

Software Design and Construction

159.251

Design Patterns, Antipatterns, Code Smells and Refactoring

Amjed Tahir

a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

[GangOf4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1st edition 1995.

[PJ] Mark Grand: Patterns in Java Volume 1. Wiley 1998.

[REFACT] Martin Fowlers: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional; 1st edition (June 28, 1999)

[PPPR] The Portland Pattern Repository:

<http://c2.com/ppr/index.html>

[REFACT.COM] refactoring.com - an online refactoring catalogue

Summary

- Design Patterns – Introduction
- Design Patterns in Context: BigBank Case Study
- Other Design Patterns
- Code-Level And Architectural Patterns
- Anti-patterns
- Code Smells
- Refactoring

Design Patterns

- Inspired by work on common patterns in architecture by Christopher Alexander.

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander].

- Made popular by the design pattern catalogue compiled by the **GangOf4** (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) in the 90ties.

There are five principles of class design (aka SOLID):

- (SRP) The SingleResponsibilityPrinciple
- (OCP) The OpenClosedPrinciple
- (LSP) The LiskovSubstitutionPrinciple
- (ISP) The InterfaceSegregationPrinciple
- (DIP) The DependencyInversionPrinciple

Design Patterns in Architecture



suspension bridge - reusable in different contexts, with different materials, on different scales

Pattern Languages

- a pattern is described using a **pattern language** – a structured template
- a template usually contains (at least) the following information:
 - description of a common problem
 - a description of the a solutions for this problem
 - a description how to move from the problem to the solution

The GangOf4 Design Pattern Language

The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

The GangOf4 D.P. Language (ctd)

The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

From E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley 1997.

Design Patterns BigBank Case Study

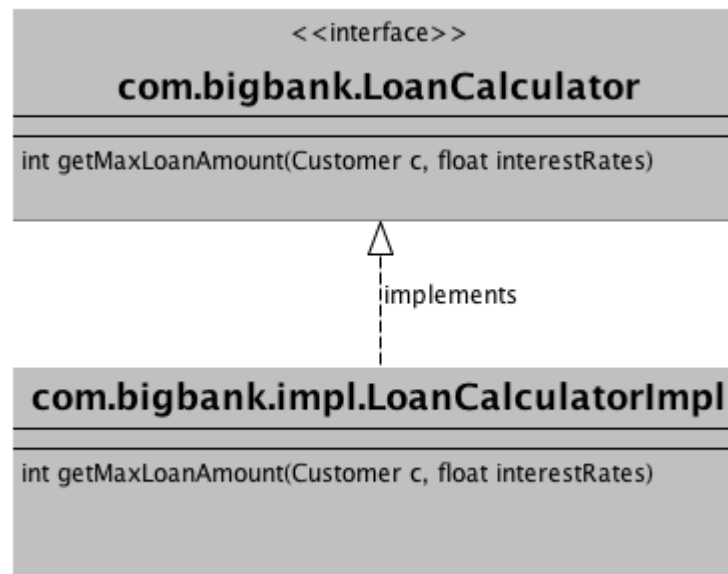
Assume that....

- you are working for a bank (bigbank.com) and want to calculate the maximum amount of money the bank can lend to a customer
- this is a sophisticated formula that uses multiple parameters such as: customer data, customer history, the current interest rates, the anticipated interest rates, and the securities a customer has
- lets start with an interface:

```
<<interface>>  
com.bigbank.LoanCalculator  
-----  
int getMaxLoanAmount(Customer c, float interestRates)
```

BigBank Case Study (ctd)

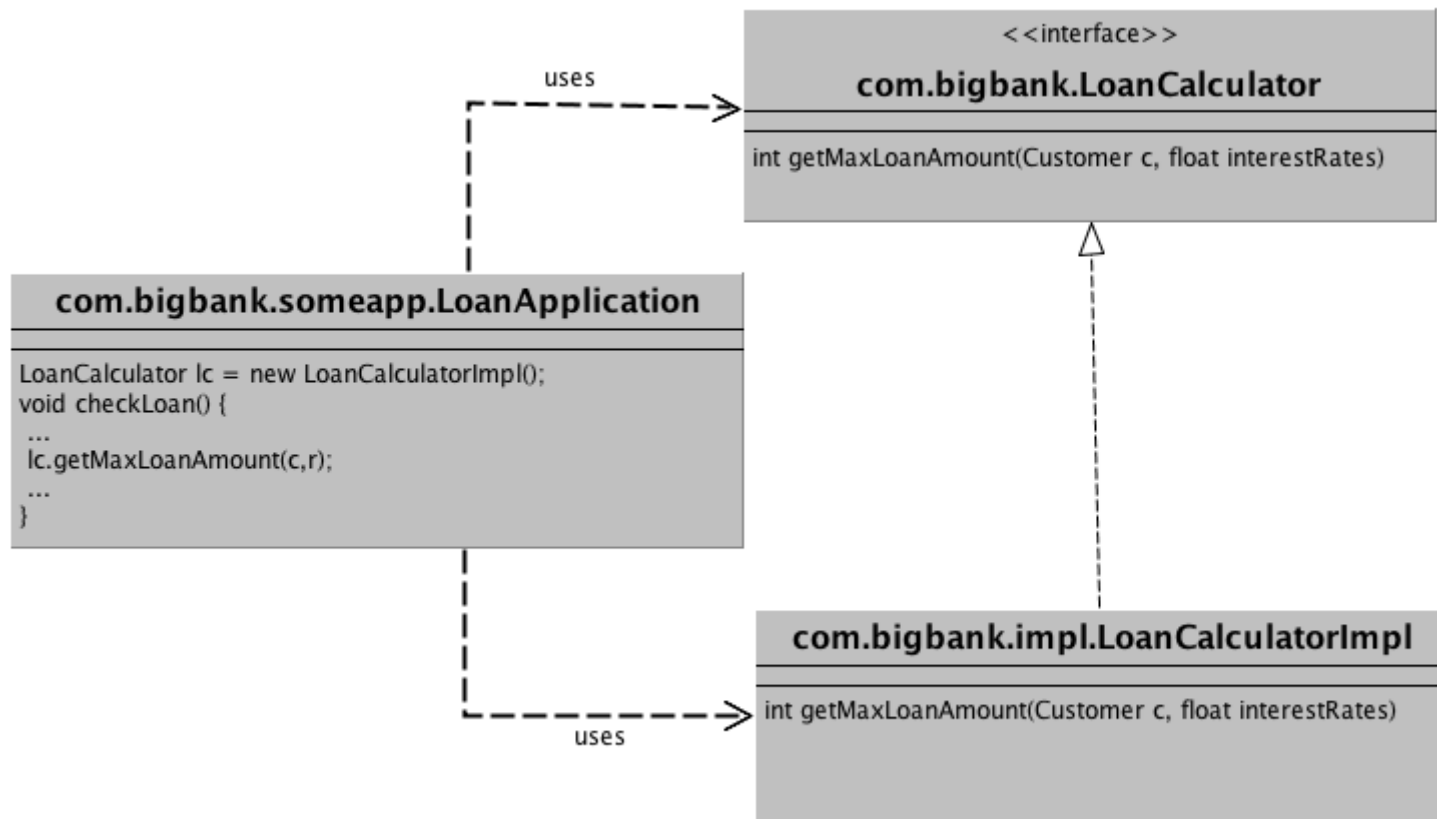
the next step is to implement a loan calculator:



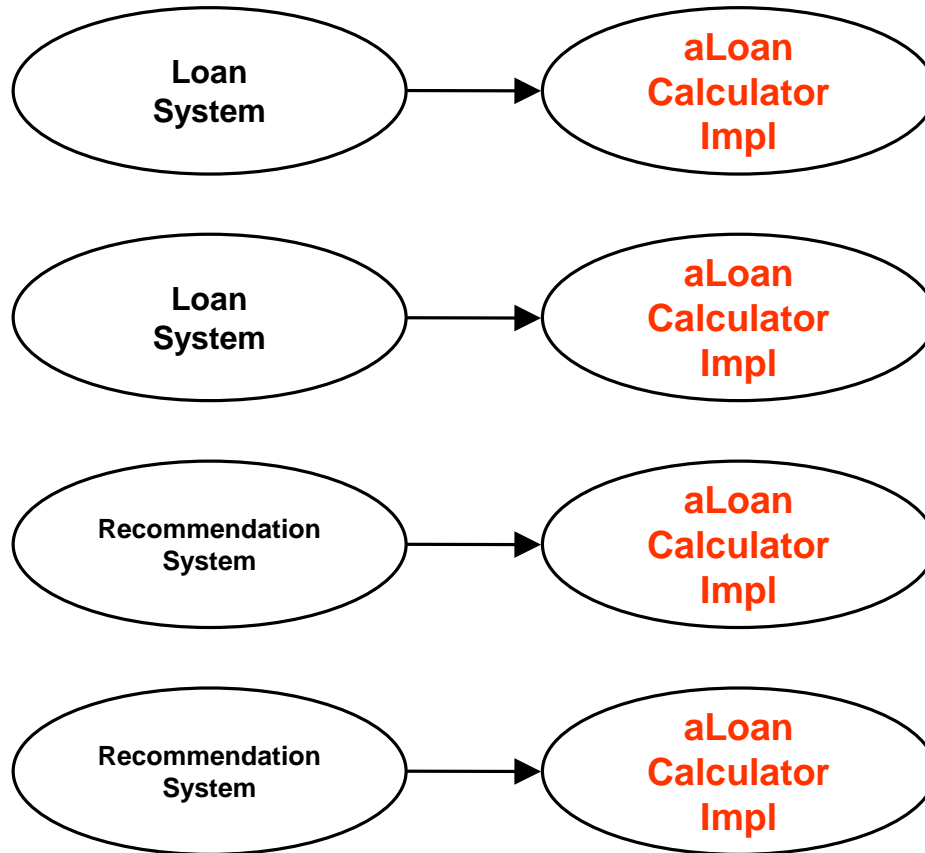
BigBank Case Study (ctd)

