Design Patterns

+ Animation
  Undo/Redo
  Graphics and Hints
Design Patterns

• Design:
  – the planning
  – that lays the basis for the making
  – of every object
  – or system

• Pattern:
  – a type of theme
  – of recurring
  – events or objects
Design Patterns in OOP

• Recipes for solving standardized problems

• Abstractions of concrete problems
  – *How to make a class that allows only one instance of itself:*
    • The Singleton Design Pattern
    • Provides a general solution that needs to be implemented
Started from Architecture

• Architectural Patterns
  – Lecture rooms
  – Studies
  – Town Squares
  – Restaurants
  – Etc.

• Adopted in many non-related areas
Design patterns

• describe...
• how objects communicate...
• without becoming entangled...
• in each other’s data models...
• and/or methods.
What is a design pattern?

• A general recipe

• A framework

• A template

• Pre-made solution
Design Pattern Content

• Intent – what is the goal of the pattern?
• Applicability – when should it be used?
• Structure – the pattern description
• Consequences – pros and cons of using the pattern.
Good List of Patterns

**Creational patterns**
These patterns have to do with class instantiation. They can be further divided into class creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation to get the job done.
- **Abstract Factory** groups object factories that have a common theme.
- **Builder** constructs complex objects by separating construction and representation.
- **Factory Method** creates objects without specifying the exact class to create.
- **Prototype** creates objects by cloning an existing object.
- **Singleton** restricts object creation for a class to only one instance.
- **Multiton** restricts object creation for a class to only one instance per given key.

**Structural patterns**
These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.
- **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behaviour in an existing method of an object.
- **Facade** provides a simplified interface to a large body of code.
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

**Behavioral patterns**
Most of these design patterns are specifically concerned with communication between objects.
- **Chain of responsibility** delegates commands to a chain of processing objects.
- **Command** creates objects which encapsulate actions and parameters.
- **Interpreter** implements a specialized language.
- **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento** provides the ability to restore an object to its previous state (undo).
- **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **State** allows an object to alter its behavior when its internal state changes.
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

The "Gang of Four":
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
*Design Patterns: Elements of Reusable Object-Oriented Software*
Example

• The "Singleton pattern"
• Ensures that one and only one instance can be created from a class

```java
private static Singleton m_instance;
private Singleton() { ... }    // private constructor

public static synchronized Singleton getInstance()
{
    if (m_instance == null)    // only create new first time
        m_instance = new Singleton();

    return m_instance;    // return the single instance
}
```
Design Patterns vs. Libraries

• Libraries contain concrete classes
  – Implement concrete solutions

• Design Patterns are abstract solutions
  – Advice on general solutions

• You can’t import a design pattern
DP Factory

• Another interesting Design Pattern

• What is a factory?
  – Production unit
    • produces instances
    • multiple instances of different classes
  – Ask for a type and you get it!
Factory Example

• A graphic Shape
  – circles, rectangles, stars, etc.
  – All shapes have size, and can be drawn!

• The Shape interface defines methods

• Use a ShapeFactory:
  
  public Shape getShape(String type, int size) {...}
ShapeFactory, Why?

• One class provides all shapes!

• Dynamic creation!

• Consistent usage of Shapes
  – Unless explicitly needed, we don’t have to know which Shape we have created afterwards
public interface ImageReader {
    public DecodedImage getDecodedImage();
}

class GifReader implements ImageReader {
    public DecodedImage getDecodedImage() {
        // ... return decodedImage;
    }
}

class JpegReader implements ImageReader {
    public DecodedImage getDecodedImage() {
        // ... return decodedImage;
    }
}
public class ImageReaderFactory {
    public static ImageReader getImageReader(InputStream is) {
        int imageType = determineImageType(is);
        switch(imageType) {
            case ImageReaderFactory.GIF: return new GifReader(is);
            case ImageReaderFactory.JPEG: return new JpegReader(is);
            // etc. }
    }
}
UNDO

• The Undo/Redo mechanism should provide:
  – Single Undo
  – Single Redo (of Undone activity)

• and, if possible:
  – Undo and redo of several consecutive actions

• How can this be achieved?
Command Design Pattern

• One way to handle queues of actions!

