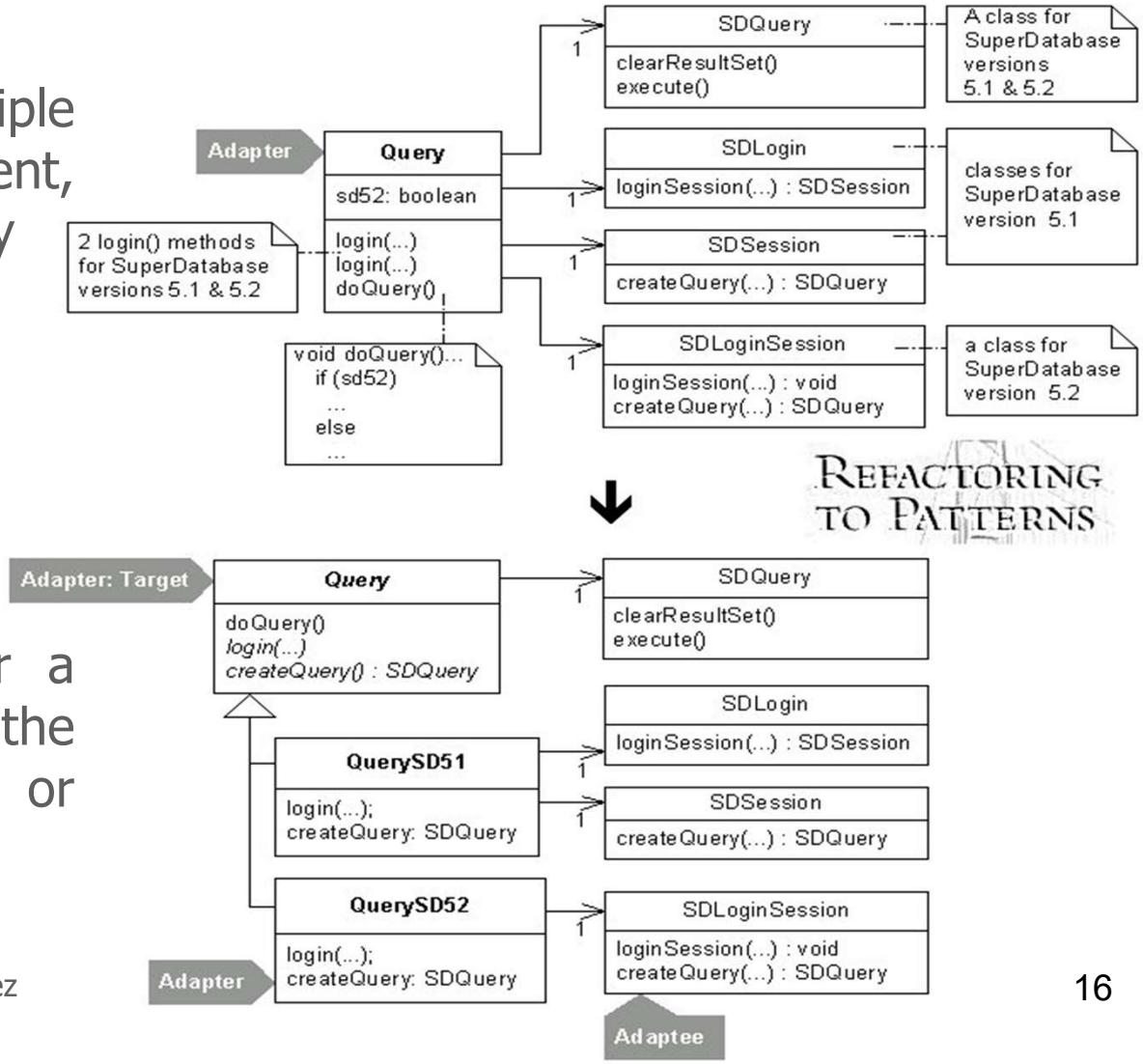
Replace Conditional with Polymorphism

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

class European extends Bird {
  double getSpeed() { return getBaseSpeed(); } }

class African extends Bird {
  double getSpeed() { return getSpeed() * weight; } }

class NorvegianBlue extends Bird {
  double getSpeed() { return 50; } }
```

Refactoring to Patterns