- the calculator is used by many different users / applications at the same time
- this may lead to a large number of instances of **LoanCalculatorImpl** being created
- this is particularly bad for application servers - one instantiation happens per (web/network) request
- Initiating **LoanCalculatorImpl** might be expensive: it could include connecting to databases and external services, and connecting includes expensive authentication and authorisation checks

BigBank Case Study (ctd)



BigBank Case Study (ctd)



BigBank Case Study (ctd)

- stateless objects are replicated unnecessarily: each (web session) in each application creates a new object
- this increases memory consumption, and the unnecessary creation and garbage collection of objects may slow down the application

BigBank Case Study – *Singleton Pattern*

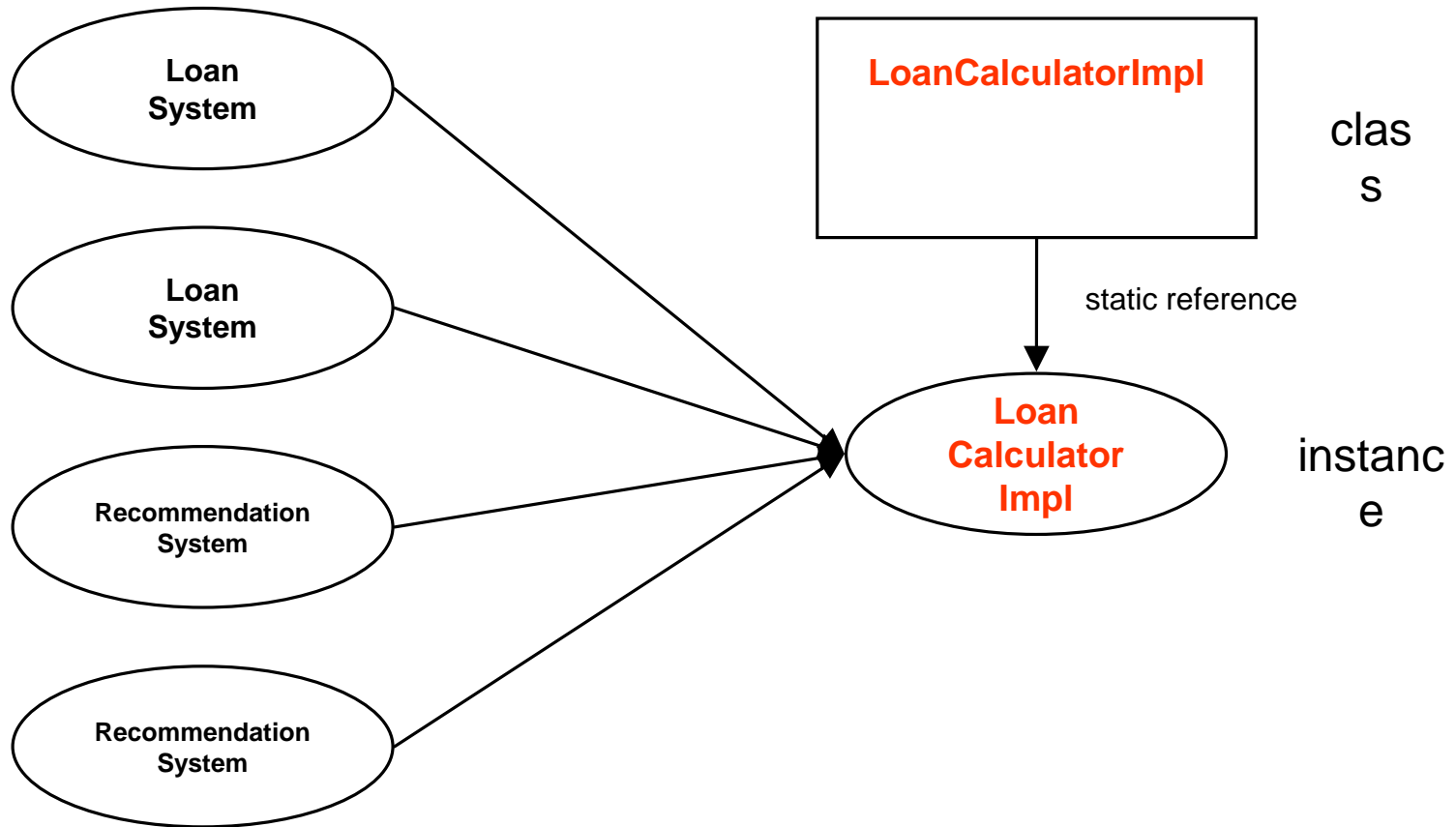
- **Idea:** make sure that there is only one instance of LoanCalculatorImpl
- **solution:**
 - provide one instance using a static variable in the class (define a public static operation (getInstance()) that returns the sole instance of the class.)
 - make constructor *private* so that nobody else can instantiate class (i.e., hide the constructor of the class)
- applications can get a reference to the sole instance as follows: **LoanCalculatorImpl.defaultInstance**

BigBank Case Study - Singleton (ctd)

```
public class LoanCalculatorImpl implements LoadCalculator{  
    public static LoanCalculatorImpl defaultInstance =  
        new LoanCalculatorImpl();  
  
    // private constructor  
    private LoanCalculatorImpl () {  
        super();  
    }  
    @Override  
    int getMaxLoanAmount(Customer c,float interestRates) {  
        ...  
    }  
}
```

this instantiation is still possible
as the now private constructor
is accessible from within the
class!

BigBank Case Study - Singleton (ctd)



BigBank Case Study - Singleton (ctd)

- it controls the number of instances (restricts it to either exactly or at most one) by using a **static variable**
- variant: make static field private, and access it through public access method - often used with **lazy initialisation** of the field
- the singleton pattern has the following benefits:
 - optimise the use of resources
 - have a single point of control (e.g., exchange the sole instance at runtime by an instance of a subclass)
 - make sure that access to certain resources is serialised

BigBank Case Study (ctd)

Assume the following:

- the calculation algorithm implemented turns out to be fragile, and the bank does not have the know how to implement a really good algorithm
- fortunately, there is a company called Mafia (**Market Finance Applications**) specialising in this sort of applications
- they use state of the art algorithms which are also compliant with government regulations on risk management in the financial industry (such as [Basel II](#) or [Sarbanes-Oxley](#))

BigBank Case Study (ctd)

- Mafia has a class **MafiaAlgorithms** with the following API:

```
int maxPossibleLoanAmount(String customerID,String rates);
```

- obviously, this class does implement the (bigbank) **LoanCalculator** interface, but this problem can be easily fixed as the respective methods are **semantically similar** - they provide the same service through a different interface

BigBank Case Study - Adapter

```
package com.bigbank.mafia.*;
import com.mafia.MafiaAlgorithms ;
class LoanCalculator implements com.bigbank.LoanCalculator {
    private MafiaAlgorithms delegate = new MafiaAlgorithms ();
    @Override
    public int getMaxLoanAmount(Customer c,float interestRates) {
        String id = c.getId();
        String rates = String.valueOf (interestRates);
        return delegate.maxPossibleLoanAmount(id, rates);
    }
}
```

BigBank Case Study - Adapter (ctd)

- instances of `com.bigbank.mafia.LoanCalculator` **wrap** instances of `MafiaAlgorithms` so that they fit into the overall design (and the interface can be implemented)
- `com.bigbank.mafia.LoanCalculator` **delegates** most of its functionality, it only does some data type conversion
- it is an **adapter** between bigbank and mafia classes

BigBank Case Study - Adapter (ctd)

the interface: shape of pins, current, ..



a "**regular**" implementation of the interface

an object we want to use, but its **interface does not match** - this is the **adaptee**. The interface is different, but it has a **similar semantics** (current, frequency,..)



solution: use an **adapter** to fit the interface!

