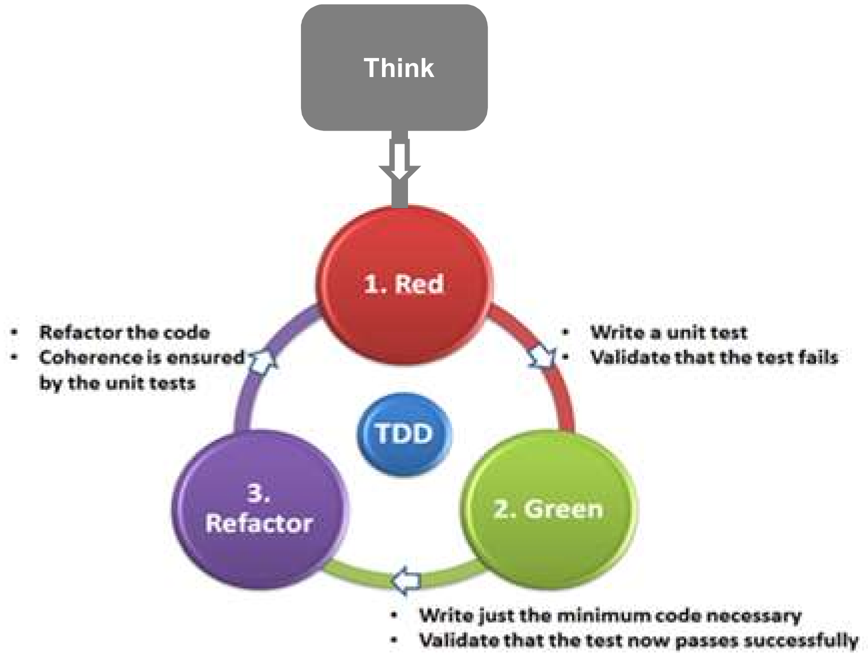
Refactoring Code in TDD

Refactoring Code in TDD

- Test Driven Development
- Fourth Step: Refactoring
  - Bad Smells in Code
  - Refactorings
- References

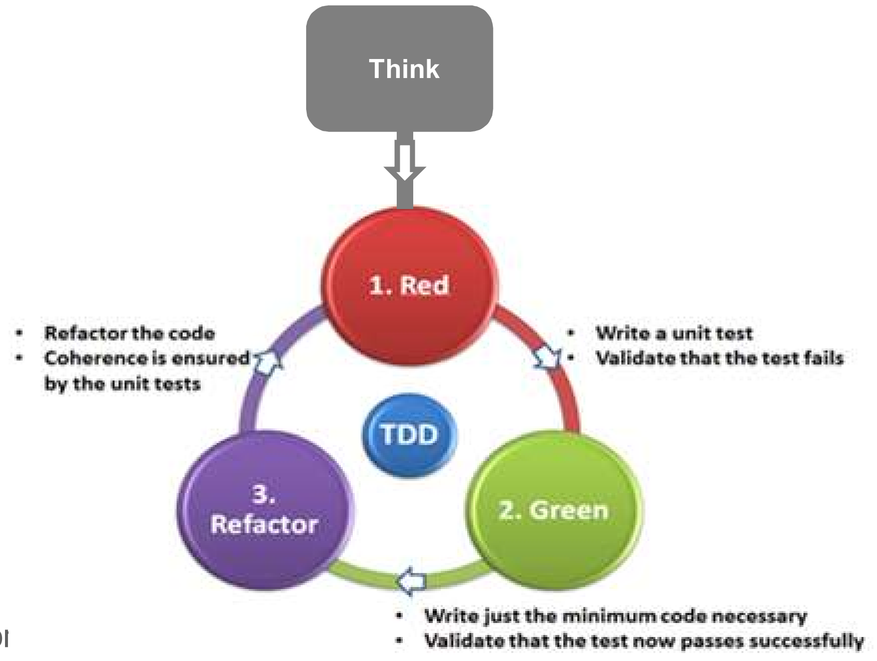
Test Driven Development

- TDD is an approach that drive the design of software.
- **First Step: Think.** Think of a small increment that will require fewer than five lines of code and think of a test that will fail unless that behavior is present.
- **Second Step: Red bar.** Write the test in terms of the class' behavior and its public interface, run it and watch the new fail.


Think
1. Red
Write a unit test
Validate that the test fails
TDD
2. Green
Write just the minimum code necessary
Validate that the test now passes successfully
3. Refactor
Refactor the code
Coherence is ensured by the unit tests

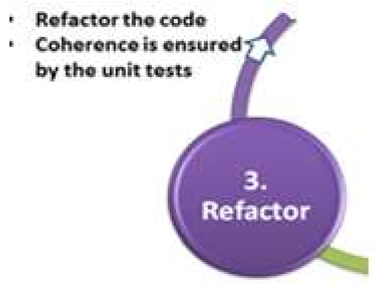
Test Driven Development

- **Third Step: Green bar.** Write just the enough production code to get the test to pass. Run again and watch all tests pass.
- **Fourth Step: Refactor.** Review the code for improvements and apply small refactorings. Run again and watch all tests pass.


Think
1. Red
Write a unit test
Validate that the test fails
TDD
2. Green
Write just the minimum code necessary
Validate that the test now passes successfully
3. Refactor
Refactor the code
Coherence is ensured by the unit tests

Fourth Step: Refactoring

- **Fourth Step: Refactor.** Inspect the code and detect bad smells in it. Then, select and apply the appropriate refactoring/s. Run again and watch all tests pass.



- In this step, we focus on:
  - Detecting bad smells in code
  - Determining and applying refactorings

Fourth Step: Refactoring. Bad 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).

Fourth Step: Refactoring. Bad 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, Inapppropriate Intimacy, Message Chains, Middle Man, Incomplete Library Class

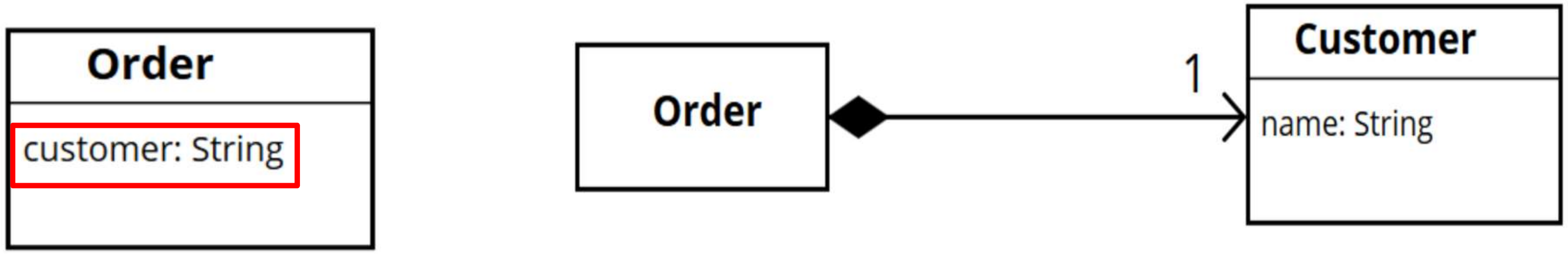
Fourth Step: Refactoring. Bad Smells in Code

BLOATERS

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

Large Class
A class contains many fields/methods/lines of code.

Primitive Obsession
Use of primitives instead of small objects for simple tasks, use of constants and use of string constants as field names for use in data arrays.

