

## 'Gang of Four' (GoF) Design Patterns

### An introduction

1

## Why do we want them?

### ■ New emphasis in software engineering

- Used to think the process was to write down requirements very carefully then proceed in careful steps to achieve them with code
- In the above view, functional decomposition (top-down or bottom-up structural design) is a good way to proceed
- Now we have come to understand that clear, fixed requirements are rare; requirements usually change *during* development and lifetime of code

2

## Process and Product issues

### ■ Process

- As discussed by Larman, new processes of development more suited to evolving requirements, have been proposed: the Unified Process, XP

### ■ Product

- But process does not really address the *form* of a design; maybe some forms are more adaptable than others, and hence better for evolving requirements?...
- What would 'better' mean?
  - Changed requirements can always be accommodated (code again from scratch). But some way to minimise the changes to code would be best, where minimise does not just refer to the amount of code change. Changes might be large, but straightforward to make, following a known procedure.

3

## OOD

### ■ OOD involves new approaches to both process and product

- The two are interrelated and support each other
- This unit is more concerned with product (the *form* of a design)...
- OOD goes beyond functional decomposition (OOD still uses functional decomposition in the sense that any requirement must still ultimately be satisfied by a sequence of routine calls, but approaches it in a specialised way)
- if we can predict potential future changes, we can use forms of OOD that allow these changes to be accommodated more easily in the future

4

## Product-based OOD

### ■ Need guidelines that control the design forms we come up with, so that they can cope with (some types of) future change

- "Use functional decomposition" is a guideline, but it is too general, it does not achieve what we want
- "Use data-driven design" (quite old idea: ADTs) - tying data closely to a limited set of operations that can be performed on it. Nice idea, that aids maintenance by avoiding side-effects, but hardly solves our problem
- "Use inheritance and polymorphism" is a newer guideline - does directly address our aims, but the latest guidance from the DPs literature says "its often better to prefer aggregation to inheritance"...why?...
- hard to describe but latest ideas say something like... "use responsibility-driven design, plus a whole collection of specific design guides called design patterns" (

5

## Responsibility-driven design

### ■ Motivate with an example

- adapted from "*Design Patterns Explained: a new perspective on object-oriented design*" 2002 (details in unit web materials)
- suppose it is a lecturer's job to make sure everyone in the lecture theatre gets to their next lecture
- functional design might break the task down...
  - get the list of all students
  - for each student:
    - [find their next lecture and the location of that class;
    - find the route to that lecture;
    - lecturer tells the person what they have to do]

6

### Example (cont.)

- Not as bad as it sounds since most students will fall into groups - there will not be that many different lectures to go to (could just read out the different routes, expecting students to know which room they needed to get to)
- An alternative solution
  - email the unit mailing list to tell students they must find out the room number for their next lecture
  - post a map(s) somewhere
  - At end of lecture, tell students where the map is, and to use it
- Main difference is transfer of responsibility to students

7

### Example (cont.)

- Now change requirements
  - the Law department decides they want some of their students to attend the lecturer's unit
  - in the first solution this means extra tasks for the lecturer
  - in the second solution, the lecturer's task don't change
  - in both cases there must be a map+other instructions somewhere, but in the second case there is immunity to change in part of the system i.e. the lecturer's algorithm. The sets of students are a '*potential source of future variation.*'

8

### Design Patterns

- Inspired by ideas from architectural design (buildings, spaces)
  - Christopher Alexander - can read about this in "Design Patterns Explained" or most books on DPs, but can understand the ideas without looking into their roots
- What are they?
  - "Design patterns are recurring solutions to design problems you see over and over" *The Design Patterns Smalltalk Companion*
  - Patterns identify and specify abstractions that are above the level of single classes and instances, or of components" *Design Patterns - Elements of Reusable Software [GoF]*