Extract Adapter

One class adapts multiple versions of a component, library, API or other entity

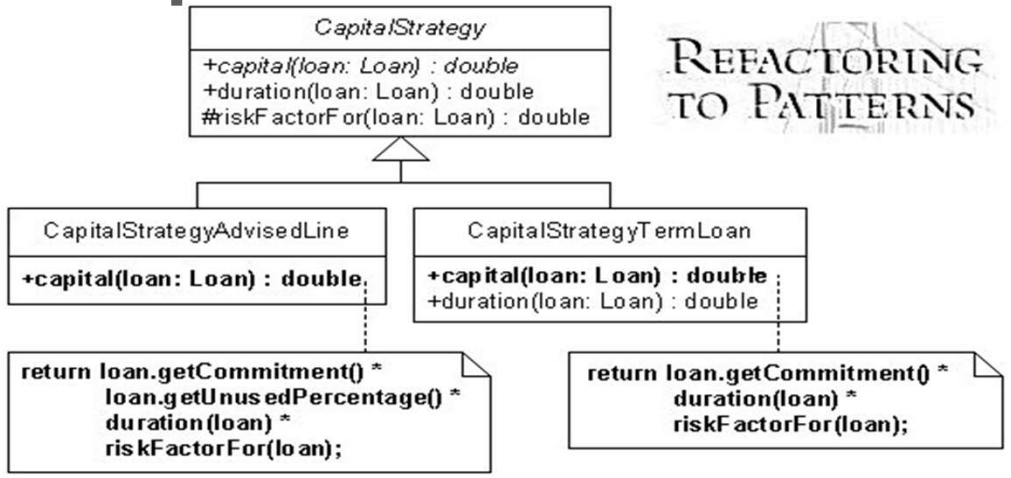


Extract an **Adapter** for a single version of the component, library, API or other entity

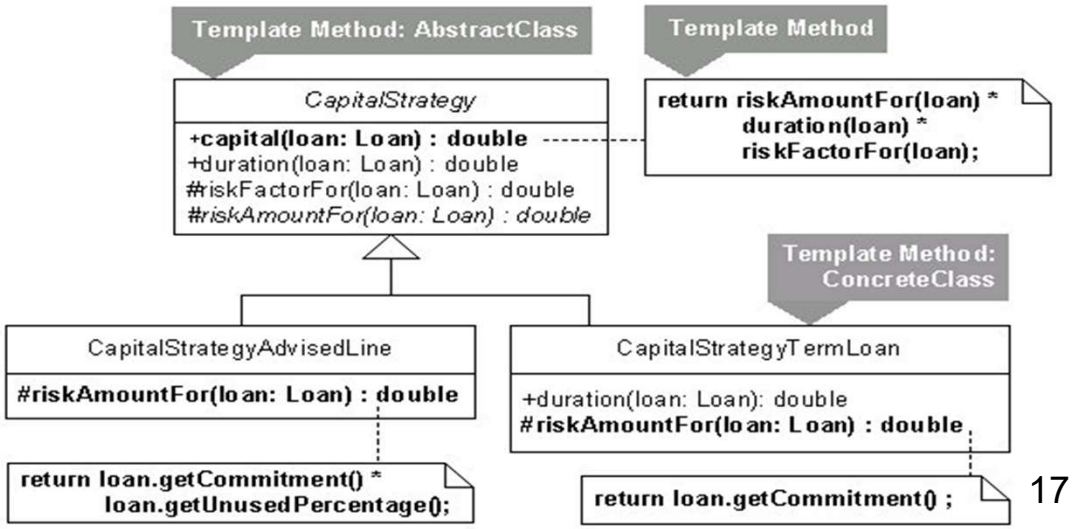
Refactoring to Patterns

Form Template Method

Two methods in subclasses perform similar steps in the same order, yet the steps are different



