Review of Object-Orientation

Dr Ian Bayley

Oxford Brookes

Advanced O-O Prog

Outline

1. Introduction to the Module
2. Review of UML Class Diagrams
3. Key Concepts in Object-Oriented Programming
4. Readable code
5. Conclusion

Coursework (50%)

- produce a strategy game with a GUI using MVC (45%)
    - with a Model (5%)
    - specified in JML or Spec# (5%)
    - unit tested with JUnit or .NET equivalent (5%)
    - translated into C# (5%)
    - of good code quality (5%) and, later, with
    - a Controller (5%) and
    - a View for the state of the game (5%) and
    - another View for the scoreboard (5%) and
    - and an artificial intelligence (5%)
- investigate four design patterns of your choice (5%)
    - find four examples of designs and
    - show they conform to design (4× 1%) and
    - translate any one of the classes into Java (1%)

Exam (50%)

- one compulsory question, worth 40%, on object-orientation (this lecture) and design patterns (three more lectures)
- two optional questions, worth 30% each, chosen from three
  - one question on unit testing and Spec#
  - one question on C# and .NET
  - one question on Java 5.0 / 7.0

Lecture Overview

1. Review of Object-Orientation
2. Introduction to Design Patterns
3. Model View Controller
4. More Design Patterns
5. Design By Contract and JUnit testing
6. Coursework Week
7. Remaining Design Patterns
8. Feedback on Patterns, DBC and unit testing
9. Java 5 / 6 / 7 / 8
10. The Language C#
11. The .NET platform
12. Revision Lecture and Feedback on MVC

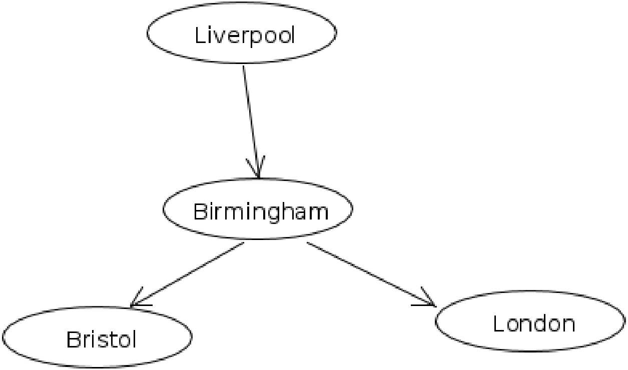
Recommended Reading

- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
    - `http://www.silversoft.net/docs/dp/hires/contfso.htm`
- *Head First Design Patterns* by Elisabeth Freeman and Eric Freeman
- *Head First Java* by Kathy Sierra and Bert Bates
- *C# to the Point* by Hanspeter Mössenböck
- other Java and C# websites of your choice

Outline

1. Introduction to the Module
2. Review of UML Class Diagrams
3. Key Concepts in Object-Oriented Programming
4. Readable code
5. Conclusion

Example of a Directed Graph



Directed Graphs More Formally

Formal Definition of directed graph

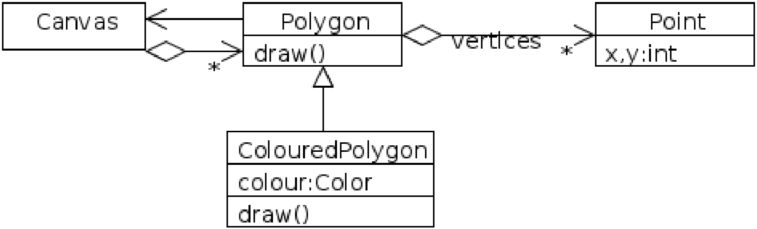
A *directed graph* is a pair (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges connecting these nodes.

- recall that $\subseteq$ means subset and $X \times Y$ is the set of all pairs (x, y) for which $x \in X$ and $y \in Y$
- for later on, recall that $\mathbb{P}\, X$ is the set of subsets of X so $xs : \mathbb{P}\, X$ means xs is a set of X

Example of a directed graph

N={Liverpool, Birmingham, Bristol, London} and E = {(Liverpool, Birmingham), (Birmingham,Bristol), (Birmingham,London)}

Example of a UML Class Diagram



UML Class Diagrams as Generalisations of Directed Graphs

- each node is a class, a template for creating objects
    - not one but *several* sets of edges $E_1, E_2, \ldots = \longrightarrow\!\triangleright, \diamond\!\!\longrightarrow, \ldots$
- each class has a name and, optionally,
    - a set of attributes that the object "knows" about
        - aka fields, instance variables
    - a set of operations (aka methods) that the object can "do"
- so we have partial functions on the type of classes *Class*:
    - $name : Class \rightarrow String$
    - $attrs : Class \rightarrow \mathbb{P}\, Attribute$
    - $opers : Class \rightarrow \mathbb{P}\, Operation$
- parameter and field declarations are written in Pascal style
    - eg `i:int` rather than `int i`, with `i` optional
- abstract operations are written in italics and have no code
- abstract classes are written in italics and can't be instantiated

