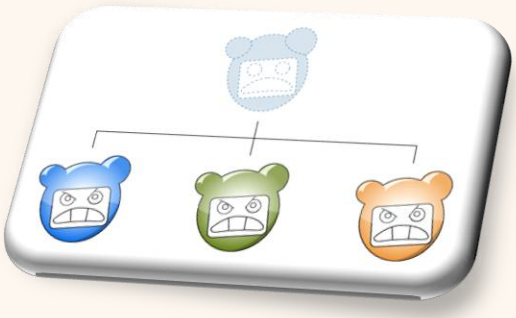
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 8

OOP principles



Design patterns

Ex1,2 Feedback

- Problem: we want to build a set of media players
  – Display images, play sound, play video

- Each player:
  – Needs to know its operation system (unix, windows, mac)
  – Can play a file of its media type
  – Playing a file requires retrieving file information (same for each media type). **Not** part of the player's public API

- Possible solution:
  – Use inheritance

MediaPlayer

```
public abstract class MediaPlayer{
    String os;

    public MediaPlayer(String os) {
        this.os = os;
    }

    public abstract void play(File file);
}

public class SoundPlayer extends MediaPlayer { ... }

public class ImagePlayer extends MediaPlayer { ... }

public class VideoPlayer extends MediaPlayer { ... }
```

- Think about your modifiers!
- Use the empty (default) modifier only if this is what you meant to do

MediaPlayer Classes

```
public class SoundPlayer extends MediaPlayer {
    ~~private String os;~~

    public SoundPlayer(String os) {
        super(os);
        ~~this.os = os;~~
    }

    ....
}
```

- No need to redefine parent class data members
- No need to run code that runs in **super** constructor

Retrieving File Information

```
public class SoundPlayer extends
    MediaPlayer {
    private String retri      (File file) {
        ...
    }

     public void play(File file
        String info = retrieve
        ...
    }
}
```

```
public class VideoPlayer extends
    MediaPlayer {
    privat      retrieveInfo(File file) {



           void play(File file) {
        String info = retrieveInfo(file);


}
```

Retrieving File Information

```
public abstract class MediaPlayer {
    protected String retrieveInfo(File file) {
        ...
    }

    public abstract void play(File file);
}
```

```
public class VideoPlayer extends
    MediaPlayer {

    public void play(File file) {
        String info = retrieveInfo(file);
        ...
    }
}
```

- Inheritance can be used to share code
- No need to implement the same method in every extending class

Overview

- There are many important Object Oriented Design (OOD) principles

- Today we'll focus on several basic principles closely related to design-patterns:
  - Modularity
  - The Open-Closed principle
  - The Single-Choice principle

Modularity

- A Modular design results in a software that can be broken down to several individual units, denoted **modules**

- Modularity is a desired property of software design

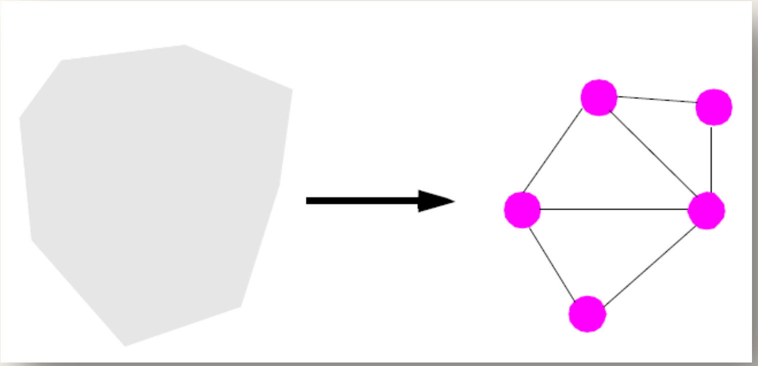
- Modular programs have several benefits
  - Easy to maintain (debug, update)
  - Allow breaking a complex problem into easier sub-problems
  - Allow to easily divide the project into several team members or groups

Modularity

- A design method which is “modular” should satisfy 4 fundamental requirements:
  - Decomposability
  - Composability
  - Understandability
  - Continuity

Decomposability

- A software construction method satisfies *Modular Decomposability* if it:
  - Decomposes a software problem into a small number of **less complex** sub-problems
  - These sub-problems are connected by a **simple structure**, and independent enough to allow further work to proceed separately on each of them



