MODULE: Coupling and cohesion

• Central terms when talking about the quality of design

• Coupling – concerns relationships between modules

• Cohesion – concerns relationships within a module*

• Goal: We want loosely coupled modules with high internal cohesion

*) A “module” is here used in the sense of a class or of a unit consisting of several classes (e.g. a “software component” with its own package hierarchy). Cohesion and coupling are also relevant at the level of logical tiers (resource:integration:logic:presentetion:view).
Coupling

• “Interconnectedness”

• The more one module has to “know” about another module, the higher is the coupling between them

• Modules should be as independent as possible

• Modules should communicate only via small, well-defined (“narrow”) interfaces
Coupling

• Directly related to how hard it is to make changes in a program (– “localizing” change)

• Interfaces help separate “what” from “how”

• Proper encapsulation reduces coupling

• “Responsibility-driven design” (each unit should handle its own data)
Implicit coupling

Change in one unit affects other units implicitly

1. Ex: Adding a new command to a GUI, but forgetting to add it to the help menu… (how could this be fixed?)

2. Ex: Adding a new command to the help menu, but forgetting to add help text for the command in the help function… (how could this be fixed?)
In class `DocPrint`:

```java
public void printAddress(StudentRecord sr) {
    // Here we need to know how to extract
    // the relevant info from StudentRecord.
    // What if class StudentRecord change?
}
```

OR:

```java
public void printAddress(String address) {
    // This will print any address, not only student’s
}
```
Code duplication

• Sign of bad design…

• Same code in two places $\rightarrow$ high coupling

• Makes it difficult to modify and extend

• Duplicated code often is an indication that a better solution (in the implementation) is possible (cf. “refactoring”)
Cohesion

• A module should provide a well-defined task (service)

• A module should have well-defined responsibilities

• High cohesion facilitates reuse (well-defined modules)

• High cohesion simplifies modification (all relevant code in one place)

• High cohesion $\rightarrow$ low coupling to other modules (but high coupling within the module..)

• Reducing coupling $\rightarrow$ higher cohesion
Cohesion

• When should we subdivide (and thus try to attain low coupling and between the different parts)?

• Subdivide when:
  – You cannot keep all aspects of the module in memory
Cohesion, classes

• Do the methods take advantage of having direct access to the representation?
  – Some static methods may conceptually belong to a type although they do not use the type’s representation (e.g. `computeMileageAllowance(int miles)` in class `Employee`)

• Do the methods conceptually “belong” to the type?
  – E.g. `getSalary()` in class `PersonnelList`

• Is a method “doing the work” for some other class!
Cohesion, methods

- Applicable also to methods

- High cohesion
  - short method
  - one task
  - descriptive name

- Possible indicators of low cohesion
  - difficult to describe what the method does
  - difficult to to give the method a name
  - If-then-else, switch
Cohesion, methods

/** This returns the first or last element of an ordered collection, depending on flag. */
Object getEnd(List l, boolean firstFlag) {
    if (firstFlag) {
        // return first element
    } else {
        // return last element
    }
}

Why not use two separate methods instead? (That would be easier to understand and to use, and it would eliminate the ‘firstFlag’ argument.)
“Law of Demeter”

• Inside a method M it is only permitted to use/call:
  1. methods of arguments to M
  2. methods of this
  3. public static variables and methods
  4. local variables

• There must be some observer (accessor) methods, and these must obviously be allowed to also call methods in instance variables

[Lieberherr, 1989 (modified from Budd, 1991)]
Which statement?

class Person {
    private String name;
    private Address addr;
    
    String s;
    // s = getStreet() { return addr.getStreet(); }  
    // s = String getStreet() { return addr.street; }  
    // s = addr.getStreet();  
    // s = addr.street;  
    }

MODULE: Testing and debugging

• Black-box testing
  – testing based on specification only
  – requires no knowledge of software

• Glass-box testing
  – allows for path-complete test sets (testing each path through the program)
  – requires knowledge about the code
Testing