Semantics of the Relationships Between Classes

Symbol	Meaning
$A \diamond\!\!\rightarrow B$	each A contains an B (“aggregation”)
$A \diamond\!\!\rightarrow^{*} B$	each A contains several B s
$A \longrightarrow B$	each A has a reference to a B (“association”)
$A - \rightarrow B$	a change to B causes a change to A (“static dependency”)
$A \relbar\!\!\triangleright B$	each A has the properties of a B (“inheritance”)

- aggregation and association both translated into Java as fields
  - name of field (“role”) as written over the arrow
- dependency is a weaker relationship (so aggregation and association are special “stronger” dependencies and more)
- translated as a parameter or local variable instead of a field
  - normally used with stereotype creates
- inheritance is translated as the keyword `extends`

Outline

1. Introduction to the Module
2. Review of UML Class Diagrams
3. Key Concepts in Object-Oriented Programming
4. Readable code
5. Conclusion

1. Encapsulation

Definition of Encapsulation

- bundling data with the methods that operate on the data and
- restricting access to some of the object's components

- ie writing classes with private fields and public restrictive getters and setters to maintain business logic eg maximum stack size

Example of Encapsulation - the Stack Abstract Data Type

```
class Stack<E> {
  private ArrayList<E> items;
  public void push(E elem){/* code */}
  public E pop(){/* code */}
}
```

- if a client class didn't "know" how you implemented the stack, you can change the implementation without having to change the client

2. Inheritance

Definition of inheritance

deriving a new specialised class from a base class to reuse code

Example of inheritance

```
class BoundedStack<E> extends Stack<E> {
// items inheritted but can't be accessed by class (except
// with getter) so change access modifier to protected
// pop inheritted without change
  public void push(E elem){ /* blah */ super.push(elem); }
}
```

- you can add a method, add to a method or change it completely
- this example breaks the Liskov Substitution Principle (see later)

3. Polymorphism

Definition of subtype polymorphism

defining a method that has different versions for each subtype

Example of subtype polymorphism

```
abstract class Animal {public void talk();}
class Dog extends Animal {public void talk(){/*woofs*/}}
class Cat extends Animal {public void talk(){/*meows*/}}
```

Example of calling a subtype polymorphic method

```
Animal a = new Dog(); // can change to Cat ...
a.talk(); // ...  and this does not need to change
```

- a client class can change from one type of Animal to another at run time or at compile time and no code needs to be rewritten
    - or one type of Stack to another / one type of Collection to another

The Procedural Programmers Alternative to Polymorphism

```
class Animal {
  int type;
  static final int CAT=1;
  static final int DOG=2;
  void talk() {
    switch (type) {
      case Animal.CAT: System.out.println(“meow”); break;
      case Animal.DOG: System.out.println(“woof”); break;
    }
  }
}
```

Why Polymorphism (the O-O Way) is Thought to be Better

- O-O way makes it easy to introduce a new subtype (eg Cow) without changing existing code (which might already have been tested)
  - the Open-Closed Principle: classes are open to extension but closed to modification
- procedural way requires that the switch statement must be changed
  - there may be other methods (eg eat, move etc) - change every one
- polymorphism is the key to most design patterns, as all can be implemented procedurally
  - and you will be expected to know how to do this
- do not confuse subtype polymorphism with ad-hoc polymorphism (aka overloading) and parametric polymorphism (use of generics)

Outline

1 Introduction to the Module

2 Review of UML Class Diagrams

3 Key Concepts in Object-Oriented Programming

4 Readable code

5 Conclusion

Naming Conventions

`http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367`

broken in subsequent slides, from Martin Fowler's Refactoring website

- variables: informative truthful nouns, in mixed case, start in lower case
- methods: informative truthful verbs, in mixed case, start in lower case
- classes: informative truthful nouns in mixed case starting in upper case
  - eg `String`, `ArrayList`
- interfaces: as for classes but they can be adjectives
  - eg `List`, `Runnable`
- constants: informative truthful nouns all in upper case with underscores as separators

Refactoring to improve readability

Definition of refactoring (due to Martin Fowler)

a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour, using a series of small behaviour-preserving transformations.