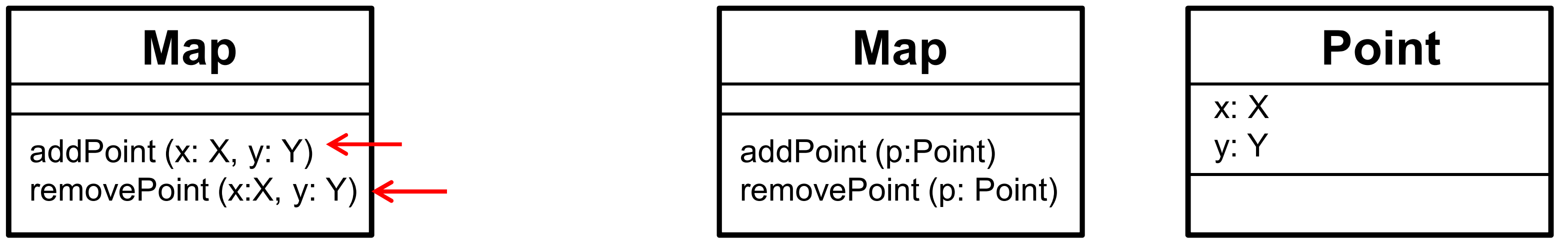


Fourth Step: Refactoring. Bad Smells in Code

BLOATERS

Long Parameter List
More than three or four parameters for a method.

Data Clumps
Different parts of the code contain identical groups of variables.



OBJECT ORIENTATION ABUSERS

Fourth Step: Refactoring. Bad Smells in Code

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

Temporary Field
Temporary fields get their values only under certain circumstances. Outside of these circumstances, they are empty.

Refused Bequest
If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.

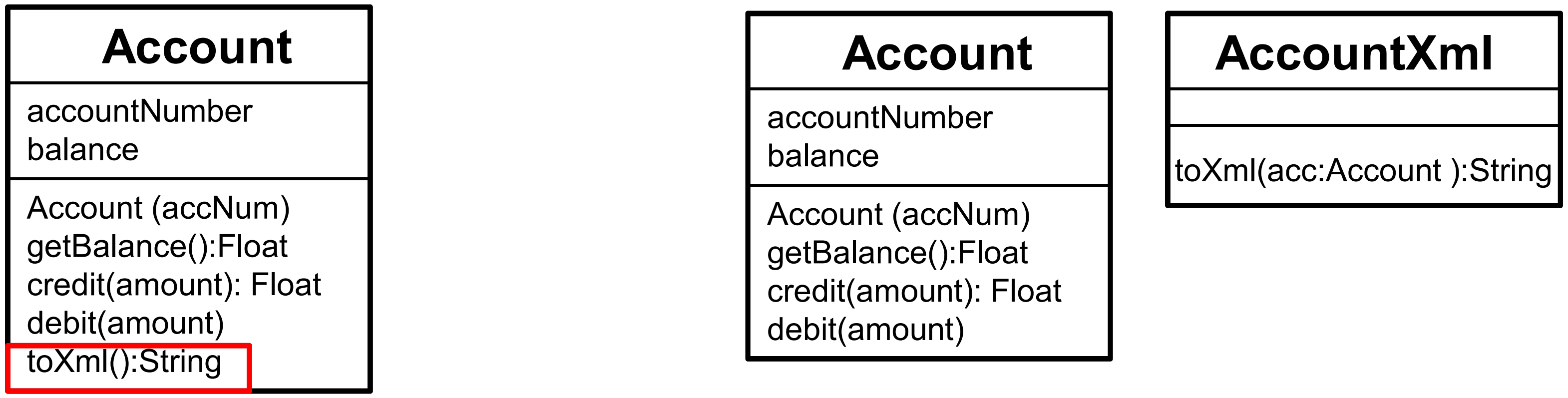
Alternative Classes With Different Interfaces
Two classes perform identical functions but have different method names.

Fourth Step: Refactoring. Bad Smells in Code

CHANGE PREVENTERS

Divergent Change

Divergent change occurs when one class is commonly changed in different ways for different reasons.



Shotgun Surgery

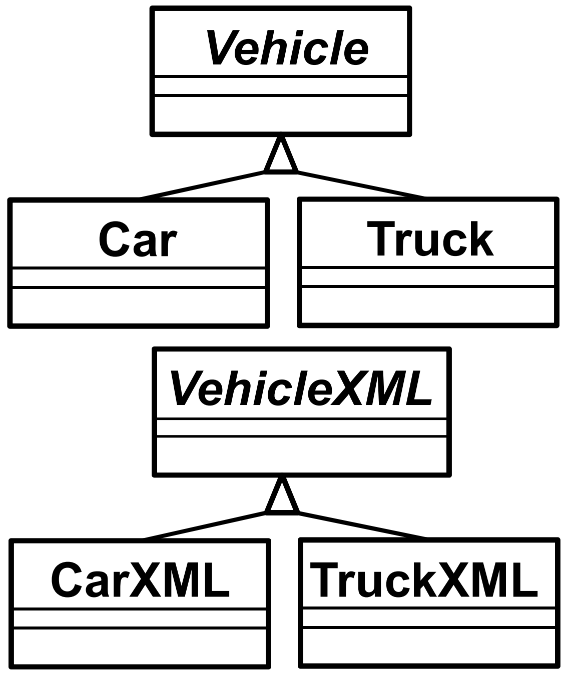
Making any modifications requires that you make many small changes to many different classes.

Fourth Step: Refactoring. Bad Smells in Code

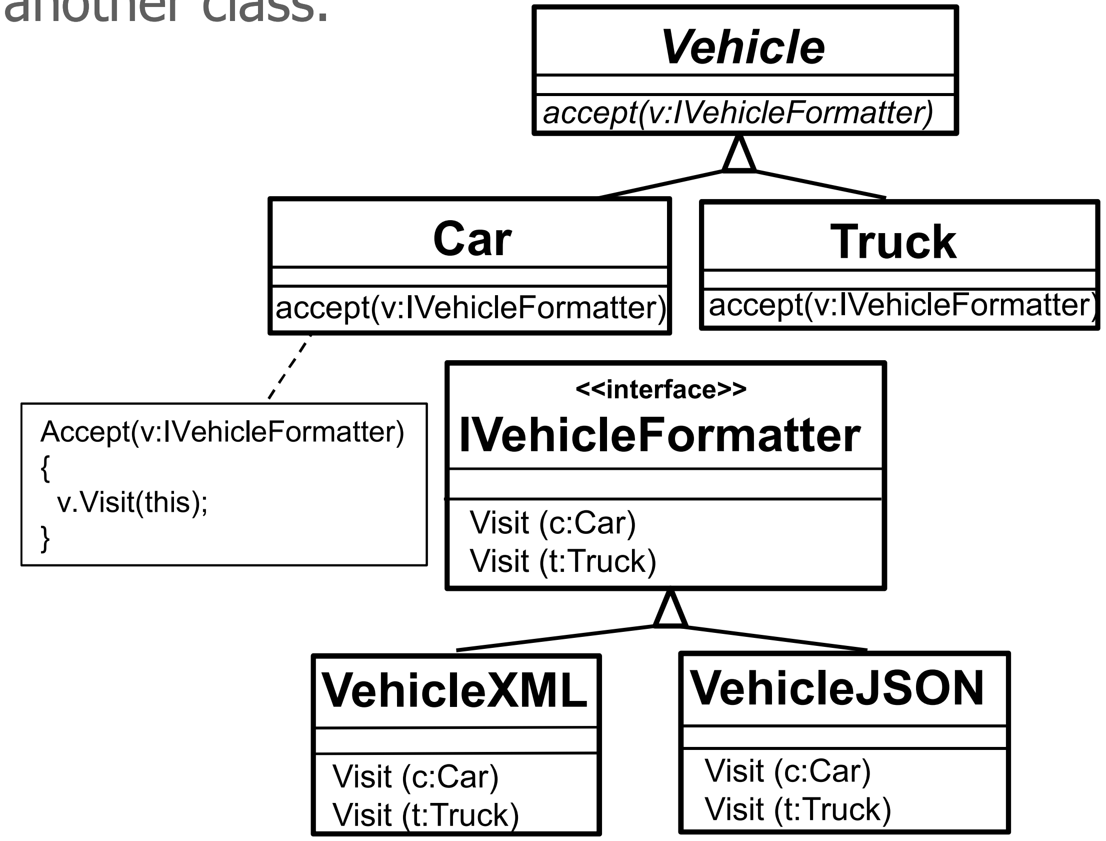
CHANGE PREVENTERS

Parallel Inheritance Hierarchies

Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.



