Design Patterns (DP)
- It’s not a course about DP (just a little)
- A lot of good design and efficient implementation is based on DP
- In order to make sense, DPs require experience
  - They can be hard to understand at first
  - They can be hard to master, knowing when to use, how to use etc.

In the beginning…
- Francesco di Giorgio (1439-1501)
  - Master builders in collected drawings of successful buildings
  - The drawings were not precise drawings, but contained examples of good solutions and how they worked
The Flower Power years

- Christopher Alexander
- Mathematician that became an architect
  - Notes on the Synthesis of Form, 1964
  - A Pattern Language, 1977
  - The Timeless Way of Building, 1979

Why are there bad buildings?

- In 'Notes' Alexander discusses how bad some buildings are:
  - "Buildings are not characterized by adaptability, quality or usability. Instead they are based on existing unspoken 'rules'. Over time the buildings are adapted for use, and some of the rules are modified for the better – without any explicit reason."

Alexandrian patterns

- Alexander's view is that the buildings are bad because the people that are going to live there are not involved in the design process.
- He says that what matters is what happens in the building. This is nothing a designer can control. There has to be user involvement.
Alexander’s conclusions

- In order to enable user participation, we all need to speak the same "language".
- This language is Design Patterns.
- Patterns are tools.

DP a background in Smalltalk

- Ward Cunningham & Kent Beck had in 1987 problem to complete a project and let a couple of users finish the design
  - The result was 5 Smalltalk DP

First DP ever!

- #1: WindowPerTask
  - Make a specific window for each task the user must perform. All of the information needed to complete a task should be available in the FewPanels of the window. Assume prerequisite tasks have been completed (if they haven’t, the user will simply change windows.)
  - The first DPs were about User Interfaces!
Gang of four (GoF)

- OOPSLA workshop 1991:
  - Coincidentally, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides were all there; they would later become the Gang of Four that authored the Design Patterns book.

Definition

- A solution of a defined problem in a certain context
  - Context is a set of recurring situations
  - Problems are goals and limits in this context.
  - A solution describes the steps needed for reaching your goal.

Not everything is a DP

- Problem
  - How do I get back my winning lottery ticket before it's too late?
- Context
  - The dog ate my ticket.
- Solution
  - Dissect the dog.
Why use DP?

- Reuse design. More powerful than to reuse code.
- A common language. High level meaning.
  - We can actually use DPs as words
  - We do that by referring to things as "cache" or "proxy" – these are DPs.
- Increased understanding of solutions.

Iterator

Goal: to be able to traverse all objects in a set without knowing about how they are stored.
public interface Iterator
    boolean hasNext();
    object next();
    void remove();
};

HashSet hash = new HashSet();
hash.add("Hi"); hash.add("Ho");
LinkedList list = new LinkedList();
list.add("Hi"); list.add("Ho");

void print(Iterator it) {
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}

print(hash.iterator());
print(list.iterator());
Singleton

"a card that is the only one of its suit"

Singelton

- There can be only one instance of a certain class:
  - One database, one catalogue, one backup, one rot object, etc.
- As developers, we need to make sure this property holds true – by design.

```java
final class Singleton {
    private static Singleton s = new Singleton(47);
    private int i;
    private Singleton(int x) { i = x; }
    public static Singleton getReference() { return s; }
    public int getValue() { return i; }
    public void setValue(int x) { i = x; }
}
```
We need to be able to extend the number of objects, without knowing exactly with what/how many.

It is not enough to use "interface", because we then spread this knowledge to many places in the code.

Solution: to use a "factory" that will create objects for us.
- Should we need to add new objects, it is enough to change the factory.
Factory, solution

```java
abstract class Shape {
    public abstract void draw();
    public abstract void erase();
    public static Shape factory(String type) {
        if(type.equals("Circle")) return new Circle();
        if(type.equals("Square")) return new Square();
        throw new RuntimeException("Bad shape creation: "+type);
    }
}

class Circle extends Shape {
    Circle() {} // Friendly constructor
    public void draw() {
        System.out.println("Circle.draw");
    }
    public void erase() {
        System.out.println("Circle.erase");
    }
}

Shape shape = Shape.factory("Circle");
Shape.draw();
```

Factory, common use

```java
interface Crypto {
    // encode, decode, keys, etc...
}

Message msg = new Message("top secret string");
Crypto topSecret = OurSecureFactory.getCrypto("RSA");
Crypto notSoSecret = OurSecureFactory.getCrypto("Caesar");
Crypto std = OurSecureFactory.getCrypto();

send(topSecret.encode(msg));
send(notSoSecret.encode(msg));
send(std.encode(msg));
```
Command

- **Purpose**
  - Wrap commands in an object. Then we can pass commands around, to other objects.

- **Why**
  - We can stack commands until later, we can run them in a certain order, etc.
  - Easy to implement "undo".
Command

interface Command {
    void execute();
}

class Hello implements Command {
    public void execute() {
        System.out.print("Hello ");
    }
}

Command

void runAllCommands(Collection col) {
    Iterator it = col.iterator();
    while (it.hasNext()) {
        Object o = it.next();
        if (o instanceof Command) {
            ((Command)o).execute();
        }
    }
}

Command, common use

JButton bold = new JButton("Quit");
// command
ActionListener boldCmd = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        setBold();
        // save what we did so we may undo
        undo.put(this);
    }
};
bold.addActionListener(boldCmd);
Observer

Purpose

- Define a one-to-many relationship, so that when an object changes state, all dependent objects are automatically notified.

Why

- Our code consists of many independent parts. We want to keep them independent.
- Information that is shared should be consistent.
- Common use is in user interfaces; if a value changes, then the presentation of that value needs to be updated.
Applicability

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Delegation

- Inheritance is a static relation. Sometimes we need a more dynamic solution.
Inheritance a solution?

Delegation instead?

Delegation

```java
class Instrument {
    public void play() { System.out.println("Instrument play");
    }
}
class Guitar extends Instrument {
    public void play() { System.out.println("Guitar play");
    }
    --------
    interface Instrument {
        public void play();
    }
    class Guitar implements Instrument {
        public void play() { System.out.println("Guitar play");
    }
    // clear
    instrumentShop.add(new Instrument());
    public void play() {
        System.out.println("Oink.." cautionary)
    }
```
Decorator, solution

A Decorator adds responsibility to a component by wrapping it, but the Decorator conforms to the interface of the component it encloses, so the wrapping is transparent.

```
Decorator, solution

interface DrinkComponent {
    String getDescription();
    float getTotalCost();
}

abstract class Decorator implements DrinkComponent {
    protected DrinkComponent component;
    Decorator(DrinkComponent component) {
        this.component = component;
    }
    public float getTotalCost() {
        return component.getTotalCost();
    }
    public abstract String getDescription();
}

class Espresso extends Decorator {
    private float cost = 0.75f;
    private String description = "espresso";
    public Espresso(DrinkComponent component) {
        super(component);
    }
    public float getTotalCost() {
        return component.getTotalCost() + cost;
    }
    public String getDescription() {
        return component.getDescription() + description;
    }
}
```

Decorators in Java/Swing

```
public interface Border {
    Insets getBorderInsets(Component c)
    boolean isBorderOpaque()
    void paintBorder(Component c, Graphics g, int x, int y, int width, int height)
}
```