COMP241
Software Engineering Development
Lecture 5: Designing Classes

Mark Hall

Readings: Horstman Chap 3

- Overview
- Choosing classes
- Cohesion & Coupling
- Side effects

THE UNIVERSITY OF WAIKATO
DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Overview

- Designing classes can be a challenge
  - How to start?
  - Is the result of good quality?
- Good class design involves understanding and utilizing a number of software design "best practices"
  - Strive for classes that are modular, reusable and bug-free
  - A good class abstracts away implementation detail behind a simple and intuitive interface
  - Apply solutions described in *design patterns*
    - These are good OO techniques applied successfully to different types of problems by others

Choosing Classes

- Identify objects and the classes to which they belong
  - Rule of thumb: Class names should be **nouns**, and method names should be **verbs**
- What makes a good class?
  - A class should *represent a single concept*
  - Eg. Concepts from mathematics:
    - `Point`, `PlanarPoint`, `Rectangle`, `Eclipse`
  - Other classes are abstractions of real-life entities
    - `BankAccount`, `LibraryBook`, `Customer`

Choosing Classes

- For these classes, the properties of a typical object are easy to understand
  - A `Rectangle` object has width and height
  - Given a `BankAccount` object you can deposit and withdraw money
- Generally, concepts from the part of the universe that our program concerns make good classes
  - The name for such a class should be a **noun** that describes the concept

Choosing Classes

- Another useful category of classes are those that do some kind of work for you
  - A `StreamTokenizer` object breaks up an input stream into individual tokens
  - A `Random` object (from the `java.util` package) generates random numbers
- It is a good idea to choose class names for these types of classes that end in "-er" or "-or"
  - A better name for `Random` might be `RandomNumberGenerator`

Choosing Classes

- Very occasionally, a class has no objects, but it contains a collection of related **static** (class) methods and constants
  - Eg `java.lang.Math:`
    - **`static`** `double PI`
    - **`static`** `double E`
    - **`static`** `double abs(double a)`
    - **`static`** `double cos(double a)`
    - Etc.
  - Such a class is called a *utility* class

Choosing Classes

- What might not be a good class?
  - If you can't tell from the class name what an object of the class is supposed to do
- Eg. Say you had an assignment to write a program that prints pay-cheques
  - You might decide to design a `PaychequeProgram` class
    - What would an object of this class do? Everything that the assignment requires!
  - To simplify things, it would be better to have a `Paycheque` class. Then you program could manipulate `Paycheque` objects
    - The `Paycheque` class might also be able to be re-used in another application

Choosing Classes

- What might not be a good class?
  - Turning a **function** into a class
- Eg. A `ComputePaycheque` class
  - Can you visualize a "ComputePaycheque" object?
  - "ComputePayCheque" isn't a **noun**
- On the other hand, a `Paycheque` object makes intuitive sense
  - You can think about useful methods of the `Paycheque` class, such as `compute`

Cohesion and Coupling

- Cohesion and coupling are useful criteria for analyzing the quality of the public interface of a class
- Cohesion: A class should represent a single concept
  - The public methods and constants exposed by the interface should be ***cohesive***—ie. all interface features should be closely related to the single concept that the class represents
- If the public interface to a class refers to multiple concepts, then it may be time to use separate classes instead

Cohesion

- Consider the public interface for a (American) `Purse` class:

```
public class Purse {
   public Purse() { . . . }
   public void addNickels(int count) {. . .}
   public void addDimes(int count) {. . .}
   public void addQuarters(int count) {. . .}
   public double getTotal() {. . .}
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   . . .
}
```

Cohesion

- There are actually **two** concepts being referred to by the `Purse` interface:
  - A purse that holds coins and computes their total
  - The values of the individual coins
- It would make more sense to have a separate `Coin` class and have coins responsible for knowing their own values

```
public class Coin {
   public Coin(double value, String name) {. . .}
   public double getValue() {. . .}
   . . .
}
```


Cohesion

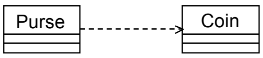
- Then the `Purse` class can be simplified:

```
public class Purse {
   public purse() {. . .}
   public void add(Coin aCoin) {. . .}
   public double getTotal() {. . .}
   . . .
}
```

- This is clearly a better solution
  - Separates the responsibilities of the purse and the coins