JSON Format?



Fourth Step: Refactoring. Bad Smells in Code

DISPENSABLES

Comments

A method is filled with explanatory comments.

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

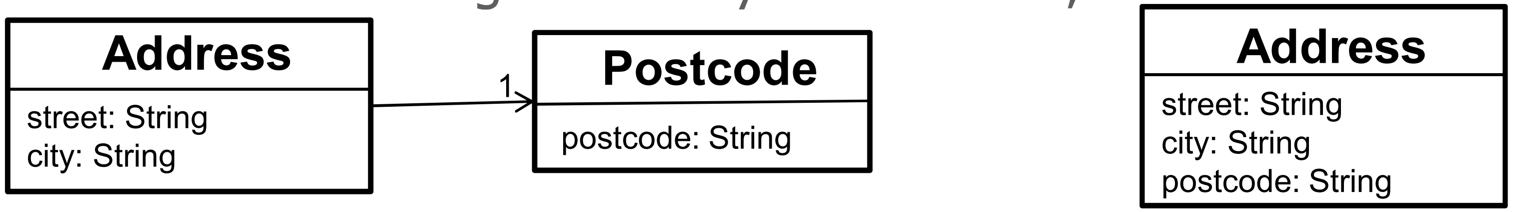
```
….
amountInEuros = AmountInDollars * exchangeRate;
```

Duplicated Code

Two code fragments look almost identical.

Lazy Class

Understanding and maintaining classes costs time and money. If a class doesn't do enough to earn your attention, it should be deleted.



Speculative Generality

There is an unused class, method, field or parameter.

Fourth Step: Refactoring. Bad Smells in Code

DISPENSABLES

Data Class

A data class refers to a class that contains only fields and crud methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own.

```
public class CustomerSummaryView {
  private Customer customer;
  public String getCustomerSummary()  {
    Address address = customer.getAddress();
    return customer.getFirstName() + " " + customer.getLastName() + ", " + address.getCity() + ", " +
           address.getCountry(); }  }
```

```
public class Customer {
  private String firstName;
  private String lastName;
  private Address address;
  public String getFirstName() {…}
  public String getLastName() {…}
  public Address getAddress() {…} }
```

```
public class Address {
  private String city;
  private String country;
  public String getCity() {…}
  public String getCountry() {…} }
```

Fourth Step: Refactoring. Bad Smells in Code

DISPENSABLES

Data Class

A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own.

```
public class Customer {
  private String firstName;
  private String lastName;
  private Address address;
  public String getFirstName() {…}
  public String getLastName() {…}
  public Address getAddress() {…}
  public String getCustomerSummary()  {
     return getFirstName() + " " + getLastName() + ", " + address.getAddressSummary(); } }
```

```
public class Address {
  private String city;
  private String country;
  public String getAddressSummary() {
     return city + ", " + country; }}
```

Fourth Step: Refactoring. Bad Smells in Code

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(); }}
```

Fourth Step: Refactoring. Bad Smells in Code

COUPLERS

Inappropriate Intimacy
One class uses the internal fields and methods of another class.

Message Chains
In code you see a series of calls resembling a.b().c().d().

Middle Man
If a class performs only one action, delegating work to another class, why does it exist at all?

Fourth Step: Refactoring

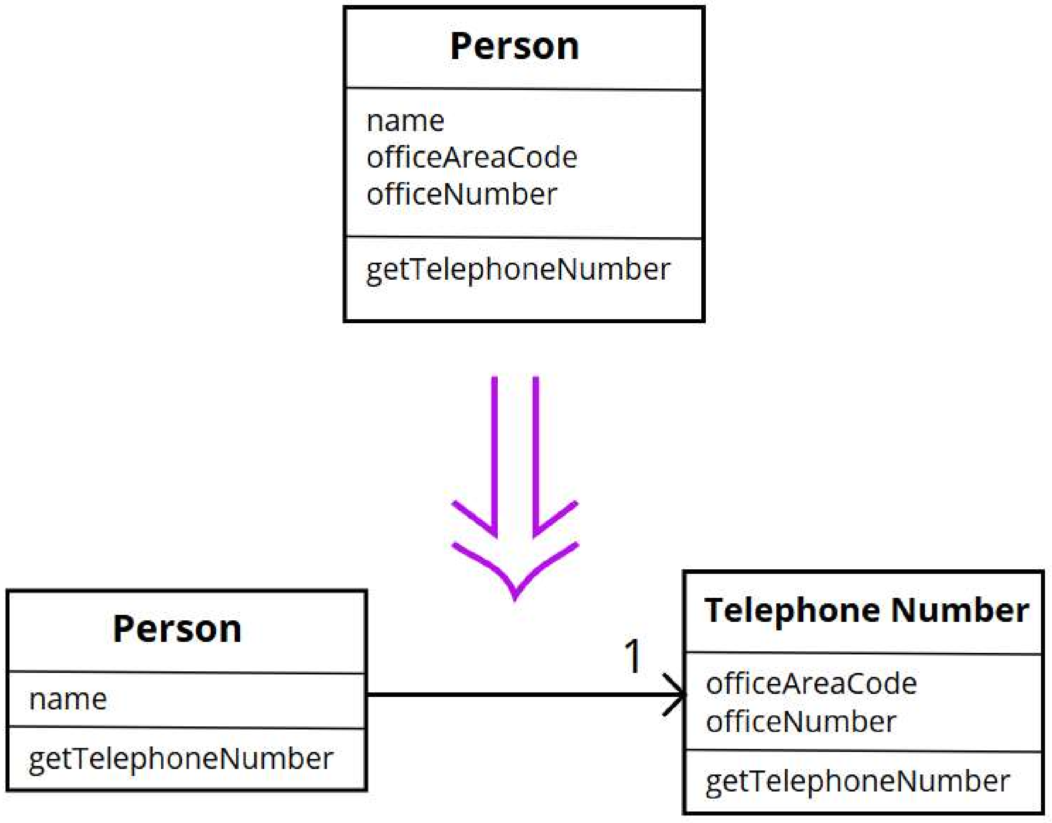
- **Refactoring** is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [Fow99].

- It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.

Fourth Step: Refactoring. Example

Extract Class

A class doing work that should be done by two



Fourth Step: Refactoring. Why?

- **We should refactor because …**
- Refactoring Improves the Design of Software
  - Refactoring is rather like tidying up the code.
- Refactoring Makes Software Easier to Understand
  - A little time spent refactoring can make the code better communicate its purpose, so more readable.
- Refactoring Helps Us Find Bugs
  - For refactoring code it is necessary to work deeply on understanding what the code does. This understanding helps us to find bugs.
- Refactoring Helps Us Program Faster
  - A good design is essential to maintaining speed in software development.

Fourth Step: Refactoring. When?

- **We should refactor when we …**
- Add a function
  - When a design does not help me to add a feature easily, I fix it by refactoring. This make future enhancements easy.
- Need to fix a bug
  - In fixing bugs much of the use of refactoring comes from making code more understandable.
- Do a code review
  - Refactoring also helps the code review have more concrete results. Not only are there suggestions, but also many suggestions are implemented.

Fourth Step: Refactoring. Catalogs

- 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

Fourth Step: Refactoring. Code Smells and Refactorings

Refactorings for Bloaters Code Smells

Code Smells	Refactorings
Long Method	Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object
Large Class	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Primitive Obsession	Replace Data Value with Object, Introduce Parameter Object, Extract Class, Replace Type Code with Class, Replace Type Code with State/Strategy, Replace Type Code with Subclasses, Replace Array With Object
Long Parameter List	Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object
Data Clumps	Extract Class, Preserve Whole Object, Introduce Parameter Object

Fourth Step: Refactoring. 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() {…}
```