• Important DP for the operation of UNDO

• Simple principle
Command

• A very simple interface!
  – One method
    execute()

```java
public interface Command {
    public void execute();
}
```
Command

• All activities are driven by the `execute()` method
  – Encapsulate all user activities in "Command objects"
Simple example: Macro

```java
public class IAm implements Command {
    public void execute() {
        System.out.println("I'm the command pattern!");
    }
}
// An object that holds commands:
public class Macro {
    private List commands = new ArrayList();

    public void add(Command c) {
        commands.add(c);
    }

    public void run() {
        Iterator it = commands.iterator();
        while(it.hasNext()) {
            ((Command) it.next()).execute(); // Casting!
        }
    }
}
```
UNDO?

• The execute() method describes what happens during an edit activity...
• UNDO means reversing the editing activity...
• In the command interface add:
  – unexecute()
  – reexecute()
  – Describes the backward process

```java
public interface Command {
    public void execute();
    public void unexecute(); // undo
    public void reexecute(); // redo
}
```
UndoManager

• UndoManager is support for UNDO in Swing

• "edit types" - the effect of a user invoked command
  – each edit type must still have a definition of the edit and its effects

• Simplifies the management of the Undo/Redo
Swing Undo

- UndoableEdit (Interface)
- AbstractUndoableEdit (Abstract class)
- CompoundEdit (class for sequences of undoables)
- UndoableEditListener (Interface)
- UndoableEditEvent (notification object)
- UndoableManager (Queue manager)
  - Like EventManager
- UndoableEditSupport (Support class)
Undo/Redo-queue

UndoManager

op1  op2  op3  op4
Undo/Redo-queue

UndoManager

op1  op2  op3  op4

op1  op2  op3  op4
Undo/Redo-queue

UndoManager

op1  op2  op3  op4

op1  op2  op3  op4

op1  op2  op5
Undo

• Good example program in linked page on course homepage (Assignment 2)

Add an undo/redo function to your Java apps with Swing
Don’ts in GUIs

• We will be able to do almost anything in GUI design

• But some things maybe we shouldn’t

• (but no rule without exception, of course)
Move things for the user

• You **can** control the mouse pointer for the user
  – java.awt.Robot library
    • java.awt.Robot
    • mouseMove(int x, int y);

• Should you?
  – Most of the time – NO!
  – Bad GUI!

• One possible exception: Guiding help systems
The Robot is still Useful!

Can make screenCaptures!

createScreenCapture(Rectangle screenRect)

returns a BufferedImage
Remove Title Bar of Window

• You can remove the Titlebar of a window

• Should you?
  – Maybe, e.g. in AboutBoxes

• But remember to provide the user with control of the window!
Use Animations

• You can use animations

• Should you?
  – Yes, sometimes
  – But restrictively

• Beware of perception exhaustion
Use Undo

• You will be able to Undo things

• Should you?
  – Yes, but only significant changes
  – Don’t undo single character insertion

• What is significant?
Use graphic effects

• Swing allows you to use graphic effects
  – E.g. GradientPaint, Transparency

• Should you?

• Yes, but always provide an alternative skin that is clear and distinct
  – Make it easy to access

• Make sure to test that the effects work on the platform
Be Creative

• It is (in principle) your imagination (and available time) that sets the limits for what you can do.

• Should you?
  – Yes, as long as it is good User Interface Design
  – Yes, as long as it is possible to understand
  – Yes, as long as it makes sense for the intended user groups
Practice Animation

• You have a large toolbox with graphics! Practice using it!

• Should you?

• Yes, even if it not all will be used in the Calendar, the more you practice, the more you will know about how and when you should use it!