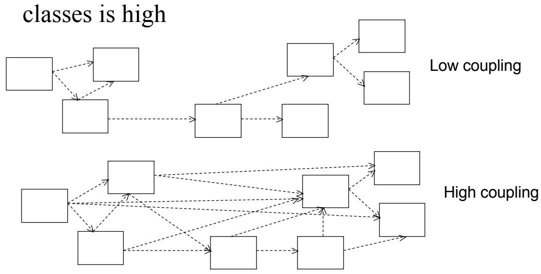
Coupling

- Many classes need other classes to do their job
  - Eg. the restructured **`Purse`** class now *depends* on the **`Coin`** class to determine the total value of the coins in the purse
- Note that the `Coin` class does not depend on the `Purse` class
  - Coins have no idea that they are being collected in purses, and they can carry out their work without ever calling any method in the **`Purse`** class



Coupling

- If many classes of a program depend on each other, then we say that *coupling* between classes is high



Coupling

- Why does coupling matter?
  - If the `Coin` class changes in the next release of the program, all the classes that depend on it **may** be affected
  - If the change is drastic, the coupled classes must all be updated
  - Furthermore, if we want to use a class in another program, we have to take with it all the classes on which it depends
- Thus, we should remove unnecessary coupling between classes

Accessor and Mutator Methods

- *Accessor* method
  - A method that accesses an object and returns some information about it, **without changing** the object
- *Mutator* method
  - A method whose purpose is to modify the state of an object

Accessor and Mutator Methods

```
public class BankAccount {
  private double mBalance;

. . .

  public void deposit(double amount) {
    mBalance = mBalance + amount;
  }

  public double getBalance() {
    return mBalance;
  }

. . .
}
```

Mutator method

Accessor method

Accessor and Mutator Methods

- As a rule of thumb, it is a good idea for mutators to have return type `void`
  - Makes it easy to differentiate between mutators and accessors
- You can call an accessor method as many times as you like—you always get the same answer
  - Does not change the state of the object
  - Makes the behaviour of such a method very predictable

Accessor and Mutator Methods

- Some classes have been designed to have **only** accessor methods
  - Such classes are called *immutable*
  - Eg. `String` class—once constructed, its contents never change
  - Advantage of an immutable class: it is safe to give out references to its objects freely
    - No code can unexpectedly modify an object

Side Effects

- All instance methods have one *implicit* parameter—a reference to the object containing the method
  - Can be accessed (if necessary) via the **`this`** keyword
- A method may have zero or more *explicit* parameters—ie. those that are passed in as arguments
- If a method modifies some outside value other than its implicit parameter, we call that modification a *side effect*
  - ie. a side effect is *any kind of observable behaviour* outside the object

Side Effects

```
public void transfer(BankAccount other, double amount) {
   withdraw(amount);
   other.deposit(amount);
}
```

Modifies the *implicit* parameter—ie. this object's state is changed

Modifies an *explicit* parameter—ie another objects state is changed

- As a rule of thumb, updating an explicit parameter can be surprising to programmers
  - Best to avoid it whenever possible

Side Effects

- Another example of a side effect is output
- Consider printing a bank balance:

```
System.out.println("The balance is now $" +
     mySavings.getBalance());
```

- We could simply have a `printBalance` method instead:

```
public void printBalance() {
   System.out.println("The balance is now $" +
      mBalance);
}
```


Side Effects

- `printBalance` would be more convenient when we actually want to print the value
  - However, can't just drop `getBalance` in favour of `printBalance` as there cases we might want the value for other purposes
- `printBalance` forces strong assumptions on the `BankAccount` class
  - The message is in English
  - Relies on `System.out`—may not work in an embedded system such as an ATM
- A method with side effects introduces additional dependencies and thus increases coupling

Side Effects

- Side effects cannot be completely eliminated in an OO programming language
  - Can be the cause of surprises and problems and should be minimised when possible
- Classification of method behaviour:
  - *Best:* **Accessor** methods with no changes to any explicit parameters—no side effects. Eg: `getBalance`
  - *Good:* **Mutator** methods with no changes to any explicit parameters—no side effects. Eg: `deposit`
  - *Fair:* Methods that change an explicit parameter: Eg `transfer`
  - *Poor:* Methods that change a **static** field of another class

Summary

- Designing good classes is a learned art
- Applying rules of "best practice" and common sense can help
- Look for classes that
  - Have a good descriptive name that is a noun
  - Are cohesive with low coupling (dependencies)
  - Have minimal side effects (externally observable behaviour)
- More on designing classes in the next lecture