BigBank Case Study - Adapter (ctd)

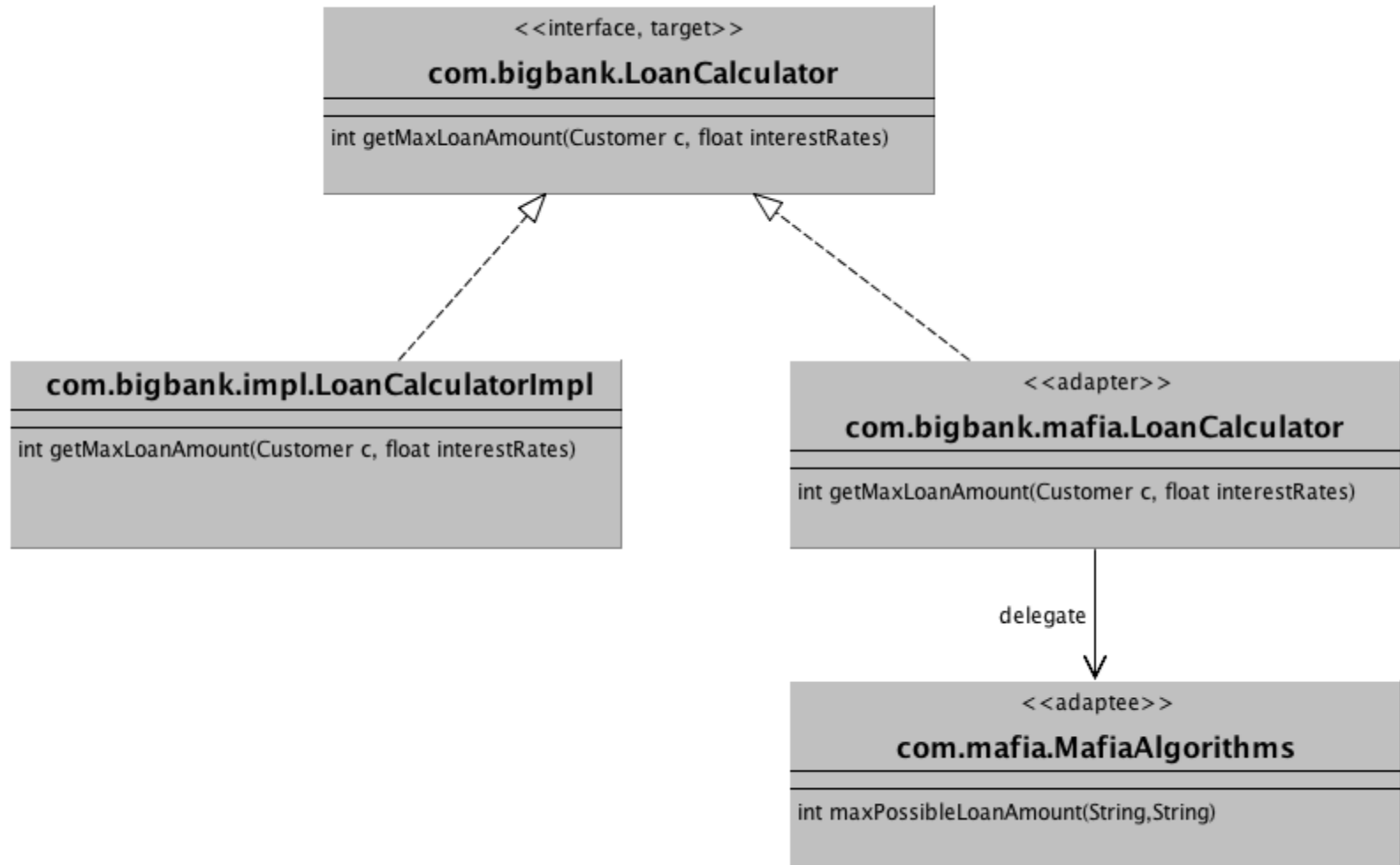
- **Problem:**

the **adaptee** has a service (implemented as a method) which we want to use to provide another service (a method in a **target** class)

- **Solution:**

we write an **adapter** that subclasses / implements the target but uses also references the adaptee and can use the services of the adaptee to provide the target service

BigBank Case Study - Adapter (ctd)



BigBank Case Study (ctd)

- the migration of all applications to use the new mafia calculator turns out to be difficult
- the reason is that many deployed applications have this line of code hardcoded into them:

LoanCalculatorImpl.defaultInstance

- this is an explicit reference to the old implementation class, and every occurrence must be replaced

BigBank Case Study (ctd)

- to avoid this situation in the future, the bank decided to introduce a special class that has only one purpose: to create instances of **LoanCalculator**
- in the future, whenever the implementation class has to be changed again (e.g., in case the bank misunderstood the definition of Mafia), only one class has to be updated
- such a class is called a **factory** class
- since the factory is stateless by nature, it can be combined with singleton so that there is only one factory instance

BigBank Case Study - Factory (ctd)

```
package com.bigbank.*;
class LoanCalculatorFactory {

    LoanCalculator getCalculator() {
        return com.bigbank.mafia.LoanCalculator.defaultInstance;
    }
}
```

- this is an instance of the **Factory** design pattern
- a factory encapsulates the process of instance creation
- it **decouples** client applications from service providers

BigBank Case Study - Factory (ctd)

even if we have centralised instantiation of a service, there are still problems

problem 1: deployment

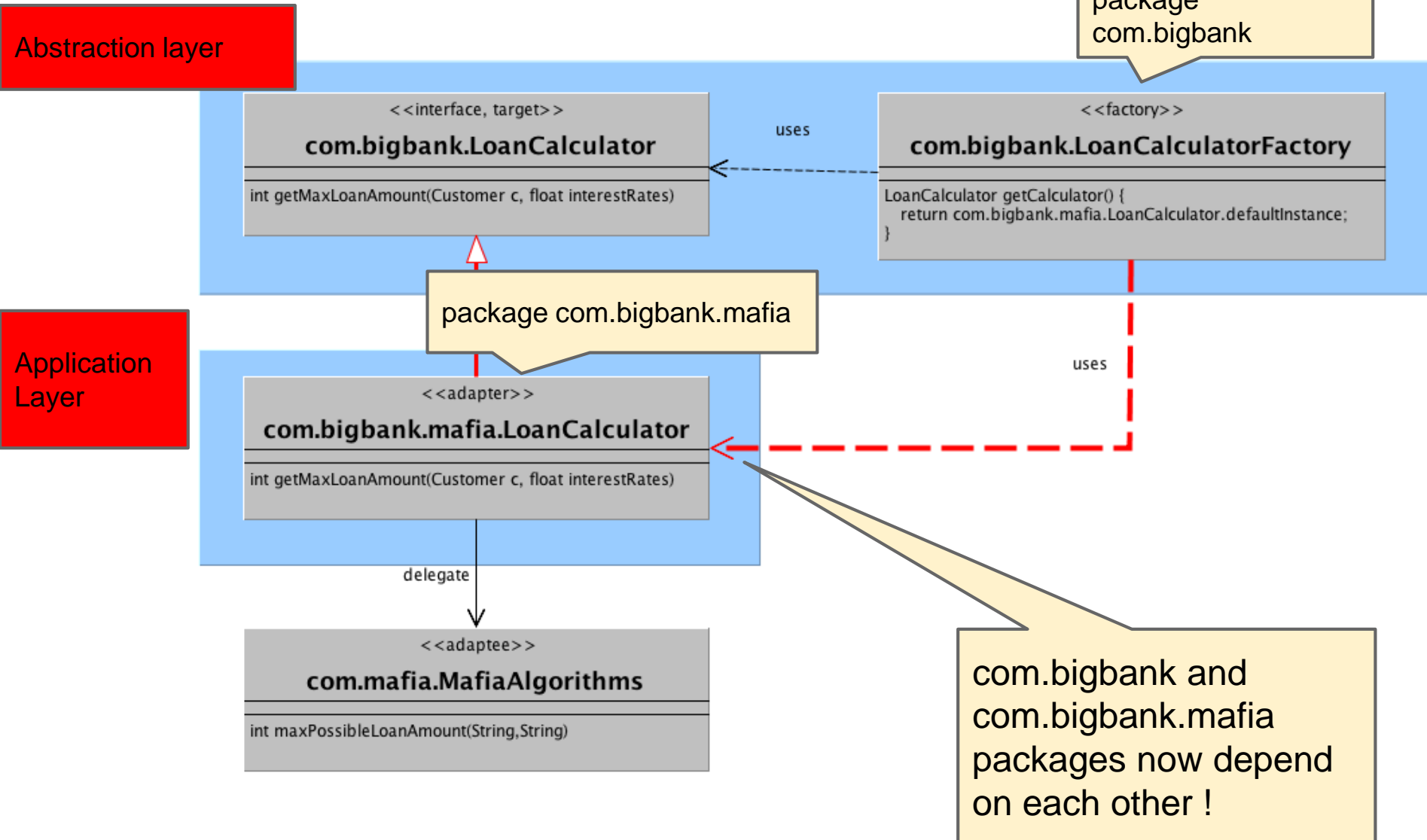
- the factory class is deployed multiple times (multiple copies of the jar used in different applications)
- when the factory changes, they all must be updated
- Java has an elegant solution for this problem: network class loaders can load libraries over the network (from one central deployment server)
- several higher level technologies are based on this idea, incl. Applets, WebStart and [OSGi](#)

BigBank Case Study - Factory (ctd)

problem 2: package dependencies

- the implementation class **com.bigbank.mafia.LoanCalculator** is referenced from a class in the package **com.bigbank**
- i.e., the package **com.bigbank** depends on **com.bigbank.mafia**, and vice versa !
- this violates two core OO principle:
 - the [acyclic dependency principle](#) (no circular dependencies between packages):
 - the [dependency inversion principle](#) (abstract/high level modules should not depend on details/low level modules)