- also makes code easier to understand, maintain and debug

Examples of Refactoring

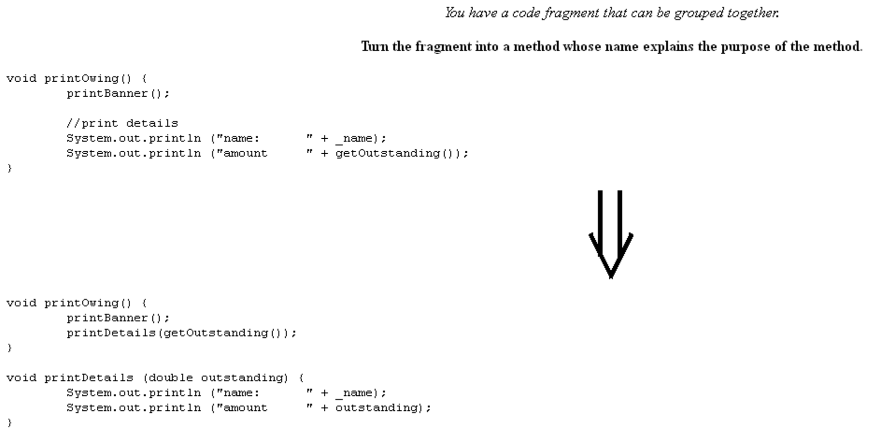
- Rename Method
- Encapsulate Field (make it private, add getters and setters)
- Replace Conditional with Polymorphism (see earlier, in reverse)
- Extract Superclass, Extract Interface
- Extract Class

Bad Smells in Code (and Refactoring Cures)

- Duplicated Code (use Extract Method, Extract Superclass)
    - a maintenance nightmare (as they must be kept the same)
- Long Method (use Extract Method)
- Large Class (use Extract Class)
    - a class should do only one job (related to high cohesion)

- remember that 5% of your module marks is for not being smelly and not having poor variable, method, class, identifier names

Extract Method[1]



You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

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

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

⇓

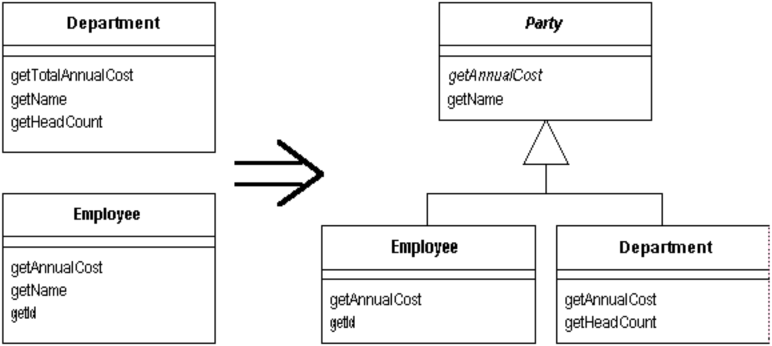
```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

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

[1] from refactoring.com

Extract Superclass[2]

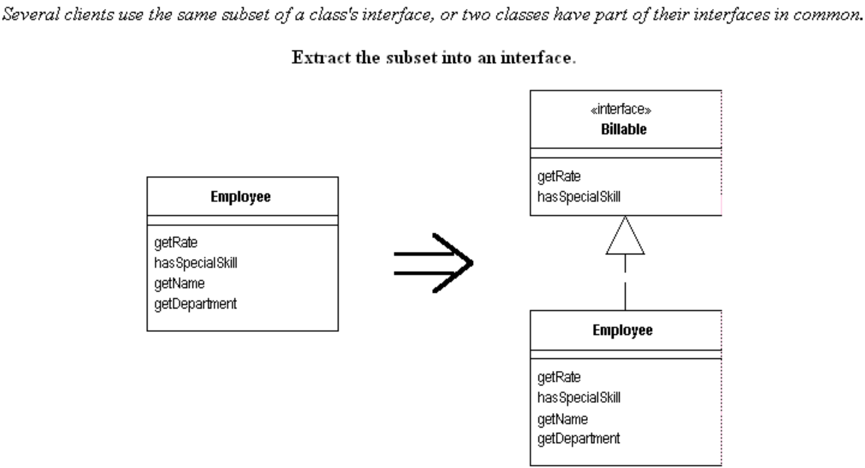
You have two classes with similar features.

Create a superclass and move the common features to the superclass.



[2] from refactoring.com

Extract Interface[3]

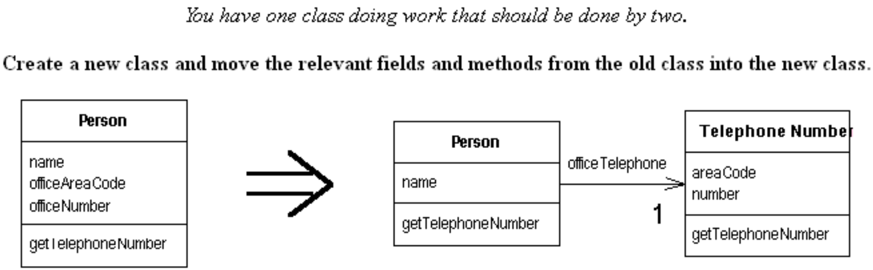


Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.

Extract the subset into an interface.

[3] from refactoring.com

Extract Class[4]

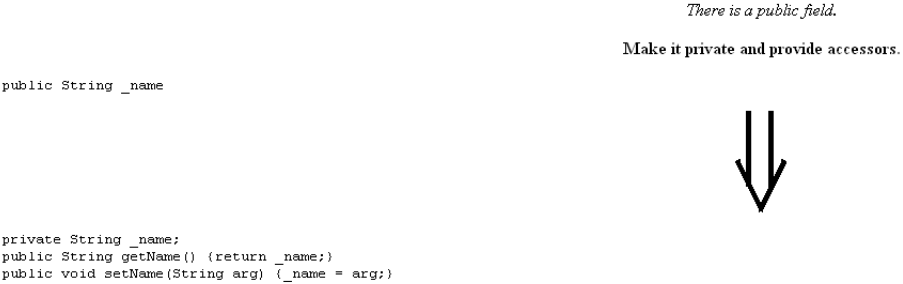


You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.

[4] from refactoring.com

Encapsulate Field[5]



There is a public field.

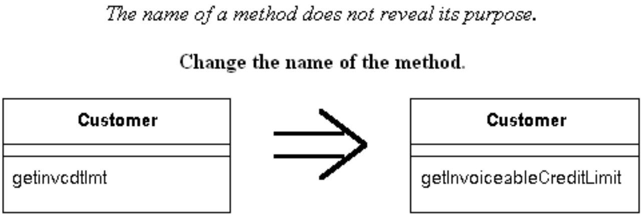
Make it private and provide accessors.