• Unit testing – testing of each unit (module, class) individually (finds implementation errors)

• Integration testing – testing of the entire program (finds specification errors)

• Regression testing – (automatic?) re-running of all tests whenever a change is made
Driver and Stub

• Driver – code that automatically runs tests for a module (e.g. in the form of a test-main())

• Stub – code that simulates the behavior of any external modules used by the module being tested (e.g. a user)
class ClassA {
    public static void main(String[] args) {
        // Run tests for ClassA
    }
}

class TestAll {
    public static void main(String[] args) {
        ClassA.main(null);
        ClassB.main(null);
        ClassC.main(null);  // Could have arguments!
    }
}

...or write a test()-method for each class to be tested!
Debugging

- Debugging usually requires more time than programming (so it’s worthwhile getting it right the first time!)

- The “scientific” method:
  - Study available data
  - Form a hypothesis (in the form of a test?)
  - Try to prove the hypothesis wrong

- Useful tools:
  - testing!
  - assertions
  - toString()
  - code coverage analysis
  - (IDE with tracing)
  - etc.
Debugging

- Keep an open mind
- Try the simple (“too obvious”) things first
- Get somebody’s help (your reasoning may be “locked”)
- The problem is almost always in your code (it’s not in OS, or due to bugs in standard classes, etc.)
- Are you editing one file and running another?
- When finding a bug, search for others like it…
• Inheritance – “is-a” relation

• Realization – “is-a” relation, realization of a type

• Association – “uses-a” relation, often used in a general sense for some connection (any connection) between objects, classes or modules

• Aggregation – “has-a” relation, e.g. a car stereo

• Composition – “consists-of” relation, e.g. a car’s chassis

Note: There is some disagreement regarding the meaning of “association”, “aggregation”, and “composition” (cf. Fowler, 1997: “UML Distilled”, p. 57)
• Inheritance represents an “is-a” relationship

• The subtype must conceptually be a supertype

• Extension-specialization paradox (extension as seen from outside, providing more services; specialization as seen from inside: child objects constitute a subset of parent objects)

• Subtypes provide:
  – redefined functionality, and/or
  – extended functionality
Inheritance, UML

ParentClass

aParentMethod():void

generalization

ChildClass

The parent’s methods are not repeated here...
Realization

• An interface is a type

• Contains only abstract methods, no implementation

• An interface is realized (implemented) by one or more classes

• A class can implement more than one interface (can be of more than one type)

• A class that use an interface is a dependant class
If interface changes, the dependant class may have to change too.
BankGUI

<<interface>>
Bank

CustRegGUI

<<interface>>
CustomerRegistry

BankImpl
• A “uses-a” relation

• In general terms: two classes are associated if one need to know about the other…

• Can often be identified through verbs in a description, e.g. “the library customer **borrows** the book” (the nouns are the potential classes)
Aggregation

- A “has-a” relation

- Exchangeable objects

- Not necessarily a lifelong relationship

- E.g. a car has a stereo, a person has a car, etc.

- (May be regarded as a special type of association)
Aggregation, UML

Library

Book

Author
Composition

• A “consists-of” relation (stronger than aggregation)

• One object is not valid without the other

• A life long relationship (created in constructor, not exchangeable)

• E.g. a car consists of (among other things) a chassis (and it will not be a car without it)

• (May be regarded as a special type of association)
Composition, UML

Is this really composition?
MODULE: GUI

• Context – a graphical component that can generate events

• Event – an object representing an event; knows its source context

• Command – a method defining action(s) for a certain event
In Java

- **Context** – `Component, Container, Button, TextField, etc.`

- **Event** – `WindowEvent, ActionEvent, ItemEvent, etc.`

- **Command** – methods in Listener classes, such as `actionPerformed()`, `windowClosing()`, `itemStateChanged()`, etc.