BigBank Case Study - Factory (ctd)



Solution 1: AbstractFactory

To Read-
not in exam

- these problems can be fixed using the **AbstractFactory** pattern
- it consists of abstract/concrete pairs of factories and products
- In the **com.bigbank** package, create an abstract factory class, it has the abstract *factory* method
- In the **com.bigbank.mafia** package, create the *concrete* factory
- methods to install the concrete factory: when the concrete factory is loaded (using a static initialization block)

Solution 2: Service Locators

To Read-
not in exam

- the **service locator pattern** uses a locator to **lookup services** (often identified by name)
- This is comprehensively used J2EE to look up resources like database connections

Solution 2: Service Locators

To Read-
not in exam

- Java has the **service loader** utility (`java.util.ServiceLoader`) that can be used :
 - the **interface names** are used as **service names**
 - Uses **dependency injection**
 - services are listed in text files in `META-INF/services` with names matching the interface names
 - lines are names of classes implementing this interface ("services")
 - the service loader provides an iterator for all services available for a given type, instantiation is by *reflection*

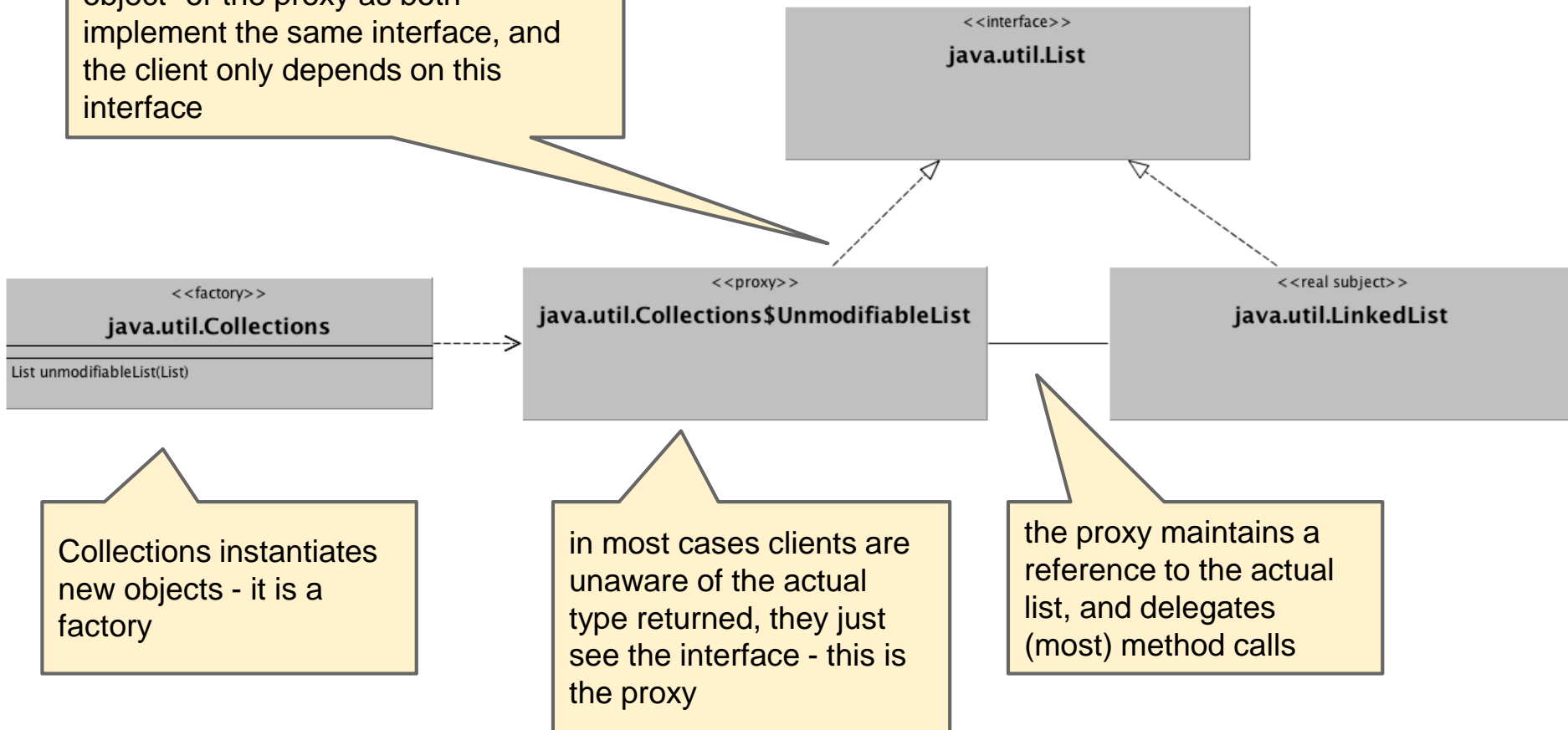
Design Patterns Used in the JDK

```
java.util.Collections.unmodifiableList(java.util.List)
```

- this method produces an unmodifiable **view** on a list
- it is a factory, the actual implementation class used is hidden
- the returned view acts like an adapter for the underlying list, however, the target and the hidden interfaces are the same
- this is another (similar) design pattern called **proxy pattern**
- proxies are widely used in networking and persistency frameworks (e.g., [CORBA](#))

Proxy and Factory in Collections

for a client application there is little difference whether this is the "real object" or the proxy as both implement the same interface, and the client only depends on this interface

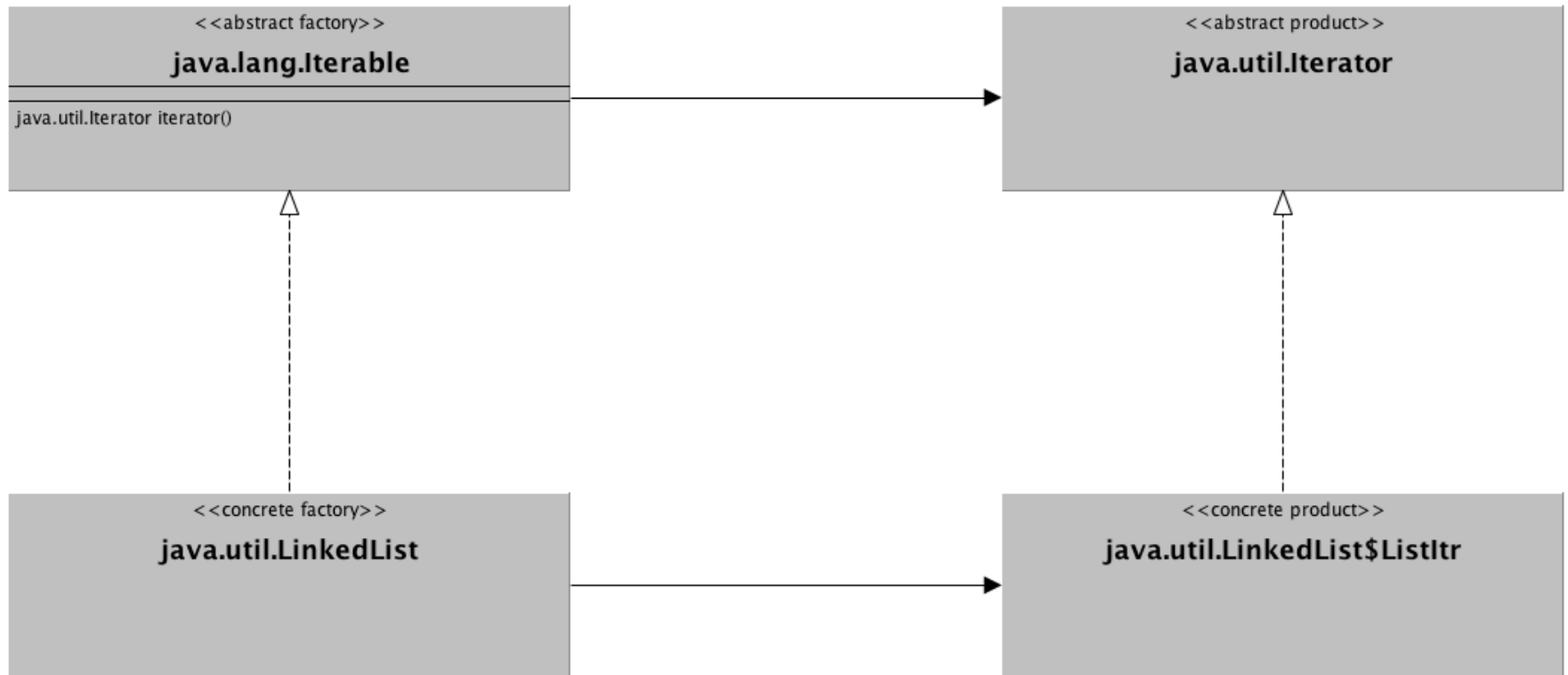


Design Patterns Used in JDK (ctd)

```
java.util.Collection.iterator()
```

- this method is used to support compact for-loops, it returns an Iterator
- the actual interface that defines this method is `java.lang.Iterable`
- this is an AbstractFactory Patterns
- `java.lang.Iterable` is the abstract factory
- concrete subclasses like `java.util.LinkedList` are the concrete factories
- `java.util.Iterator` is the abstract product
- the concrete products are encapsulated, these are classes like `java.util.LinkedList$ListItr`