Fourth Step: Refactoring. Applying Refactorings to Code Smells

Long Parameter List
More than three or four parameters for a method.

```
 …
// Create an order
Order order = new Order(customerName,
                        customerAddress, customerCity,
                        customerState, customerZip,
                        orderNumber, orderType, orderDate,
                        deliveryDate);
…
```

Introduce Parameter Object

```
 …
  //Create a customer
  Customer customer = new Customer(customerName,
                                   customerAddress, customerCity,
                                   customerState, customerZip);
…
// Create an order
  Order order = new Order(customer, orderNumber,
                          orderType, orderDate, deliveryDate);
  …
```

Fourth Step: Refactoring. Code Smells and Refactorings

Refactorings for Object-Orientation Abusers Code Smells

Code Smells	Refactorings
Switch Statements	Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object
Temporary Field	Extract Class, Introduce Null Object
Refused Bequest	Push Down Field, Push Down Method, Replace Inheritance with Delegation
Alternative Classes with Different Interfaces	Unify Interfaces with Adapter, Rename Method, Move Method

Fourth Step: Refactoring. 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; } }
```

Fourth Step: Refactoring. Code Smells and Refactorings

Refactorings for Change Preventers Code Smells

Code Smells	Refactorings
Divergent Change	Extract Class
Shotgun Surgery	Move Method, Move Field, Inline Class
Parallel Inheritance Hierarchies	Move Method, Move Field

Fourth Step: Refactoring. Applying Refactorings to Code Smells

CHANGE PREVENTERS

Shotgun Surgery

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

```
public SqlDataReader GetData() {
   SqlConnection con = new SqlConnection(ConnectionString);
   SqlCommand cmd = con.CreateCommand();
   con.Open();
   cmd.CommandType = CommandType.StoredProcedure;
   cmd.CommandText = "spSampleStoredProcToGetDataX";
   return cmd.ExecuteReader(CommandBehavior.CloseConnection); }
```

Extract Method

```
public SqlCommand PrepareCommand(string storedProcedureName) {
    SqlConnection con = new SqlConnection(ConnectionString);
    SqlCommand cmd = con.CreateCommand();
    con.Open();
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = storedProcedureName;
    return cmd;  }
public SqlDataReader GetData()  {
    SqlCommand cmd = PrepareCommand("spSampleStoredProc");
    return cmd.ExecuteReader(CommandBehavior.CloseConnection);}
```

Fourth Step: Refactoring. Code Smells and Refactorings

Refactorings for Dispensables Code Smells

Code Smells	Refactorings
Comments	Rename Method, Extract Method, Introduce Assertion
Duplicate Code	Extract Method, Extract Class, Form Template Method, Introduce Null Object, Pull Up Method, Pull Up Field, Substitue Algorithm
Lazy Class	Collapse Hierarchy, Inline Class
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Dead Code	Feature Envy, Inappppropriate Intimacy, Message Chains, Middle Man, Incomplete Library Class
Speculative Generality	Collapse Hierarchy, Rename Method, Remove Parameter, Inline Class

Fourth Step: Refactoring. Applying Refactorings to Code Smells

Duplicated Code

Two code fragments look almost identical.

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

Extract Method

```
public void credit (float amount) {
  balance +=amount;
  transactions.add(new Transaction(false, amount));
  Calendar calendar = Calendar.getInstance();
  lastDebitDate = calendar.get(calendar.DATE) + " / " +
      calendar.get(calendar.MONTH) + " / " +
      calendar.get(calendar.YEAR);  }
```

```
public void debit (float amount) {
  executeAndRecordTransact(-amount); }

public void credit (float amount) {
  executeAndRecordTransact(amount); }

public void executeAndRecordTransact(amount) {
  balance +=amount;
  isDebit = (amount<0);
  transactions.add(new Transaction(isDebit, amount));
  Calendar calendar = Calendar.getInstance();
   lastDebitDate = calendar.get(calendar.DATE)
     + " / " + calendar.get(calendar.MONTH)
     + " / " +  calendar.get(calendar.YEAR);  }
```

Fourth Step: Refactoring. Code Smells and Refactorings

Refactorings for Couplers Code Smells

Code Smells	Refactorings
Feature Envy	Extract Method, Move Method, Move Field
Inappppropriate Intimacy	Move Method, Move Field, Change Bidirectional Association to Unidirectional Association, Extract Class, Hide Delegate, Replace Inheritance with Delegation
Message Chains	Hide Delegate, Extract Method, Move Method
Middle Man	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Incomplete Library Class	Introduce Foreign Method, Introduce Local Extension

Fourth Step: Refactoring. Applying Refactorings to Code Smells

Message Chains

In code you see a series of calls resembling a.b().c().d().

```
public class Invoice {
  …
  if (customer.getAddress().isInEurope()) {…} }
```

Extract Method
Move Method

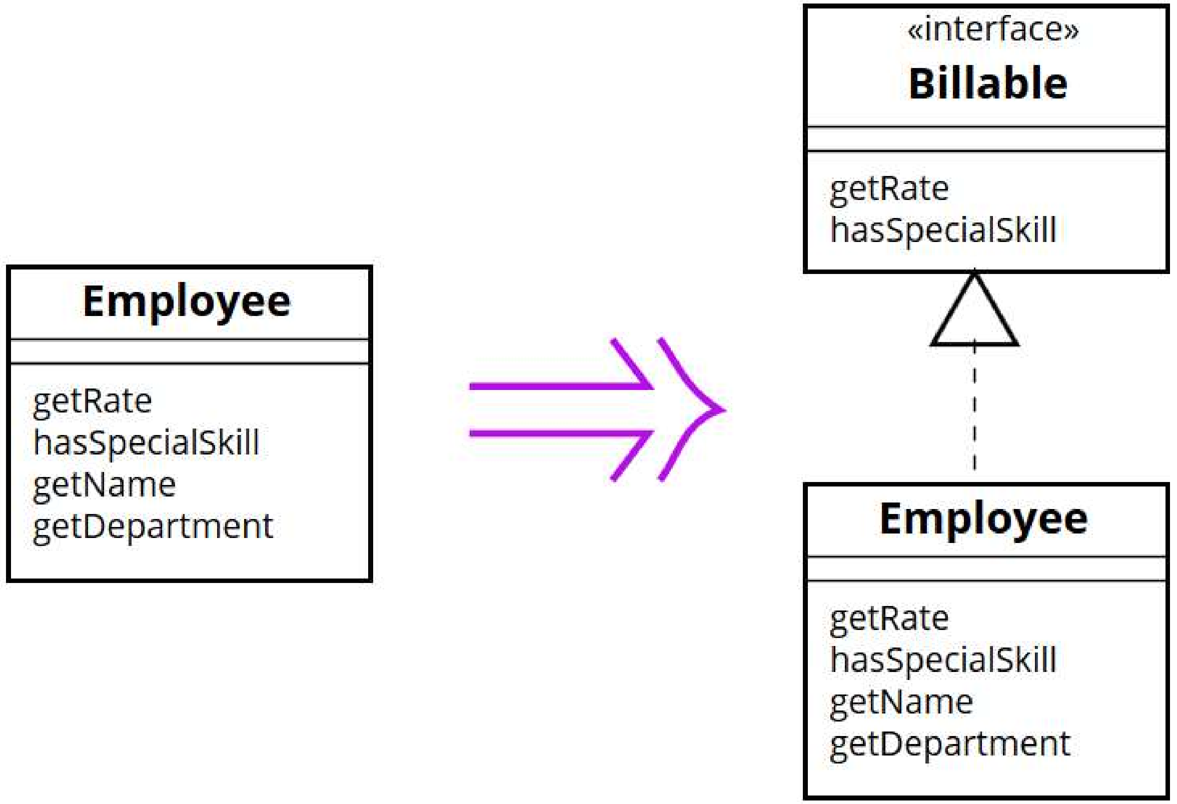
```
public class Invoice {
  …
  if (customer.isInEurope()) {…} }