9

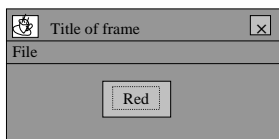
### Reasons for using them

- Standard one
  - Reusing designs, not just modules, classes, components. Same benefits of traditional re-use: benefit from experience of others, not always reinventing the wheel
- Less obvious one
  - Can design at a high level of abstraction, without having to pin-down low level details
  - This is often the best way to go (top down) - "get the architecture right and the rest falls into place more easily" (get it wrong and you will always be trying to fit square pegs in round holes)

10

### Eg1: The command pattern

- Example: design of simple Java GUI code
  - Simple window with a 'File' menu with two items 'Open' (allows user to open a file) and 'Exit' (allows user to terminate the program), and a button labelled 'Red' that turns the window's background colour to red
  - Example from 'Java Design Patterns' J.W.Cooper (Addison-Wesley)



11

### Design 1

```
//base example without any command pattern
import java.awt.*;
import java.awt.event.*;

public class noCmd extends Frame
    implements ActionListener
{
    Menu mnuFile;
    MenuItem mnuOpen, mnuExit;
    Button btnRed;
    Panel p;
    //-----
```

```

public noCmd()
{
    super("Frame without commands");
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    mnuFile = new Menu("File", true);
    mbar.add(mnuFile);

    mnuOpen = new MenuItem("Open...");
    mnuFile.add(mnuOpen);
    mnuExit = new MenuItem("Exit");
    mnuFile.add(mnuExit);

    mnuOpen.addActionListener(this);
    mnuExit.addActionListener(this);
}

```

```

    btnRed = new Button("Red");
    p = new Panel();
    add(p);
    p.add(btnRed);

    btnRed.addActionListener(this);
    setBounds(100,100,200,100);
    setVisible(true);
}
//-----

```

```

public void actionPerformed(ActionEvent e)
{
    Object obj = e.getSource();
    if(obj == mnuOpen)
        fileOpen();
    if (obj == mnuExit)
        exitClicked();
    if (obj == btnRed)
        redClicked();
}
//-----
private void exitClicked()
{
    System.exit(0);
}
//-----

```

```

private void fileOpen()
{
    FileDialog fDlg = new FileDialog(this, "Open a file", FileDialog.LOAD);
    fDlg.show();
}
//-----
private void redClicked()
{
    p.setBackground(Color.red);
}
//-----
static public void main(String argv[])
{
    new noCmd();
}
}

```

## Java GUIs in one slide

- For those unfamiliar with Java GUIs...
- GUI classes (for e.g. buttons, menu items, frames) already exist ready for you to use – a program just calls the APIs of this code. We are not concerned with the design of that code, we just use it.
- GUI objects contain (keep references to) other GUI objects – there are various rules for ‘what contains what.’ Again, you can just accept the code does this properly, but make sure you can work out where each GUI object is in relation to the others.
- Another ‘given’ is Java’s *event handling*. Think of GUI objects as running concurrently with each other i.e. they can react (execute methods) at times not specified in your code. E.g. a button recognises when a mouse press has occurred over it and sends an `actionPerformed(event_description_object)` message to tell other objects what has happened. The objects it sends to are those which ‘are interested,’ which means they have been added to the a list of objects kept by the button (see `addActionListener()` method in code)

## Design choices

- The event description object can be used to find out which GUI object sent the `actionPerformed` message, so there are choices like:
  - Which objects should be informed of mouse events?
  - Which objects should decide on the action to perform, given a mouse event?
  - Which objects should be responsible for actually performing the action?
- In this design all the above happen in the single `:noCmd` object created in `main()`