AbstractFactory in Iterable



Other Important Patterns

- **observer** - used to notify other objects but avoiding strong dependencies - used in user interface event notification and parsing
- **composite** - organise hierarchies of objects - used in user interface widget hierarchies and (DOM style) parsing
- **visitor** - inject functionality into data structures, overcome limitations of single dispatch - used in parsing
- **model-view-controller (MVC)** - organise user interfaces by separating data from views
- *more coverage in 300/400 level papers, for case study examples see: <https://bitbucket.org/jensdietrich/oop-examples>*

Types of Patterns

- the patterns discussed so far are design patterns mainly concerned with design
- there are other types of patterns:
- **code-level patterns**,
 - such as lazy initialisation
- **architectural patterns**,
 - such as layered architecture
- **organisational patterns**
 - Concerned with management and documentations

Lazy Initialisation

- example of a code level pattern
- a class has a field (instance variable) *f* that is expensive to initialise
- solution: initialise it only when it is needed (=accessed for the first time)
- takes advantage of getter encapsulation
 - all access is through a getter, and can be controlled through the getter
- opposite strategy: eager initialisation

Lazy Initialisation (ctd)

eager initialisation

```
class Foo {  
    private T f = initialiseF ();  
    // this is an expensive operation!  
    private initialiseF() {...}  
    public T getF() {  
        return f;  
    }  
}
```

Lazy Initialisation (ctd)

lazy initialisation

```
class Foo {  
    private T f = null;  
    // this is an expensive operation!  
    private initialiseF() {...}  
    public T getF() {  
        if (f==null) {  
            f = initialiseF ();  
        }  
        return f;  
    }  
}
```

first access triggers
initialisation

note that using **null** here does not always
work - if `initialiseF` can return null then
another object must be used to represent
"uninitialised"

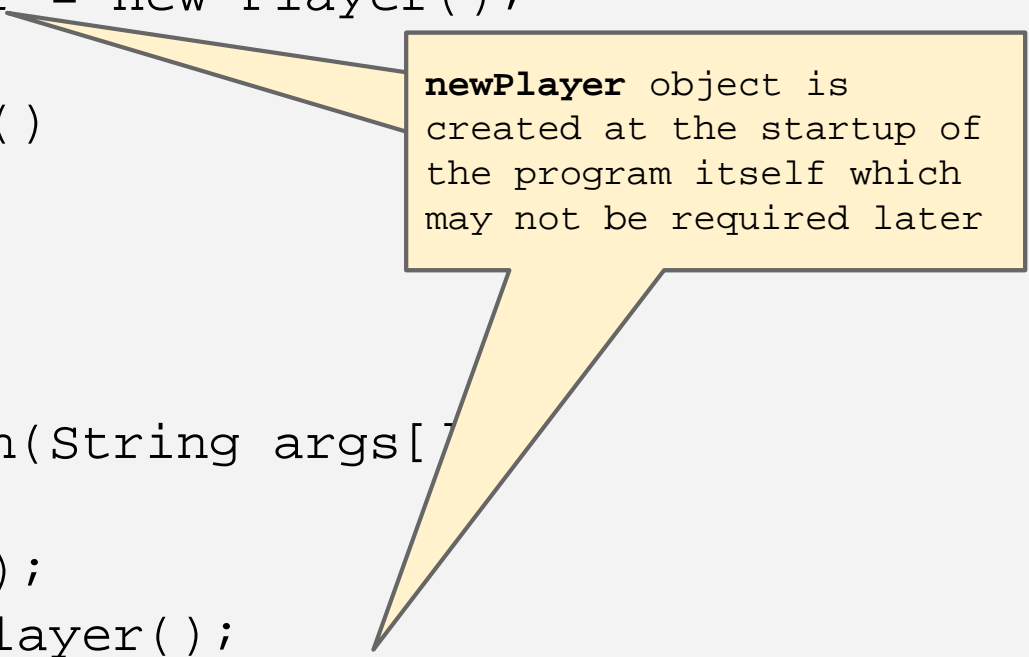
Eager vs lazy Initialisation

```
public class Demo
{

private Player newPlayer = new Player();

public Player getPlayer()
{
    return newPlayer;
}

public static void main(String args[])
{
    Demo d1 = new Demo();
    Player p1 = d1.getPlayer();
}
}
```



newPlayer object is created at the startup of the program itself which may not be required later

Eager vs lazy Initialisation (ctd)

```
public class Demo
{
    private Player newPlayer;

    public Player getPlayer()
        if(newPlayer == null){
            newPlayer = new Player();
        }
        return newPlayer;
}

public static void main(String args[])
{
    Demo d1 = new Demo();
    Player p1 = d1.getPlayer();
}
```

newPlayer object is created only when required!

Lazy Initialisation (ctd)

issues:

- null represents the “not yet initialised state” - this will not work if null is a possible initialised state (i.e., if initializeF() can return null)
- in this case a special instance of the field type (for example, I) has to be used to represent this state
- in multithreaded programs, lazy initialisation must be synchronized to guarantee that it is done only once

Antipatterns

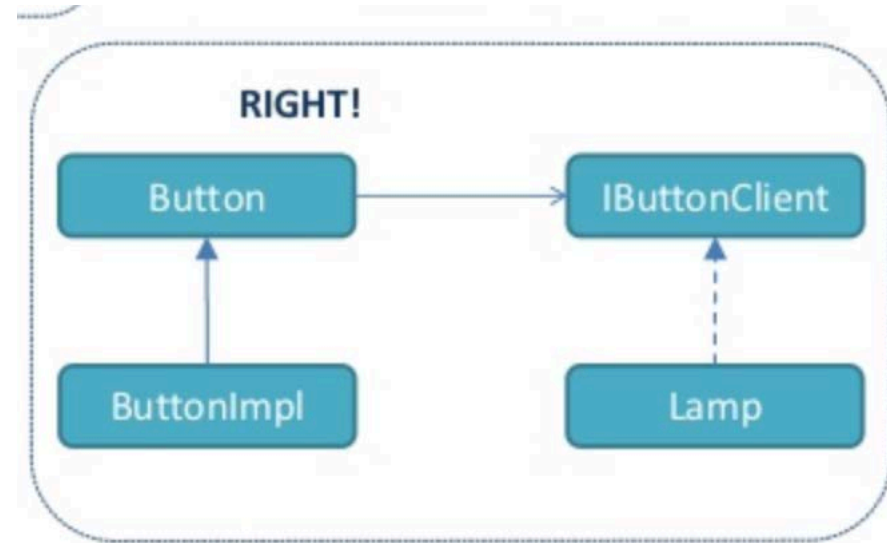
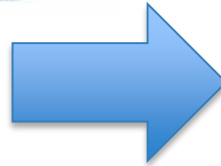
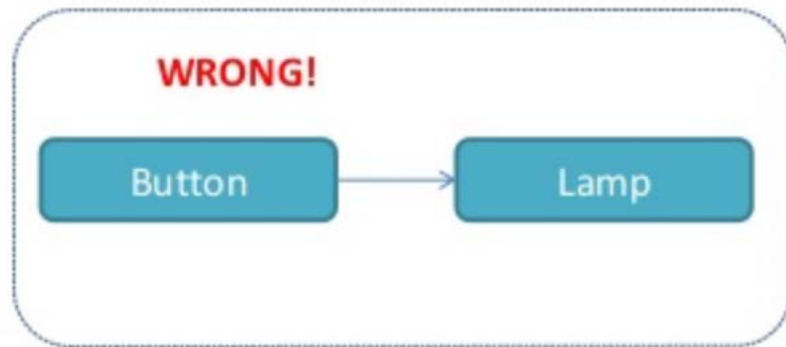
- antipatterns are patterns that are unproductive and/or error-prone and have bad consequences
- antipatterns have been investigated on several levels
- code-level antipatterns can often easily be formalised as queries on the AST of a program, and good tool support exists to find them
- code-level antipattern examples: unused variables, conditionals not followed explicitly defined blocks ({})

- A comprehensive list of anti-patterns is available here:
<http://wiki.c2.com/?AntiPattern>
- tooling:
 - code-level: PMD, FindBugs
 - design-level: JDepend, Lattix, [Massey Architecture Explorer](#)

Subtype Knowledge (STK)

- recap: [dependency inversion principle](#) (DIP):
 - High level modules should not depend upon low level modules. Both should depend upon abstractions.
 - Abstractions should not depend upon details. Details should depend upon abstractions.
- subtype knowledge antipattern: a supertype uses its own subtype
- as supertypes are more abstract, this violates DIP !