Decomposability

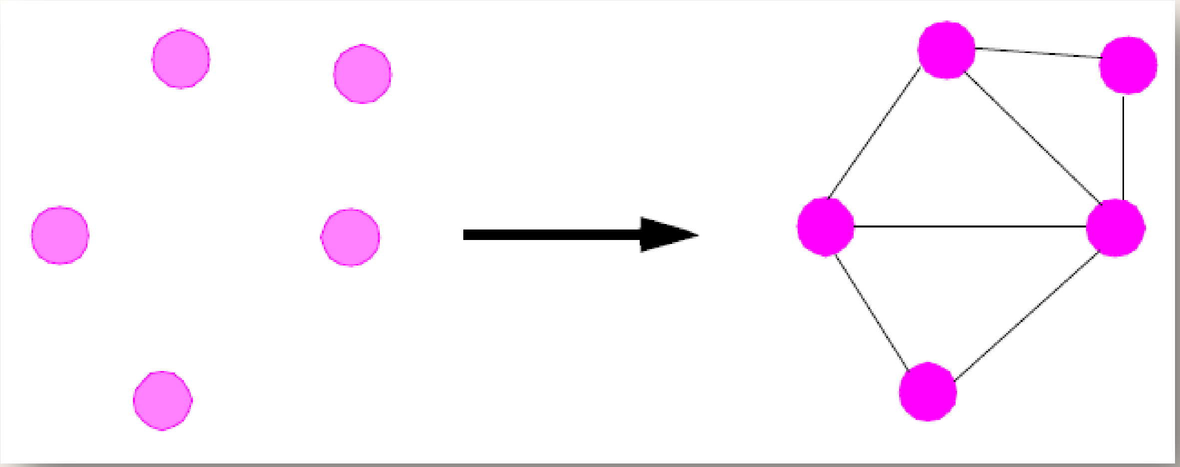
- A corollary of decomposability is division of labor
  - A decomposed system is easier to distribute among different people or groups

- **A good example**: Top-Down Design

- **A typical counter-example**: a software system that includes a global initialization module (why?)

Composability

- A method satisfies *Modular Composability* if it produces software elements which may be freely ***combined with each other to produce new systems***
  - Possibly in an environment quite different from the one in which they were initially developed



Composability

- Elements should be sufficiently **autonomous**

- Composability is directly connected with the goal of reusability
  - Design software elements performing well-defined tasks and usable in widely different contexts

- Examples: Software libraries (or packages)

Composability Vs. Decomposability

- The principles of composability and decomposability are **independent**
  - In fact, these criteria are often at odds
  - Top-down design, for example, which we saw as a technique favoring decomposability, tends to produce modules that are not easy to combine with modules from other sources

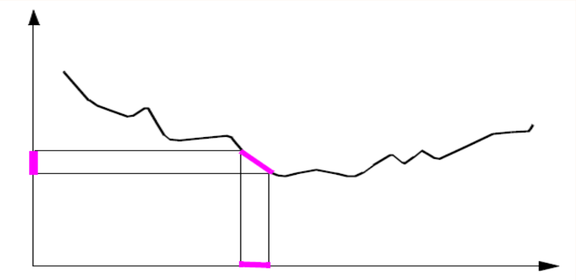
Understandability

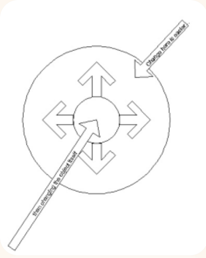
- A software design favors *Modular Understandability* if it produces software in which a human reader can understand each module without having to know anything about the others
  – At worst, by having to examine only a few of the others
  – Important for the maintenance process
  – Rule of thumb: can you explain in a few words what each module does?

- This is **not** the same as readability

Modular Continuity

- **Continuity** – A method satisfies *Modular Continuity* if, in the software architectures that it yields, a small change in the problem specification triggers a change in just one module, or a small number of modules
  - Minimize dependencies between different modules
  - The term “continuity” is drawn from an analogy with the notion of a continuous function in mathematical analysis



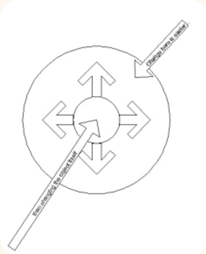


The Open-Closed principle

- Software Entities (Classes, Modules, Functions, etc.) should be **open for** **extension** **but closed for** **modification** (Meyer, 1988)

- A single change to a program ➔ a cascade of changes to dependent modules ➔ **“bad” design**
  - The program becomes fragile, rigid and un-reusable
  - Violation of the *Continuity* principle

- The open-closed principle tackles this issue by stating that you should design modules that **never change**
  - Requirements change ➔ extend the modules by **adding new code**, not by **changing old code** that already works!