- `java.awt` and `javax.swing` contains many examples of design patterns...
java.awt.Component

Component

Button | TextField | Canvas | CheckBox

Container

Window | Panel

Frame | Applet
Toolkit

Component

extends

uses

extends

Button

extends

MyButton

toolkit

createButton()

extends

xxxToolkit

extends

OS-specific Toolkit

(internal) factory
import java.awt.*;

public class MyGui {
    private Button b;
    public MyGui() {
        Frame f = new Frame("My window");
        f.setBounds(100,100,200,50);
        Panel p = new Panel();
        b = new Button("Press me!");
        frame.add(p);
        p.add(b);
        f.setVisible(true);
    }

    public static void main(String[] args) { new MyGui(); }
}
Events

**Component**

processEvent()

**AWTEvent**

- **Listener1**
- **Listener2**
- **Listener3**

Event queue

'klick'
import java.awt.*;
import java.awt.event.*;

public class MyGui {
    private Button b;
    public MyGui() {
        b.addActionListener(new MyButtonListener());
        f.setVisible(true);
    }
}

class MyButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        System.out.println(“Button was pressed!”);
    }
}
import java.awt.*;
import java.awt.event.*;

public class MyGui {
    private Button b;
    public MyGui() {
        ActionListener al = new MyButtonListener();
        b.addActionListener(al);
        f.setVisible(true);
    }

    private class MyButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            if (evt.getSource() == b) {
                System.out.println("Button was pressed!");
            }
        }
    }
}

Is this a good way of doing it..?
import java.awt.*;
import java.awt.event.*;

public class MyGui {
    public MyGui() {
        
        f.addWindowListener(new MyWindowListener());
        f.setVisible(true);
    }
}

class MyWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent evt) {
        System.exit(0);
    }
}

Note: has nothing to do with design pattern “Adaptor”
(BTW: is this a good name?)
An anonymous class, ex

```java
import java.awt.*;
import java.awt.event.*;

public class MyGui {
    public MyGui() {
        .
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent evt) {
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}
```

Could also be an interface
Separate the user interface from the business logic (the “functional” part)

Either one should be possible to modify / change without affecting the other

Allow for different types of UI (for instance, a simple command interpreter, a graphical interface, or a web interface…)

UI / logic connection

GUI

Business logic

Get data from business logic component

Component

Send update signal to GUI when data has changed

Get data from business logic component
Connection via interface

BankGUI -> <<interface>> Bank

implements

BankImpl
The interface

```
BankGUI

bank: Bank

BankGUI(Bank b)
    getCustomer(String id): CustInfo
    deposit(String id, int amnt): void
    Withdraw(String id, int amnt): void

<<interface>>
Bank

getCustomer(String id): CustInfo
    deposit(String id, int amnt): void
    Withdraw(String id, int amnt): void
```
Observer / Observable

BankGUI uses <<interface>> Bank

BankImpl implements Bank

signals
Observer / Observable

Also known as “Publish / Subscribe”

BankGUI

uses

Bank

implies

Observer

BankImpl

implies

extends

Observable

signals
Observer / Observable, ex

```java
interface Observer {
    public void update(Observable o, Object arg)
}

public class Observable {
    public void addObserver(Observer o) {...}
    public void deleteObserver(Observer o) {...}
    public void notifyObservers(Object arg) {...}
    protected void setChanged() {...}
    protected void clearChanged() {...}
    .
    .
}
```
Composite

• Representations of part-whole hierarchies

• Allows for uniform treatment of objects and compositions of objects
Expression: \((1+(2*3))-(4*5)\)

Binary tree:

```
        +
       /|
      /  *
     */ 1
    */  4 5
   2  3
```

UML:

```
Expression
  eval()

Constant
eval()

CompositeExpression
eval()
```

contains
Component
paint()

contains

Button
paint()

Container
paint()

TextField
paint()
Panel p0 = new Panel();
p0.add(new Button());
p0.add(new TextField());
Panel p1 = new Panel();
p0.add(p1);
p1.add(new Button());
p1.add(new Button());
p1.add(new TextField());
Panel p2 = new Panel();
p1.add(p2);

p0.repaint(); // calls paint()

- Method `paint()` will do different things depending on whether the object is a container or not.