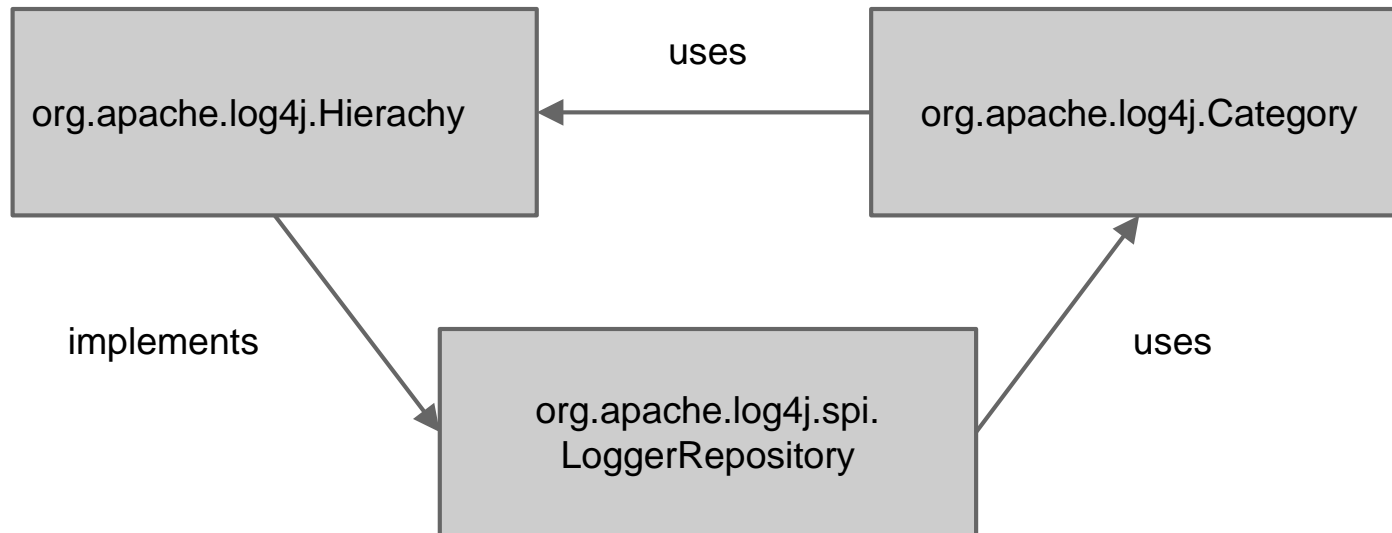
DIP example



<https://www.slideshare.net/paskavatta/mo-dip-15104789>

Subtype Knowledge (STK) (ctd)

- STK in log4j (<http://tinyurl.com/y7peevzq>)
- note that the classes are in different packages - circular dependency between packages is another antipattern!



Degenerated Inheritance (DEGINH)

- classes have multiple inheritance paths to supertypes
- in Java, the supertypes must be interfaces (due to single inheritance restriction for classes)
- this violates DRY, makes code more difficult to maintain
- aka diamond or fork-join inheritance

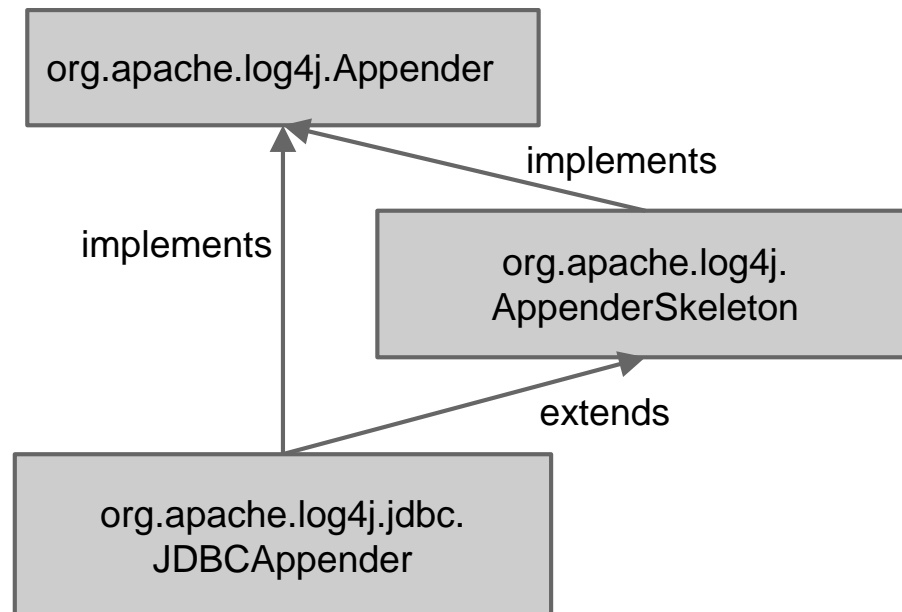
Degenerated Inheritance (DEGINH) (ctd)

- DEGINH in log4j 1.2.15 (<http://tinyurl.com/y9lexlpd>)
- caused by:

```
public class JDBCAppender
```

```
extends org.apache.log4j.AppenderSkeleton
```

```
implements org.apache.log4j.Appender
```



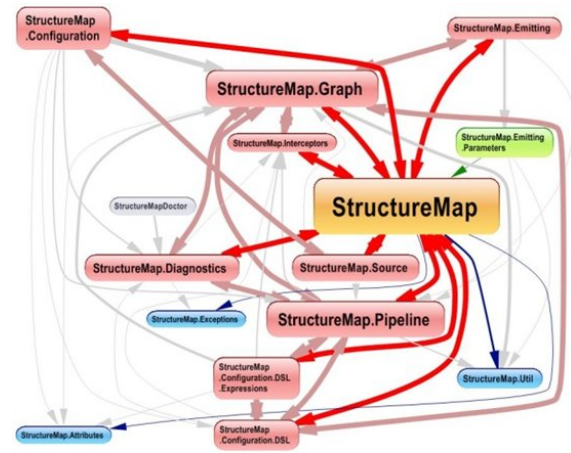
Other known anti-patterns

- Anti-pattern catalog:
<http://wiki.c2.com/?AntiPatternsCatalog>
- All anti-patterns have funny names:
 - [Swiss Army Knife](#)
 - excessively complex class interface. The designer attempts to provide for all possible uses of the class.



- Spaghetti Code

source code that has a complex and tangled control structure, especially one using many GOTO statements, exceptions, threads, or other "unstructured" branching constructs.



```

1 C      A weird program for calculating Pi written in Fortran.
2 C      From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.
3
4      PROGRAM PI
5      DIMENSION TERM(100)
6      N=1
7      3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
8      N=N+1
9      IF (N-101) 3,6,6
10     6 N=1
11     7 SUM98 = SUM98+TERM(N)
12     WRITE(*,28) N, TERM(N)
13     N=N+1
14     IF (N-99) 7, 11, 11
15     11 SUM99=SUM98+TERM(N)
16     SUM100=SUM99+TERM(N+1)
17     IF (SUM98-3.141592) 14,23,23
18     14 IF (SUM99-3.141592) 23,23,15
19     15 IF (SUM100-3.141592) 16,23,23
20     16 AV89=(SUM98+SUM99)/2.
21     AV90=(SUM99+SUM100)/2.
22     COMANS=(AV89+AV90)/2.
23     IF (COMANS-3.1415920) 21,19,19
24     19 IF (COMANS-3.1415930) 20,21,21
25     20 WRITE(*,26)
26     GO TO 22
27     21 WRITE(*,27) COMANS
28     22 STOP
29     23 WRITE(*,25)
30     GO TO 22
31     25 FORMAT('ERROR IN MAGNITUDE OF SUM')
32     26 FORMAT('PROBLEM SOLVED')
33     27 FORMAT('PROBLEM UNSOLVED', F14.6)
34     28 FORMAT(I3, F14.6)
35     END
36

```


- Killing 2 Birds with 1 Stone

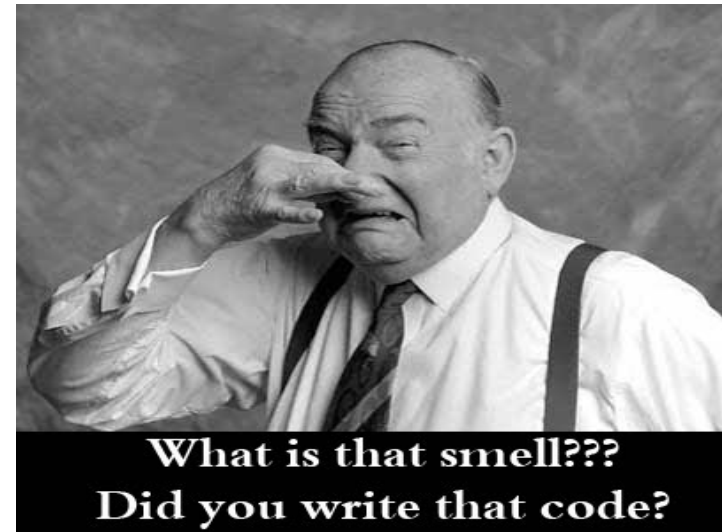
- Too many SwissArmyKnife components with decision on how to manage them!

Patterns and Anti-patterns

- Pattern usage depends on the scenario
- Don't use the pattern if it is not helpful in your case!
- Sometimes, an anti-pattern is the only way to do things, and therefore should not be considered bad.
- All depends on the scenario and the case needed!
- Always check before you decide!
- Patterns can turn anti-patterns if they are not correctly used, they have been implemented in the wrong way, or sometimes if they have been overused!