The Open-Closed principle

- Modules that conform to the open-closed principle have two primary attributes:
  – They are "**Open for Extension**": the behavior of the module can be extended; we can make the module behave in new and different ways as the requirements of the application change
  – They are "**Closed for Modification**": the source code of such a module is inviolate. No one is allowed to change it

The Open-Closed principle
Shape example

- We have an application that must be able to draw circles and squares on a standard GUI
  - The circles and squares are drawn in a particular order
- The program traverses and draws a list of circles and squares in a given order
- Possible solutions:
  - Procedural solution – shape type is queried each time we draw it
  - OOP solution – a common interface is used

Shape Example

Procedural Solution

```
void drawAll(Shape[] list) {
    for (int i=0; i<list.length; i++) {
        Shape s = list[i];
        int type = getType(s);
        switch (type){
          case SQUARE:  drawSquare((Square)s);  break;
          case CIRCLE:   drawCircle((Circle)s);  break;
        }
    }
}
```

Procedural Solution

What's Wrong Here?

- The *drawAll*() method does not conform to the open-close principle
  - It is not **closed** against new kinds of shapes
  - If we wanted to **extend** this function to draw a list of shapes that included **triangles**, we would have to **modify it**

- This program is only a simple example
  - The switch statement in *drawAll*() could be **repeated over and over** again in various functions, each one **doing something a little different**
  - Adding a new shape means hunting for every such switch statement and adding the new shape

The Shape Example

OOP Solution

```
public interface Drawable{
    public void draw();
}
public class Square implements Drawable {
    public void draw(){..}
}
public class Circle implements Drawable {
    public void draw(){..}
}
public void drawAll (Drawable[] list) {
    for (Drawable drawable: list)
        drawable.draw();
}
```

OOP Solution – Advantages

- Extending the behavior of the *drawAll*() method to draw a new kind of shape, is done by adding a new implementation of the *Drawable* interface
  - *drawAll*() **does not need to change**

- *drawAll*() now conforms to the open-closed principle. Its behavior can be extended without modifying it

The Single-Choice Principle

- Whenever a software system must support a set of alternatives, **one and only one** module in the system should know their exhaustive list

- By doing this, we prepare the scene for later changes:
  - If variants are added, we only have to update the module which has the information — the point of single choice
  - All others, in particular its clients, are able to continue their business as usual
  - This principle interacts with the **open-closed** principle:
    - Keep our exhaustive list of options in one place, so that this is the only place the needs to be changed upon updates

The Shape Example

OOP Solution

```
Drawable[] loadAll (String[] list) {
    Drawable[] drawables = new Drawable[list.length];
    for (int i = 0 ; i < list.length ; ++i) {
        if (list[i].equals("Square")) {
                drawables[i] = new Square();
        } else if (list[i].equals("Circle")) {
                drawables[i] = new Circle();
        } ...
    }
    return drawables;
}
```

One method must know all the options.
It **cannot** be closed for changes

The Shape Example

OOP Solution

```
void drawAll(Drawable[] list) {
    for (Drawable drawable: list)
        drawable.draw();
}

void deleteAll (Drawable[] list) {
    for (Drawable drawable: list)
        drawable.delete();
}
```

All other methods don‘t need to know the options.
They **are closed** to changes

So far…

- Among most important OOP design principles
  - Modularity
  - Open-closed
  - Single-choice

Design Patterns

“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 and over, without ever doing it the same way twice” (Christopher Alexander, GoF, page 2)

- Alexander was an architect who studied ways to improve the process of designing buildings and urban areas

Essential Elements of a Pattern

1. **Pattern Name**
   - Having a concise, meaningful name for a pattern improves communication among developers

2. **Problem**
   - What is the problem and context where we would use this pattern?
   - What are the conditions that must be met before this pattern should be used?

Essential Elements of a Pattern

3. **Solution**
   - A description of the elements that make up the pattern
   - Emphasizes their relationships, responsibilities and collaborations
   - Not a concrete design or implementation; rather an abstract description

4. **Consequences**
   - The pros and cons of using the pattern
   - Includes impacts on reusability, portability, extensibility

Design Patterns Properties

- Describes a proven approach to dealing with a common situation in programming / design
- Suggests what to do to obtain an elegant, modifiable, extensible, flexible & reusable solution
- Shows, at design time, how to avoid problems that may occur much later
- Is independent of specific contexts or languages

Design Patterns Types

- The different design patterns can be (roughly) divided into 3 different categories
  - Creational patterns
    - These patterns deal with creating objects (**instantiation**)
    - For example: **Factory**
  - Structural patterns
    - These patterns deal with the objects' structure (**composition**)
    - For example: **Delegation**
  - Behavioral patterns
    - These pattern handle the objects behavior (**communication between objects**)
    - For example: **Iterator**

Reminder

- Let A,B be 2 classes
  - A Composes B if
    - A **holds an instance** of B (as a member or a local variable)

Delegation

- Delegation is a way of making **composition** as powerful for reuse as **inheritance**
- In delegation, two objects are involved in handling a request: a receiving object delegates operations to its **delegate**
- This is analogous to subclasses deferring requests to parent classes
- A **structural** pattern