public class Customer {
  …
  public boolean isInEurope() {
     if (address.isInEurope()) {…} }
```

Fourth Step: Refactoring. Common Refactorings

Extract Interface

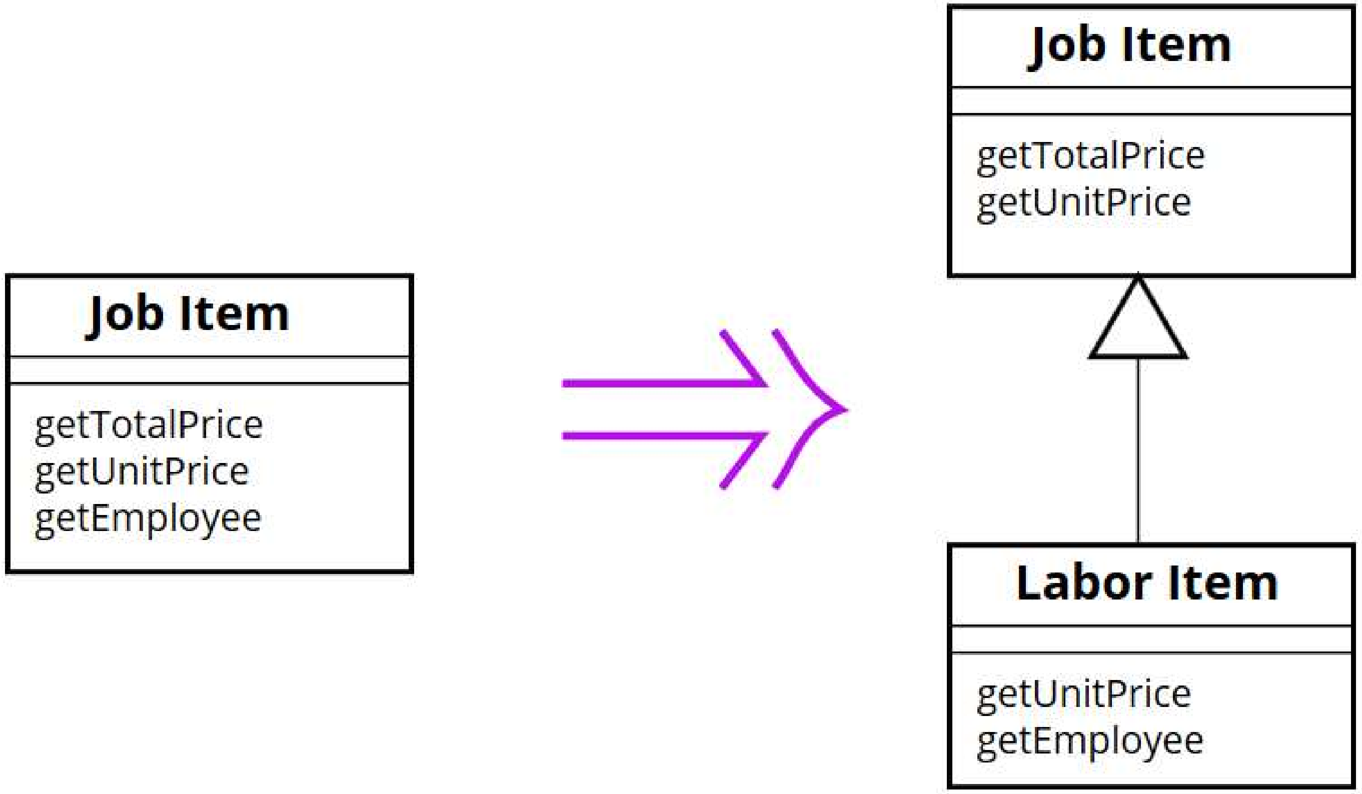
Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common



Fourth Step: Refactoring. Common Refactorings

Extract Subclass

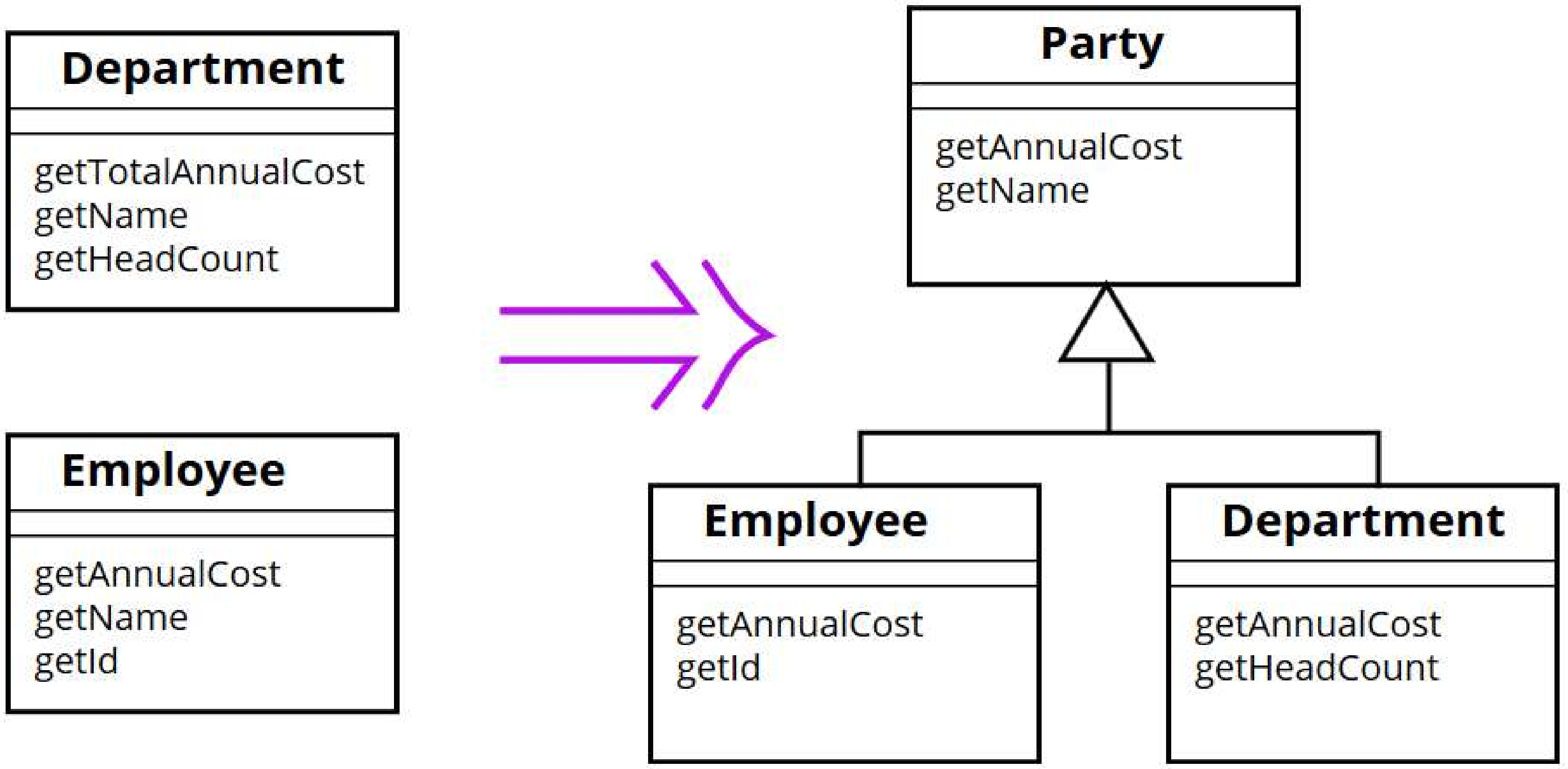
A class has features that are used only in some instances



Fourth Step: Refactoring. Common Refactorings

Extract Superclass

Two classes with similar features



Fourth Step: Refactoring. 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());
}
```