Code Smells

- smells indicate problematic code and design
- called “bad code”!
- smells are the **starting point** of refactorings – refactorings are about getting rid of bad smells in code
- similar/overlap with antipatterns, often a bit more vague (e.g., based on statistical observations)



Code Smells

(ctd)

- Code smells
 - segments of the source code that display potential design and implementation issues.
- In 1999, Martin Fowler provided an explanation of 22 different smells.
- The list has since extended to include far more smells
 - (there are over 70 code bad smells that have been identified in the literature~).
- Code smells cataloge
 - <https://sourcemaking.com/refactoring/smells>
 - <http://wiki.c2.com/?CodeSmell>

Code Smells

(ctd)

- it indicate code structures that can lead to difficulties during software evolution and maintenance.
- smells can arise as a result of poor design decisions (anti-patterns) – they are highly related
- Code smells usually refer to issues with the production code – i.e., the actual application source code.

Example of code smells

- Brain/God Class

too complex class that takes too much responsibilities.

- Feature Envy

a method making too many invocations to methods of another class.

- Type-Checking

a complicated conditional statement within a class .

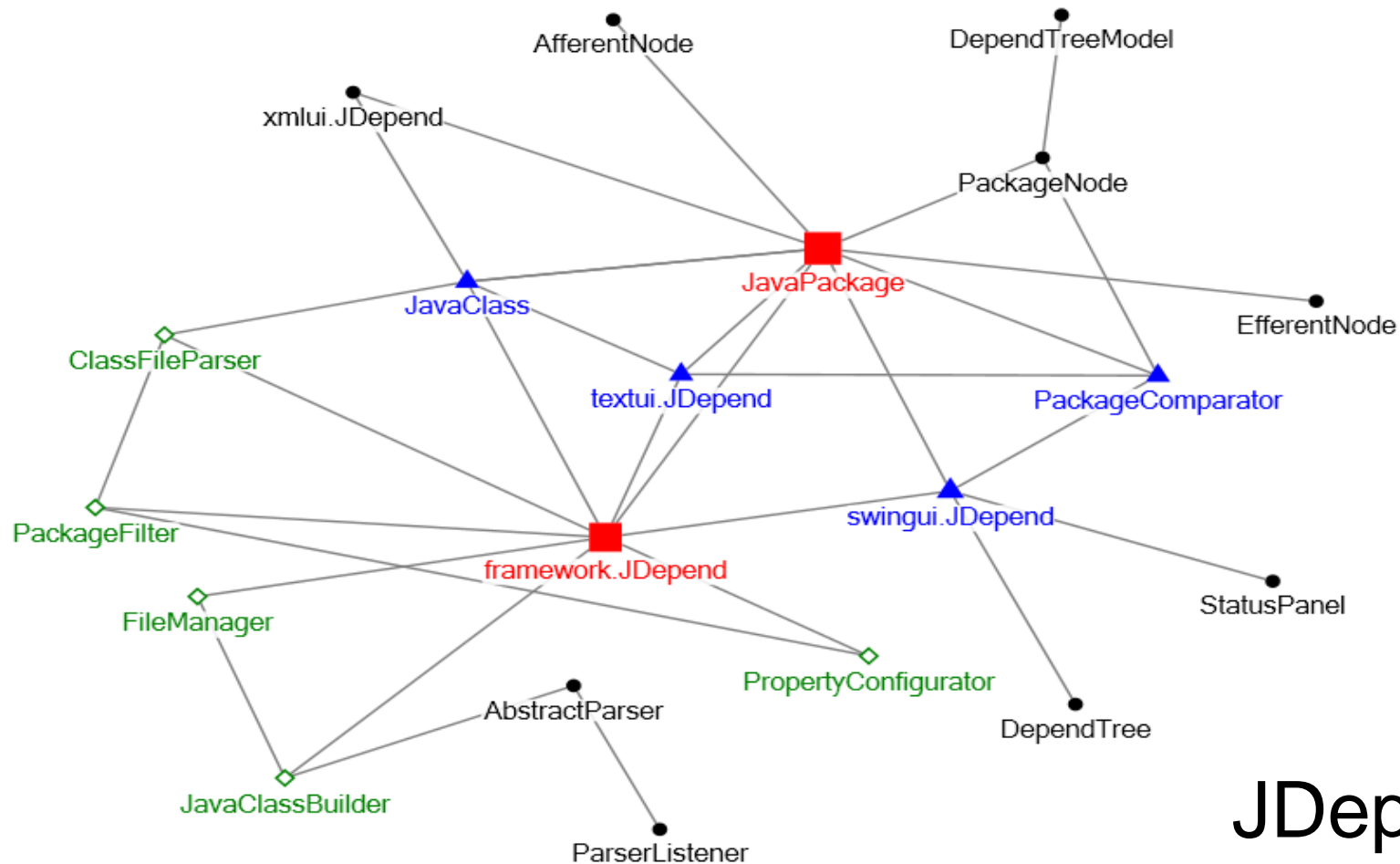
- Code Clones (duplications)

fragments of source code that appear in more than one place in the program.

Smells – other examples

- Long methods and Long classes
- Primitive obsession
 - using a lot of primitives and being afraid of classes -
 - encode complex data as strings
- Shotgun surgery
 - changes require many little changes in many
 - different places - indicates violations of DRY

God Class



JDepend
Runtime analysis

Example of smells

duplication in Code

```
public class Student {  
    public Date dob = null;  
    public String name = null;  
    public String firstName = null;  
    public Address address = null;  
    public Course course = null;  
    public int studentId = null;  
    public String getFullName()  
        return firstName +  
}  
}
```

```
public class Staff {  
    public Date dob = null;  
    public String name = null;  
    public String firstName = null;  
    public Address address = null;  
    public Department department = null;  
    public int staffId = null;  
    public String getFullName() {  
        return firstName + "  
    }  
}
```

duplicated state

duplicated behaviour

Type-Checking example

```
switch(something)
  case 'A' :
    create object 1
    break;
  case 'B' :
    create object 2
  case 'C' :
    create object 3
    break;
  case 'D' :
    create object 4
  case 'F' :

  default :
    create object 5
```

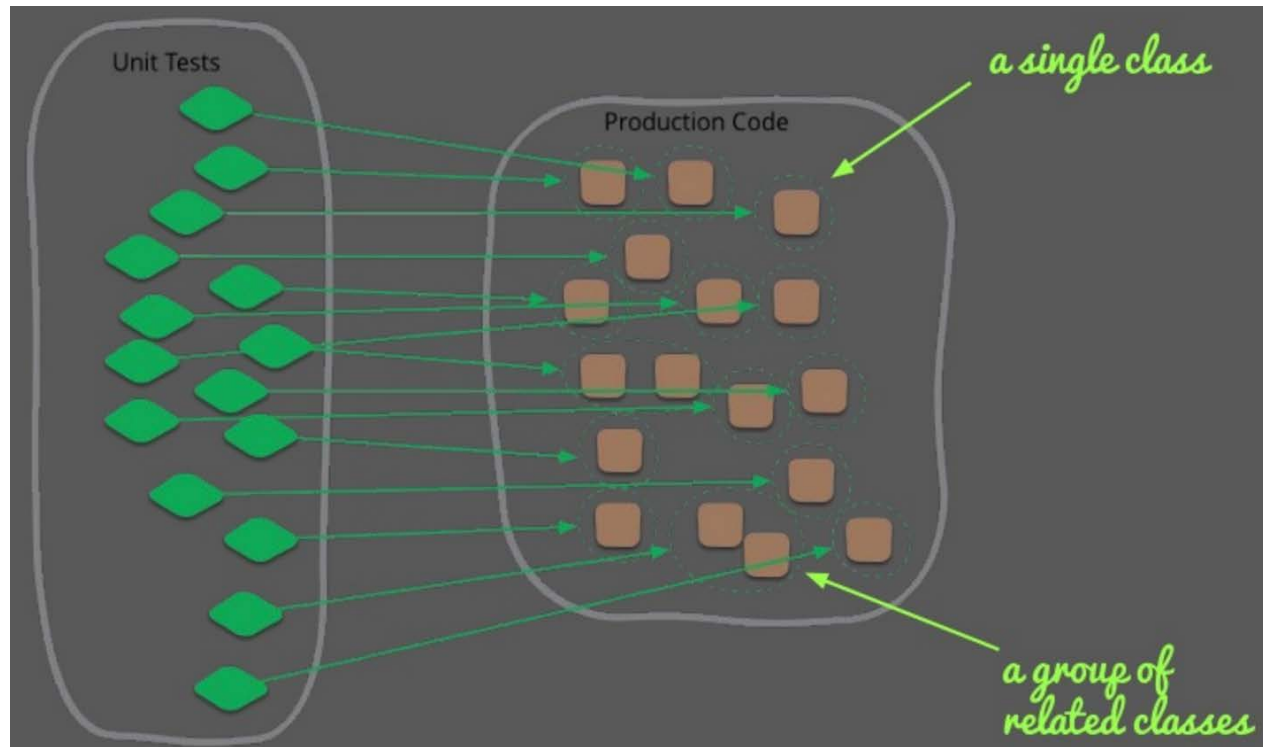
Test smells!