Delegation vs. Inheritance

```
public class B {
        public void foo() { … }
}
```

```
// Delegation
public class A {
        private B b;

        public A(B b) {
                this.b = b;
        }

        public void foo() {
                b.foo();
        }
}
```

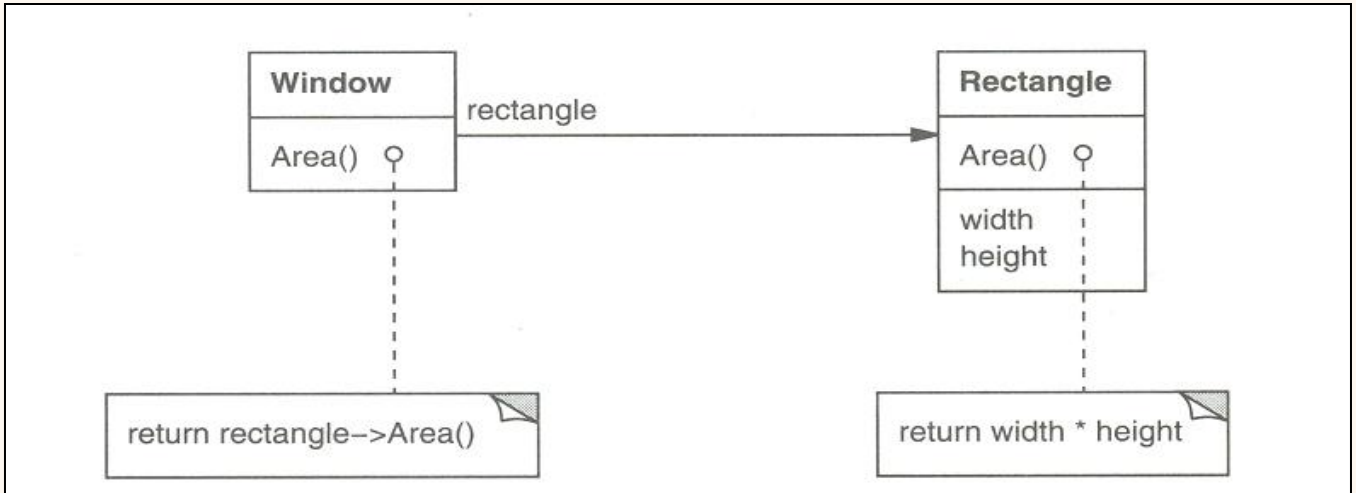
```
// Inheritance
public class C extends B {…}
```

A delegates its *foo*() requests to b

Both A.foo() and C.foo() now call B.foo()

Delegation

- Question: Should Window **be** a Rectangle or **have** a Rectangle?
- Answer: It should have a rectangle
  – The window class might reuse the behavior of Rectangle, by keeping a Rectangle instance variable and delegating Rectangle-specific behavior to it



Delegation

- Main advantage – easy to compose behaviors at runtime and to change the way they are composed
  - Example: easy to make the window circular, by replacing the Rectangle instance with a Circle instance

- Disadvantages:
  - Design is a bit more complex

- **In summary:** a good design choice when it simplifies more than it complicates!

To Summarize

- Let A,B be 2 classes
  - A Composes B if
    - A **holds an instance** of B (as a member or a local variable)
  - A Delegates B if
    - A **composes** B and **forwards requests** to the composed instance (of type B)'s **methods**

Factory Patterns

- A **factory** is an object used to create other objects
  - A factory has methods for creating the objects it supports
  - These methods may receive parameters that define the type and properties of the created object

- Factories are used in situations where deciding which object to create is a complex task
  - The factory may create the object dynamically, return it from some object pool, do complex initialization, etc.

- Many **creational design patterns** implement this concept

The Singleton Pattern

- Intent
  - Ensure a class only has exactly **one instance**, and provide a global point of access to it

- Motivation
  - If we want a single instance of a class to exist in the system
    - We want just one window manager, or just one factory for a family of products…
  - We need to have that one instance easily accessible
  - Additional instances of the class cannot be created

- A **creational** pattern, which implements the **factory** concept

Implementing Singletons

- Storing the single instance as a **static data member**
- Make the constructor private
- Creating the instance in the **static** *instance*() method
- This method always returns a reference to the same single **static** object
  - This way only one instance at most is created

Singleton example

```
public class Singleton {
    private static Singleton single = null;

    private Singleton () { … }

    public static Singleton instance() {
        if (single == null) {
            // This line is called at most once
            single = new Singleton();
        }

        return single;
    }
}
```

Singleton and Subclassing

- Using a single private constructor makes it **impossible** to subclass
  - This isn’t such a big problem, as in many cases subclassing singletons is not a good idea

So far…

- Design patterns
  - A good way to solve common problems
  - Name, problem, solution, consequences

- A few design patterns
  - Delegation
  - The Factory concept
    - Singleton