## Design 2

```
import java.awt.*;
import java.awt.event.*;
//creates separate inner class Command
objects
//for each item: all as ActionListeners
public class actionCommand extends
Frame
{
Menu mnuFile;
MenuItem mnuOpen, mnuExit;
Button btnRed;
Panel p;
Frame fr;
//-----
public actionCommand()
{
super("Frame without commands");
fr = this; //save copy of this
frame
MenuBar mbar = new MenuBar();
setMenuBar(mbar);
mnuFile = new Menu("File", true);
mbar.add(mnuFile);

mnuOpen = new
MenuItem("Open...");
mnuFile.add(mnuOpen);
mnuExit = new MenuItem("Exit");
mnuFile.add(mnuExit);

mnuOpen.addActionListener(new
fileOpen());
mnuExit.addActionListener(new
fileExit());

btnRed = new Button("Red");
p = new Panel();
add(p);
p.add(btnRed);

btnRed.addActionListener(new
btnRed());
setBounds(100,100,200,100);
setVisible(true);
}
```

```
//-----
private void exitClicked()
{
System.exit(0);
}
//-----
static public void main(String argv[])
{
new actionCommand();
}
//=====inner classes-----
class fileOpen implements ActionListener
{
public void
actionPerformed(ActionEvent e)
{
FileDialog fdlg = new FileDialog(fr,
"Open a file", FileDialog.LOAD);
fdlg.show();
}
}
//-----
class btnRed implements ActionListener
{
public void
actionPerformed(ActionEvent e)
{
p.setBackground(Color.red);
}
}
//-----
class fileExit implements
ActionListener
{
public void
actionPerformed(ActionEvent e)
{
System.exit(0);
}
}
//=====
```

## Design 2 comments

- Classic traditional Java solution
- Menu options and Buttons call actionPerformed() in special 'pure fabrication' classes that wrap the action code
- Happen to be inner classes but don't have to be – if havent met inner classes yet just think of them as normal classes (one main difference is because of their position they have automatic access to all the members of the enclosing class, even private ones)
- Instances of these inner classes can be created and references to them given to the buttons/menu items
- Action code is separated out to some extent from main Frame object (:actionCommand)

21

## Design 3

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

MenuBar mbar = new MenuBar();
setMenuBar(mbar);

mnuFile = new Menu("File", true);
mbar.add(mnuFile);
mnuOpen = new
MenuItem("Open...");
mnuFile.add(mnuOpen);
mnuExit = new MenuItem("Exit");
mnuFile.add(mnuExit);

mnuOpen.addActionListener(new
ActionListener()
{
public void
actionPerformed(ActionEvent e)
{
FileDialog fdlg = new
FileDialog(fr, "Open a file",
FileDialog.LOAD);
fdlg.show();
}
});

//here the ActionListeners are
implemented as unnamed inner
classes
public class innerCommand extends Frame
{
Menu mnuFile;
MenuItem mnuOpen, mnuExit;
Button btnRed;
Panel p;
Frame fr;
//-----
public innerCommand()
{
super("Frame without commands");
fr = this;

mnuOpen.addActionListener(new
ActionListener()
{
public void
actionPerformed(ActionEvent e)
{
System.exit(0);
}
});

btnRed = new Button("Red");
p = new Panel();
add(p);
p.add(btnRed);

btnRed.addActionListener(new
ActionListener()
{
public void
actionPerformed(ActionEvent e)
{
p.setBackground(Color.red);
}
});

setBounds(100,100,200,100);
setVisible(true);
}
```

## Design 3 comments

- Same as 2 except using anonymous inner classes
- No extra design interest (+ saves a few lines of code, -harder to read)

24

```
mnuExit.addActionListener(new ActionListener()
{
public void
actionPerformed(ActionEvent e)
{
System.exit(0);
}
});

btnRed = new Button("Red");
p = new Panel();
add(p);
p.add(btnRed);

btnRed.addActionListener(new
ActionListener()
{
public void
actionPerformed(ActionEvent e)
{
p.setBackground(Color.red);
}
});