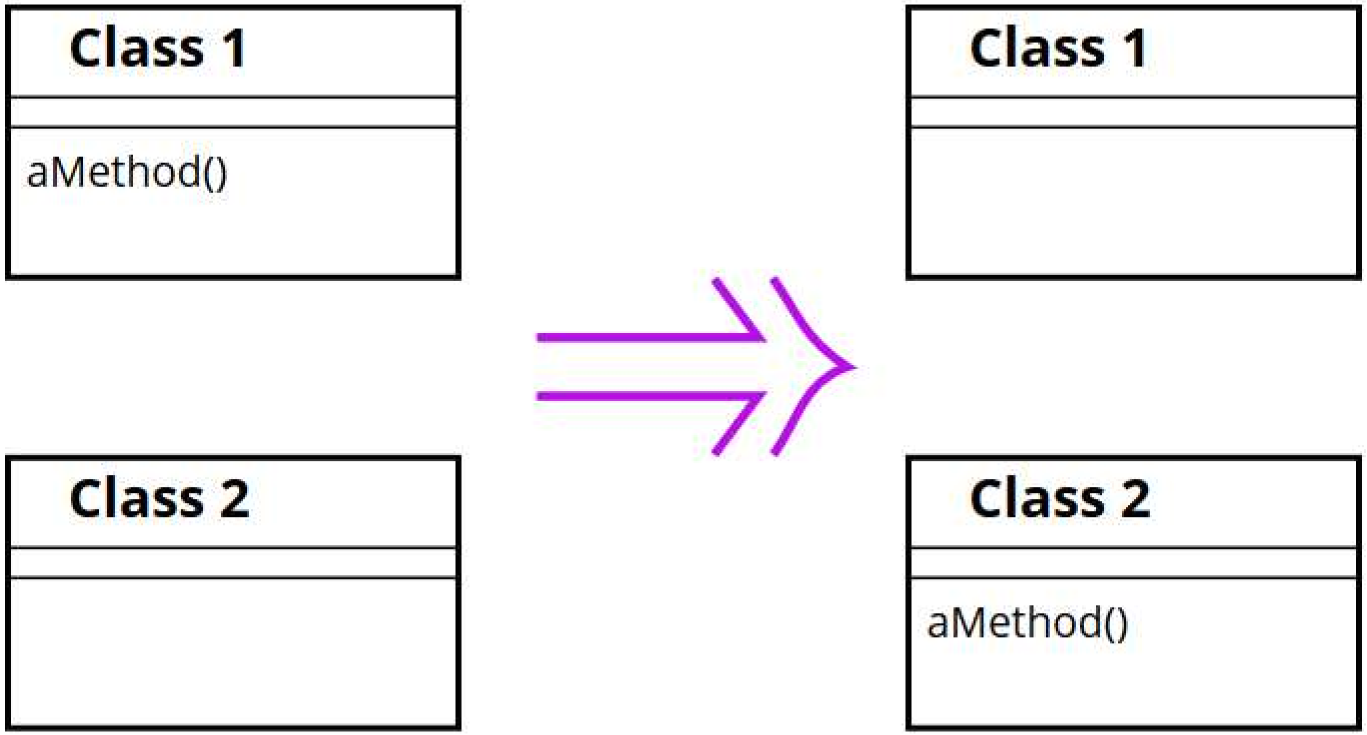
```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

Fourth Step: Refactoring. Common Refactorings

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined



Fourth Step: Refactoring. Common Refactorings

Encapsulate Field

There is a public field

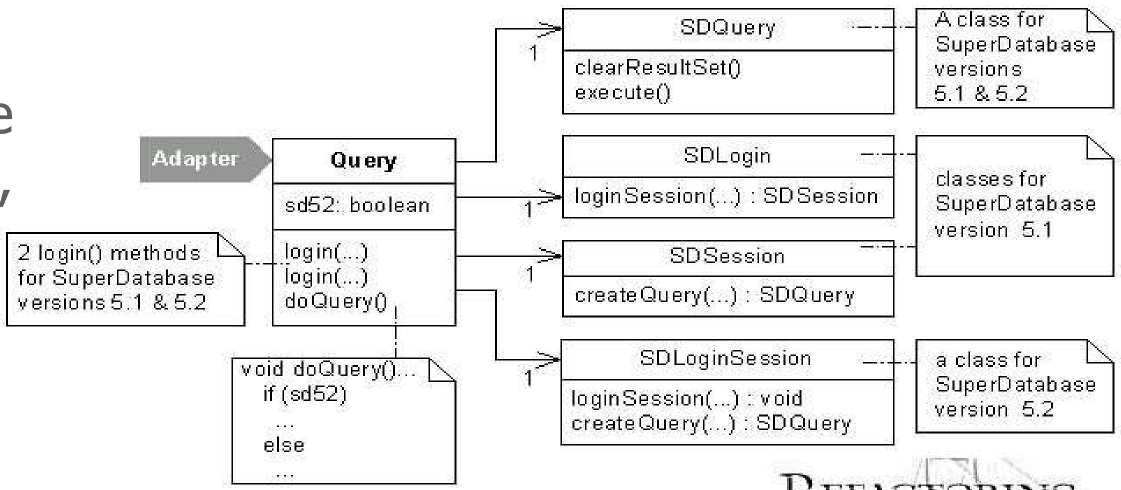
```
public String _name
```

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Fourth Step: Refactoring. Refactorings to Patterns

Extract Adapter

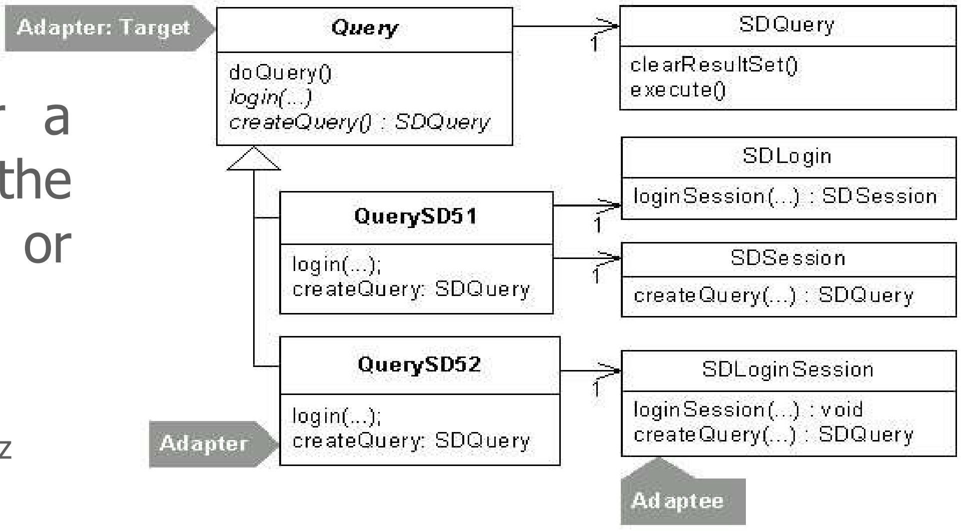
One class adapts multiple versions of a component, library, API or other entity



Adapter
Query
sd52: boolean
login(...)
login(...)
doQuery()
2 login() methods for SuperDatabase versions 5.1 & 5.2
void doQuery()...
if (sd52)
...
else
...
SDQuery
clearResultSet()
execute()
A class for SuperDatabase versions 5.1 & 5.2
SDLogin
loginSession(...) : SDSession
SDSession
createQuery(...) : SDQuery
classes for SuperDatabase version 5.1
SDLoginSession
loginSession(...) : void
createQuery(...) : SDQuery
a class for SuperDatabase version 5.2
1
1
1
1

REFACTORING TO PATTERNS

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

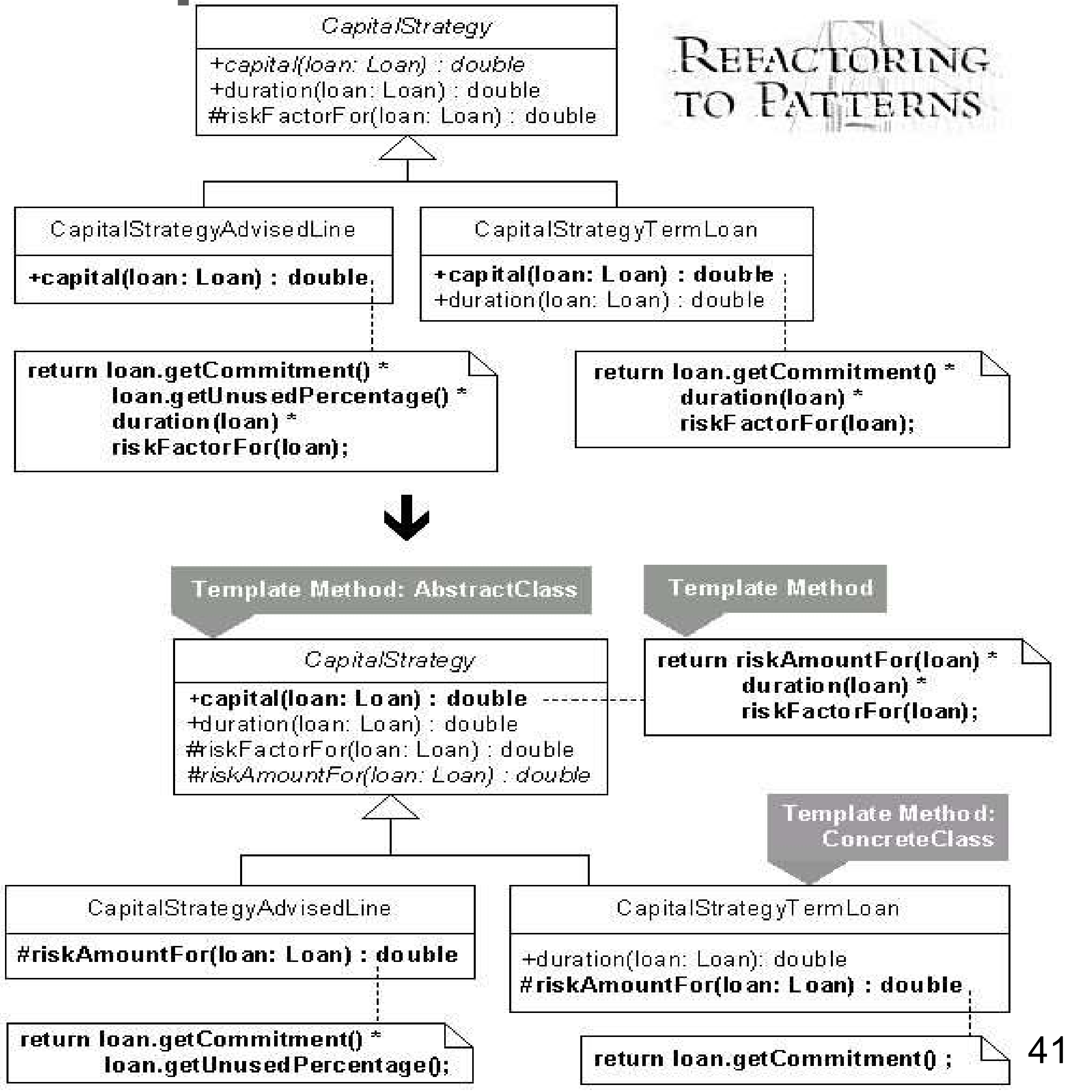


Fourth Step: Refactoring. Refactorings to Patterns

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**



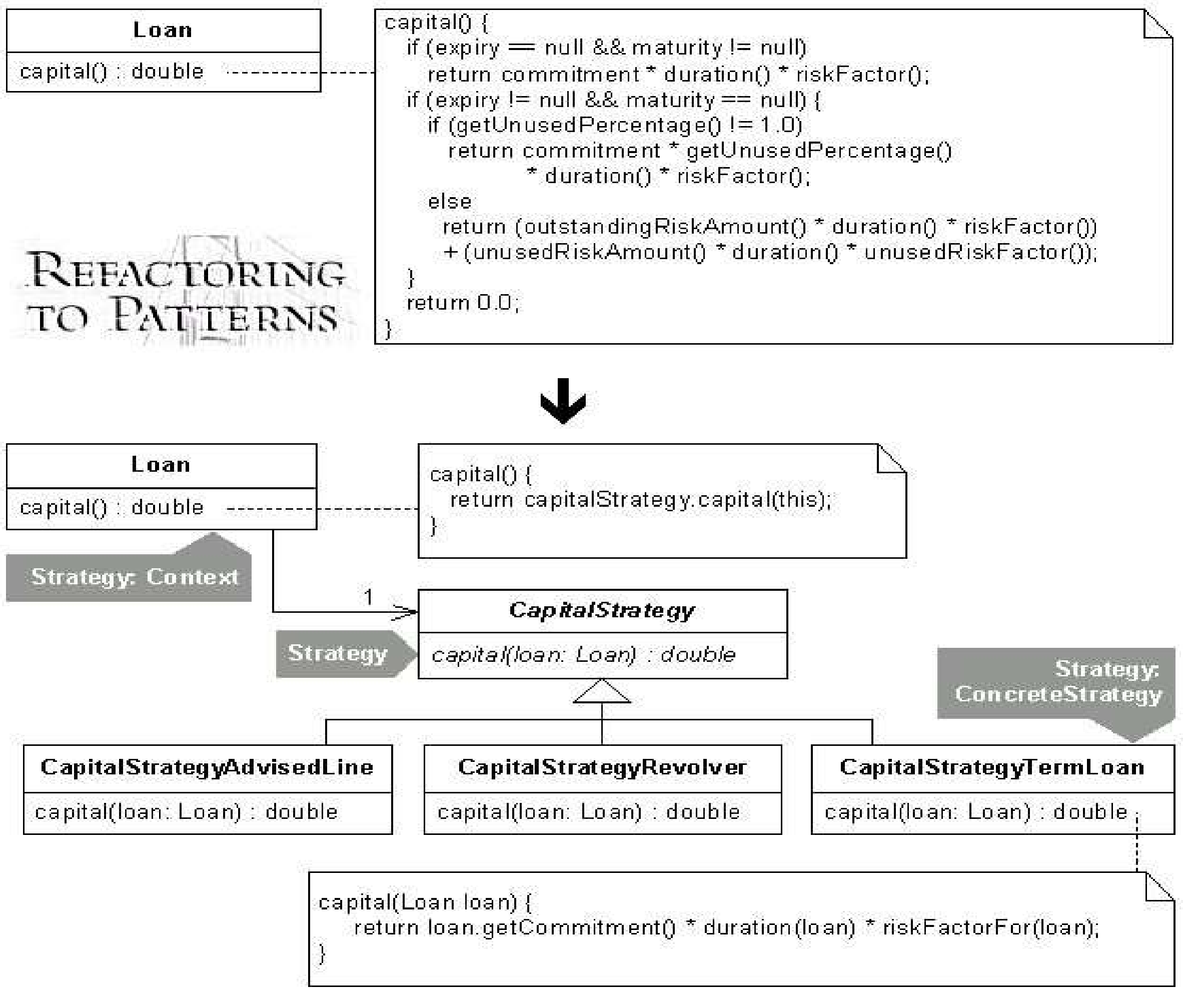
REFACTORING TO PATTERNS
CapitalStrategy