```
public String _name
```

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

[5] from refactoring.com

Rename Method[6]

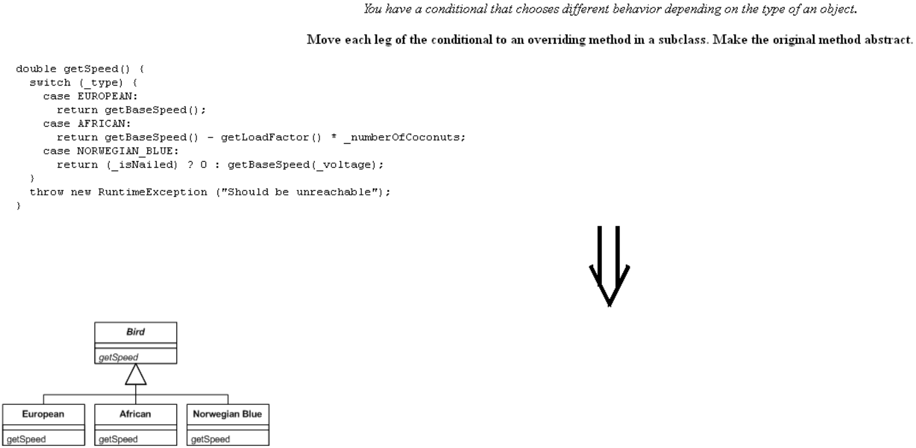


The name of a method does not reveal its purpose.

Change the name of the method.

[6] from refactoring.com

Replace Conditional with Polymorphism[7]



You have a conditional that chooses different behavior depending on the type of an object.

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

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

⇓

[7] from refactoring.com

Outline

1. Introduction to the Module
2. Review of UML Class Diagrams
3. Key Concepts in Object-Oriented Programming
4. Readable code
5. Conclusion

Conclusion

- multi-part coursework
- UML class diagrams as a generalisation of directed graphs
  - nodes labelled with name, attributes, operations
  - types of edges are inheritance, dependency, association, aggregation
- O-O concepts: encapsulation, inheritance, polymorphism
  - the procedural alternative to polymorphism
- naming conventions for constants, variables, methods, classes, interfaces
  - `cat.sitOn(mat);`
- refactorings to improve readability

- in the practical you will *refactor* to introduce *polymorphism*, writing code with *appropriate names*, and draw a *UML class diagram*