setBounds(100,100,200,100);
setVisible(true);
}

//-----
static public void main(String argv[])
{
new innerCommand();
}
//=====
```

## Design 4

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

//Implements separate inner Command
classes as extensions of
//Button and Menu items

public class tCommand extends Frame
implements ActionListener
{
    Menu mnuFile;
    fileOpenCommand mnuOpen;
    fileExitCommand mnuExit;
    btnRedCommand btnRed;
    Panel p;
    Frame fr;
    //-----
```

```
public tCommand()
{
    super("Frame with commands");
    fr = this; //save frame object
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    mnuFile = new Menu("File", true);
    mbar.add(mnuFile);

    mnuOpen = new
fileOpenCommand ("Open...");
    mnuFile.add(mnuOpen);
    mnuExit = new
fileExitCommand("Exit");
    mnuFile.add(mnuExit);

    mnuOpen.addActionListener(this);
    mnuExit.addActionListener(this);
}
```

```
btnRed = new btnRedCommand("Red"); //-----inner class-----
p = new Panel(); class btnRedCommand extends Button
add(p); implements Command
p.add(btnRed); {
    public btnRedCommand(String
caption)
    {
        super(caption);
    }
    public void Execute()
    {
        p.setBackground(Color.red);
    }
}
btnRed.addActionListener(this);
setBounds(100,100,200,100);
setVisible(true);
}
//-----
public void actionPerformed(ActionEvent
e)
{
    Command obj =
(Command)e.getSource();
    obj.Execute();
}
//-----
static public void main(String argv[])
{
    new tCommand();
}
```

```
class fileOpenCommand extends
MenuItem implements Command
{
    public fileOpenCommand(String
caption)
    {
        super(caption);
    }
    public void Execute()
    {
        FileDialog fDlg=new
FileDialog(fr,"Open file");
        fDlg.show();
    }
}
//-----
```

```
class fileExitCommand extends MenuItem
implements Command
{
    public fileExitCommand(String
caption)
    {
        super(caption);
    }
    public void Execute()
    {
        System.exit(0);
    }
}

//=====
public interface Command
{
    public void Execute();
}
```

## Design 4 comments

- First example of command objects: means implements the Command interface (a single method Execute() )
- Here these are subclasses of Buttons and MenuItems.
- actionPerformed() is back in the main Frame class, tCommand, rather than in the inner classes
- But actions themselves are in inner classes, in the form of Execute() methods
- Bit strange – e.g. :Button tells frame about a mouse press, :Frame then calls it back to tell it to do something

28

## Design 5

```
import java.awt.*;
import java.awt.event.*;
//In this version, the Command objects
are external classes
//and we pass them copies of the Frame
instance
//In their constructor
public class extrnCommand extends
Frame
implements ActionListener
{
    Menu mnuFile;
    fileOpenCommand mnuOpen;
    fileExitCommand mnuExit;
    btnRedCommand btnRed;
    Panel p;
    Frame fr;
    //-----
```

```
public extrnCommand()
{
    super("Frame with external
commands");
    fr = this; //save frame object
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    mnuFile = new Menu("File", true);
    mbar.add(mnuFile);

    mnuOpen = new
fileOpenCommand ("Open...", this);
    mnuFile.add(mnuOpen);
    mnuExit = new
fileExitCommand("Exit");
    mnuFile.add(mnuExit);

    mnuOpen.addActionListener(this);
    mnuExit.addActionListener(this);
}
```

```
p = new Panel(); //-----
add(p); static public void main(String argv[])
btnRed = new {
    new extrnCommand();
}
p.add(btnRed); //=====
class btnRedCommand extends Button
implements Command
{
    Panel p;
    public btnRedCommand(String
caption, Panel pnl)
    {
        super(caption);
        p = pnl;
    }
    public void Execute()
    {
        p.setBackground(Color.red);
    }
}

btnRed.addActionListener(this);
setBounds(100,100,200,100);
setVisible(true);
}
//-----
public void actionPerformed(ActionEvent
e)
{
    Command obj =
(Command)e.getSource();
    obj.Execute();
}
```

```
//-----
class fileOpenCommand extends
MenuItem implements Command
{
    Frame fr;
    public fileOpenCommand(String
caption, Frame frm)
    {
        super(caption);
        fr = frm;
    }
    public void Execute()
    {
        FileDialog fDlg=new
FileDialog(fr,"Open file");
        fDlg.show();
    }
}

//-----
class fileExitCommand extends
MenuItem implements Command
{
    public fileExitCommand(String
caption)
    {
        super(caption);
    }
    public void Execute()
    {
        System.exit(0);
    }
}

//=====
public interface Command
{
    public void Execute();
}
```