+capital(loan: Loan) : double
+duration(loan: Loan) : double
#riskFactorFor(loan: Loan) : double
CapitalStrategyAdvisedLine
+capital(loan: Loan) : double
CapitalStrategyTermLoan
+capital(loan: Loan) : double
+duration(loan: Loan) : double
return loan.getCommitment() * loan.getUnusedPercentage() * duration(loan) * riskFactorFor(loan);
return loan.getCommitment() * duration(loan) * riskFactorFor(loan);
Template Method: AbstractClass
Template Method
CapitalStrategy
+capital(loan: Loan) : double
+duration(loan: Loan) : double
#riskFactorFor(loan: Loan) : double
#riskAmountFor(loan: Loan) : double
return riskAmountFor(loan) * duration(loan) * riskFactorFor(loan);
Template Method: ConcreteClass
CapitalStrategyAdvisedLine
#riskAmountFor(loan: Loan) : double
CapitalStrategyTermLoan
+duration(loan: Loan): double
#riskAmountFor(loan: Loan) : double
return loan.getCommitment() * loan.getUnusedPercentage();
return loan.getCommitment() ;

Fourth Step: Refactoring. Refactorings 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.



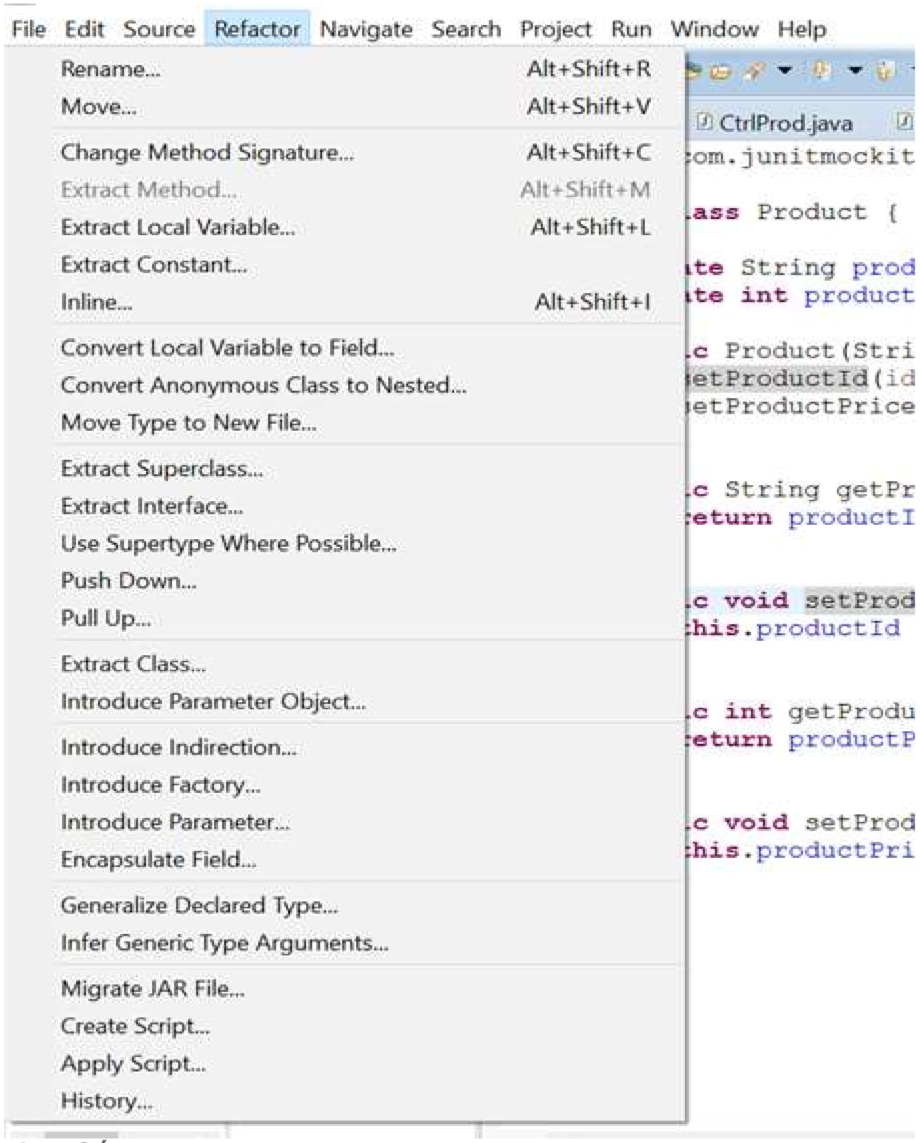
Loan
capital() : double
REFACTORING TO PATTERNS
Loan
capital() : double
Strategy: Context
1
CapitalStrategy
capital(loan: Loan) : double
Strategy
Strategy: ConcreteStrategy
CapitalStrategyAdvisedLine
capital(loan: Loan) : double
CapitalStrategyRevolver
capital(loan: Loan) : double
CapitalStrategyTermLoan
capital(loan: Loan) : double