Generalize the methods by extracting their steps into methods with identical signatures, then pull up the generalized methods to form a **Template Method**

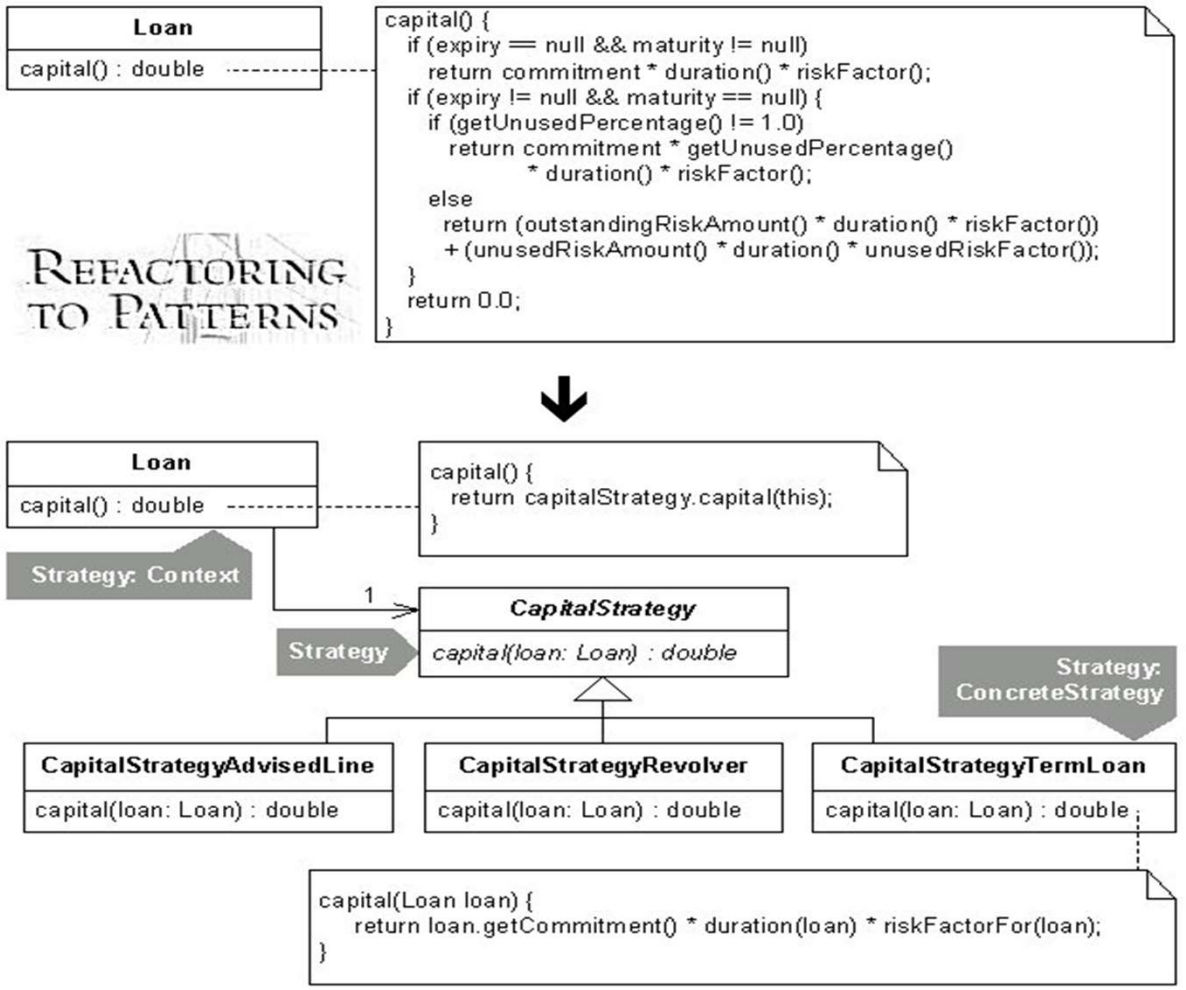


Refactoring to Patterns

Replace Conditional Logic with Strategy

Conditional logic in a method controls which of several variants of a calculation are executed.