## Design 5 Comments

- Same as design 4 except the Button/MenuItem classes are not inner classes
- Still has the ping-pong nature but there is now more sense in doing this – if you need to change the action associated with a button, you don't have to touch the main Frame class (extrnCommand). Could keep command classes in separate files.
- But, you are still altering GUI code. If you want to completely divorce the actions from the GUI you do something different...

32

## Design 6

```
//file: fullCommand.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

//In this version, we fully decouple the
Commands from
//the men and Button subclasses.
// the Command objects are external
classes

public class fullCommand extends JFrame
implements ActionListener
{
    JMenu mnuFile;
    cmdMenu mnuOpen, mnuExit;
    cmdButton btnRed;

    JPanel jp;
    JFrame fr;
    fileCommand flc;
    ExitCommand extc;
    RedCommand redc;

    //-----
    public fullCommand()
    {
        super("Frame with external
commands");
        fr = this; //save frame object
        JPanel jp = new JPanel();
        getContentPane().add(jp);
        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);

        mnuFile = new JMenu("File", true);
        mbar.add(mnuFile);
    }
}
```

```
mnuOpen = new cmdMenu("Open...",
this);
mnuFile.add(mnuOpen);

mnuOpen.setCommand(new
fileCommand(this));
mnuExit = new cmdMenu("Exit",
this);
mnuExit.setCommand(new
ExitCommand());

mnuFile.add(mnuExit);

mnuOpen.addActionListener(this);
mnuExit.addActionListener(this);

btnRed = new cmdButton("Red",
this);
redc = new RedCommand(this, jp);
btnRed.setCommand(redc);

jp.add(btnRed);

btnRed.addActionListener(this);
setBounds(100,100,200,100);
setVisible(true);

public void actionPerformed(ActionEvent
e) {
    CommandHolder obj =
(CommandHolder)e.getSource();
    obj.getCommand().Execute();
}

//-----
static public void main(String argv[])
{
    new fullCommand();
}
}
```

```
//file: JxFrame.java

import java.awt.*;
import java.awt.event.*;
import java.util.*;

//swing classes
import javax.swing.text.*;
import javax.swing.*;
import javax.swing.event.*;

public class JxFrame extends JFrame
{
    public JxFrame(String title)
    {
        super(title);
        setCloseClick();
        setLaf();
    }

    private void setCloseClick()
    {
        //create window listener to respond to
window close click
addWindowListener(new
WindowAdapter()
    {
        public void
windowClosing(WindowEvent e)
        {System.exit(0);}
    });
    }
}

//continued....
```

```
//-----
private void setLaf()
{
    // Force Swing App to come up in the
System L&F
String laf =
UIManager.getSystemLookAndFeelClas
sName();
try {
    UIManager.setLookAndFeel(laf);
}
}
catch
(UnsupportedLookAndFeelException
exc)
{System.err.println("Warning:
UnsupportedLookAndFeel: " + laf);}
catch (Exception exc)
{System.err.println("Error loading " +
laf + ": " + exc);
}
}
```

```
//file: cmdButton.java
import java.awt.*;
import javax.swing.*;

public class cmdButton extends JButton
implements CommandHolder {
    private Command btnCommand;
    private JFrame frame;

    public cmdButton(String name, JFrame fr) {
        super(name);
        frame = fr;
    }
    public void setCommand(Command cmd) {
        btnCommand = cmd;
    }
    public Command getCommand() {
        return btnCommand;
    }
}
```

```
//file: cmdMenu.java
import java.awt.*;
import javax.swing.*;

public class cmdMenu extends JMenuItem
implements CommandHolder {
    protected Command menuCommand;
    protected JFrame frame;
    //-----
    public cmdMenu(String name, JFrame
    frm) {
        super(name);
        frame = frm;
    }
    //-----
    public void setCommand(Command
    cmd) {
        menuCommand = cmd;
    }
    //-----
    public Command getCommand() {
        return menuCommand;
    }
}
```

```
//file: command.java
public interface Command
{
    public void Execute();
}

//file: commandHolder.java
public interface CommandHolder {
    public void setCommand (Command
    cmd);
    public Command getCommand();
}
```

```
//file: exitCommand.java
public class ExtCommand implements
Command {
    public void Execute () {
        System.exit(0);
    }
}

//file: fileCommand.java
import java.awt.*;
import javax.swing.*;

public class fileCommand implements
Command {
    JFrame frame;

    public fileCommand(JFrame fr) {
        frame = fr;
    }
    //-----
    public void Execute() {
        FileDialog fdlg = new
        FileDialog (frame, "Open file");
        fdlg.show();
    }
}
```

```
//file: redCommand.java
import java.awt.*;
import javax.swing.*;

public class RedCommand implements Command {
    private JFrame frame;
    private JPanel pnl;

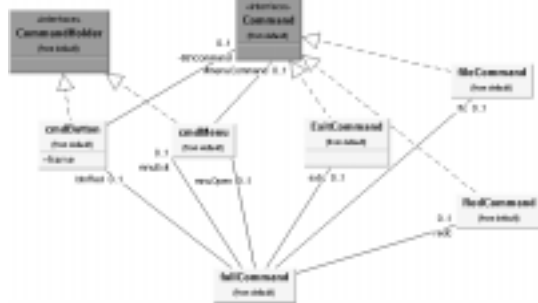
    public RedCommand(JFrame fr, JPanel p) {
        frame = fr;
        pnl = p;
    }
    public void Execute() {
        pnl.setBackground(Color.red);
    }
}
```