```
capital() {
  if (expiry == null && maturity != null)
    return commitment * duration() * riskFactor();
  if (expiry != null && maturity == null) {
    if (getUnusedPercentage() != 1.0)
      return commitment * getUnusedPercentage()
              * duration() * riskFactor();
    else
      return (outstandingRiskAmount() * duration() * riskFactor())
      + (unusedRiskAmount() * duration() * unusedRiskFactor());
  }
  return 0.0;
}
```

```
capital() {
  return capitalStrategy.capital(this);
}
```

```
capital(Loan loan) {
    return loan.getCommitment() * duration(loan) * riskFactorFor(loan);
}
```

Fourth Step: Refactoring. Eclipse Refactorings


File Edit Source Refactor Navigate Search Project Run Window Help
CtrlProd.java

```
Rename...                                 Alt+Shift+R
Move...                                   Alt+Shift+V
Change Method Signature...                Alt+Shift+C
Extract Method...                         Alt+Shift+M
Extract Local Variable...                 Alt+Shift+L
Extract Constant...
Inline...                                 Alt+Shift+I
Convert Local Variable to Field...
Convert Anonymous Class to Nested...
Move Type to New File...
Extract Superclass...
Extract Interface...
Use Supertype Where Possible...
Push Down...
Pull Up...
Extract Class...
Introduce Parameter Object...
Introduce Indirection...
Introduce Factory...
Introduce Parameter...
Encapsulate Field...
Generalize Declared Type...
Infer Generic Type Arguments...
Migrate JAR File...
Create Script...
Apply Script...
History...
```

```
om.junitmockit
ass Product {
te String prod
te int product
c Product(Stri
etProductId(id
etProductPrice
c String getPr
eturn productI
c void setProd
his.productId
c int getProdu
eturn productP
c void setProd
his.productPri
```

References

- *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://www.youtube.com/watch?v=U4hIpntxWYc&index=3&list=PL25790B85E32D00B4*