- Test code can also have bad smells
- We call this category of smells as “test smells”
- The term ***test smells*** was first defined by van Deursen *et al.* in 2001.

Van Deursen, Arie, et al. "Refactoring test code." Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001). 2001. APA

- “***test smells***” has been used to refer specifically to the group of code smells that affect only test code.

Class vs unit tests



Example of test smells:: Assertion Roulette

a test case incorporates too many assertions (mostly without description).

Lots of assertions -
but no descriptions!



```
public void testSomething() {  
    assertEquals("something", getSomething One());  
  
    assertTrue("something", getSomething Two());  
  
    assertEquals("something", getSomething Three());  
  
    assertEquals("something", blah_blah_blah());  
}
```

Example of test smells:: Eager Test

a unit test has at least one method that uses more than one method of the tested class.

```
public void testSomething() {  
    String filename = TEST_FILENAME;  
  
    assertTrue(filename + " missing", new File(filename).exists());  
  
    Some code .....  
  
    assertEquals("Number of files", 1, scanner.getNbFiles());  
    assertEquals("Number of classes", 1, scanner.getNbClasses());  
}
```

Example of test smells:: Sensitive Equality

the *toString* method is used in assert statements

```
public void testSomething() {  
    assertEquals("something", getSomething ().toString())  
}
```

Example of test smells:: Assertion-free:

. (probably the most annoying test smell!!!)

- A test case without real tests!!

```
public void testSomething() {
```

```
Lots of code .....
```

```
And code...
```

```
Some more code.....
```

```
//and no test at the end.... (assertion is missing!! )
```

```
}
```


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 ”
[REFACT]
- In issue tracker, attributes that need to be refactored can be marked as refactor, or enhance
- external behaviour = functionality
program should remain correct

Retaining External Behaviour

- this usually means the following:
 - refactored code can be compiled, if APIs have been changed, the refactoring includes changing the calls to the (new) API
 - the test cases succeed after the refactoring has been done
- note that test cases might have been part of the refactoring (e.g. in order to adapt them to a new API)!

Retaining External Behaviour (ctd)

- classical refactoring: change name of an artefact (class, package, method, ..)
- a good refactoring tool (like Eclipse) would ensure that all references using the old name are updated as well
- for references in code, this can be done by reasoning about a model of code (such as the AST)
- this is more difficult for references to code in other files (e.g., configuration files like the web.xml file used to configure Java web applications)

Refactoring operation types

- Refactoring can be done manually or automatically
- Manual refactoring: common, manually change the code
- Auto: use build tools to refactor
- Modern IDEs have refactoring functionalities.

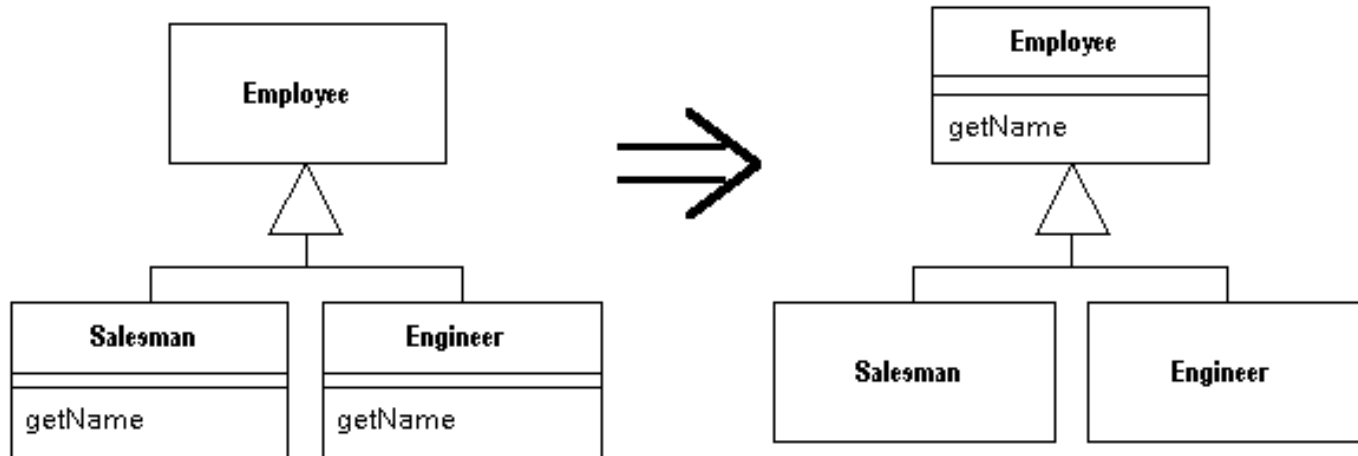
Refactoring Catalogues and Browsers

- refactorings are [cataloged](#) – different refactorings in the catalogue address different smells\anti-patterns
- tools that support the application of refactorings are sometimes called refactoring browsers
- most modern IDEs have refactoring browsers built-in
- good refactoring browsers apply refactorings as **transactions** (all or nothing) to maintain consistency
- good refactoring browsers can find and refactor references to source code artefacts in comments and configuration files (such as references to servlet class names in web.xml)

Example: Pull Up Method

- addresses duplicated code (method) in classes that have a common superclass
- violates DRY
- redundancy is removed by performing the following three transformations:
 1. copy the method from subclass 1 to the superclass
 2. remove method from subclass 1
 3. remove method from subclass 2

Example: Pull Up Method (ctd)

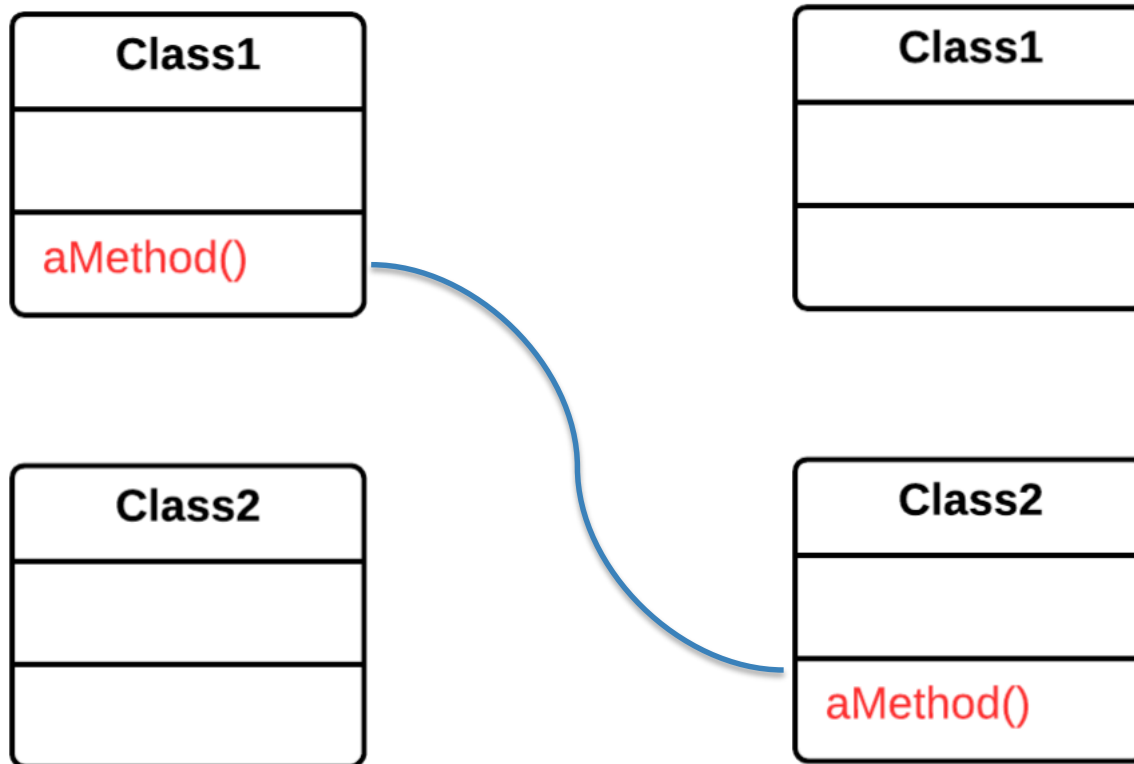


source: <http://refactoring.com/catalog/pullUpMethod.html>

Example: Move Method

- a method is using or used by more features of another class than the class on which it is defined.
- the method logically belong to another class
- results of ([feature envy](#)) code smell
 - method accesses the data of another object more than its own data.

Example: Move Method



Source: <https://sourcemaking.com/refactoring/move-method>

Tools example

- [Eclipse build-in refactoring](#)
 - part of new eclipse releases
 - does basic refactoring operations: rename, move, extract etc....
- [JDeodprent](#)
 - An Eclipse plugin
 - detect five different types of smells:
 - [Duplicated code](#) →
 - [God class](#) →
 - [Long Method](#) →
 - [Type Checking](#) →

Refactoring and Agile SE

- refactoring facilitates agile SE
- it is easier to delay design decisions
- upfront design is not longer necessary, the design changes as project evolves the understanding of the problem increases
- e.g., instead of defining the type hierarchy early, it can evolve using refactorings such as pull up method
- refactoring **encourages and facilitates change**