Create a **Strategy** for each variant and make the method delegate the calculation to a Strategy instance.



Laboratory Use Case

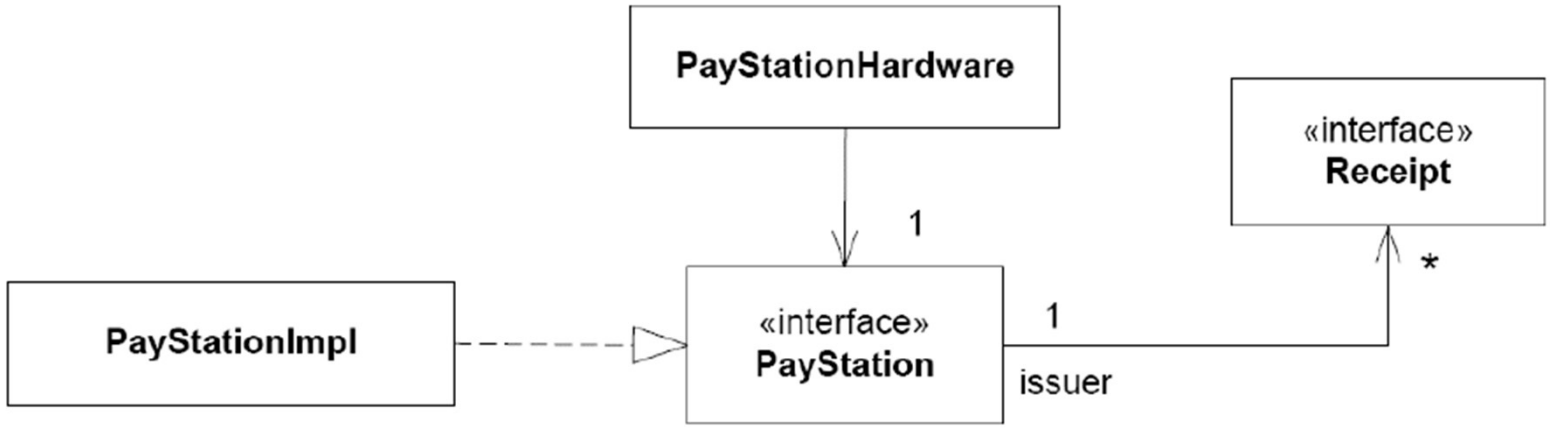
Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

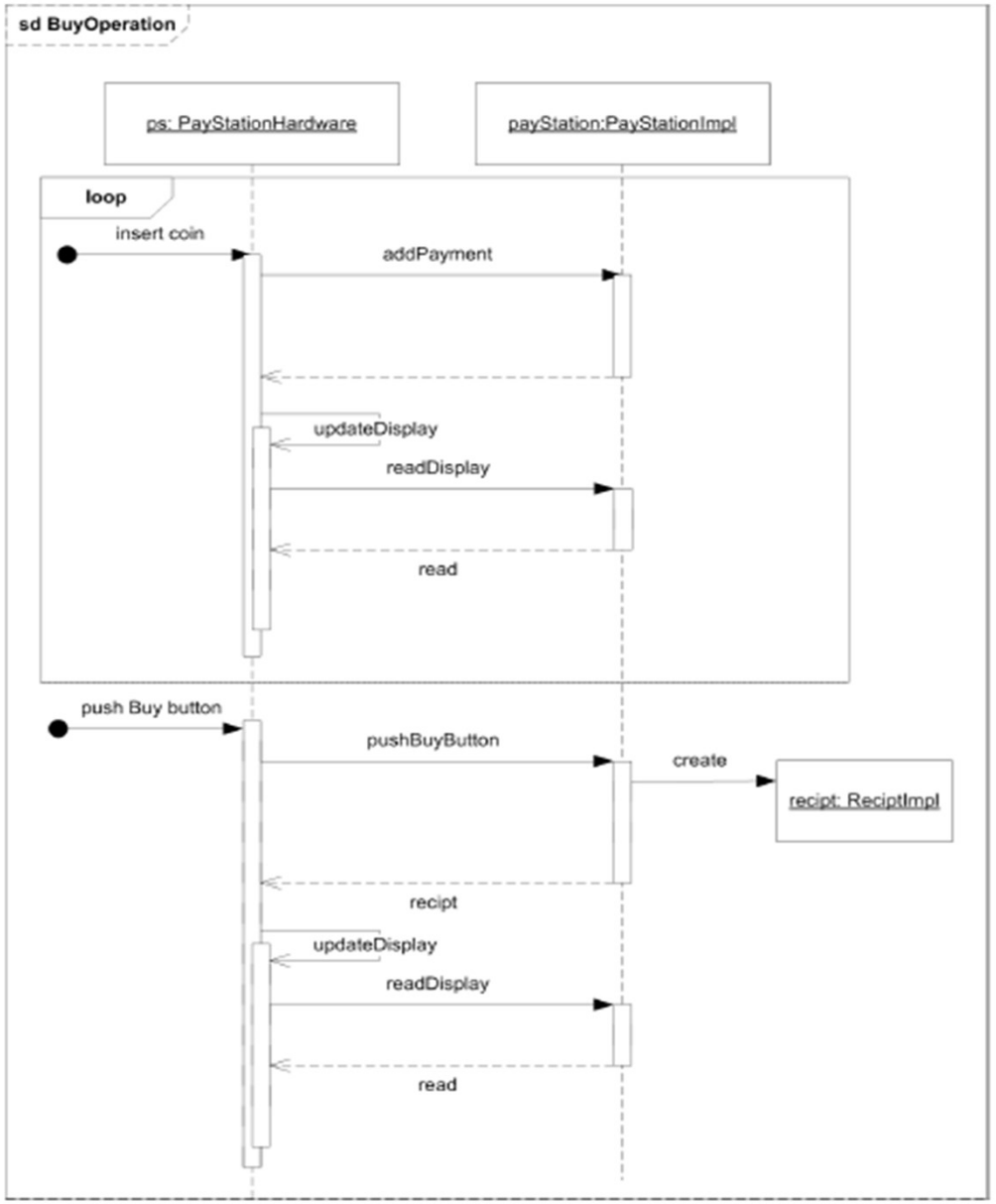
*Example extracted from: Flexible, Reliable Software: Using Patterns and Agile Development. Henrik B. Christensen

Laboratory Use Case



*Example extracted from: Flexible, Reliable Software: Using Patterns and Agile Development. Henrik B. Christensen

Laboratory Use Case



*Example extracted from: Flexible, Reliable Software: Using Patterns and Agile Development. Henrik B. Christensen

References

- *Flexible, Reliable Software: Using Patterns and Agile Development.* Henrik B. Christensen. CRC Press
- *Refactoring. Improving the Design of Existing Code.* M. Fowler (with Kent Beck, John Brant, William Opdyke, and Don Roberts). Addison Wesley, 1999
- *Refactoring to Patterns.* J. Kerievsky. Addison Wesley, 2004
- *https://sourcemaking.com/refactoring*
- *https://refactoring.guru/es/refactoring*
- *Mocks Aren't Stubs.* Martin Fowler. http://martinfowler.com/articles/mocksArentStubs.html