### Design 6 comments

- Cosmetic difference – uses Swing API instead of AWT. Very similar. Also has some extra functionality: JxFrame file adds code to close the window when 'X' clicked and sets the look and feel to the operating system the program is running under (all irrelevant to our design discussion)
- E.g. :cmdButton GUI object now *contains* a :Command object (this is implemented by giving it the commandHolder interface) rather than *being* a :Command object (implementing the Command interface)
- The action code is separated out completely from the GUI code. No need to touch the GUI package to alter the action associated with a button/menuItem.
- Small point: our MenuItem objects are now the same class – what separated them before was only the associated action

42

### UML class diagram (for 6)



■ this class diagram only contains the features necessary to explain how the command pattern is used in this example

■ lab exercise:

- reverse engineer the code (use any one of designs 1-6) to get a class diagram
- experiment by including as much detail as you can (to add standard AWT classes, you can use shortcuts – read ControlCenter online help and/or manual)
- reverse engineer the main() method to get a sequence diagram (SD)
- reverse engineer another method to get a SD to show what happens when the mouse is pressed over a button or menu item

■ Notice how the final design 6 makes it so easy to change the actions associated with a GUI object, it doesn't really matter what the actions are. The command pattern would allow you to build a design without knowing what the actual required actions will be.

### ■ Why study code? Why not UML?

- Could have explained this all in UML e.g. try reverse engineering the main() and actionPerformed() methods into a sequence diagram in ControlCenter... curiously unhelpful
- To get the idea of some patterns it is better to see complete details initially
- However, UML works best when it is not used to describe everything that is going on in the code - i.e. relying on the reader having *some* knowledge. This is particularly true with patterns. Having seen this example, in future you will look at a UML class diagram of a command pattern and understand immediately what the designer is trying to do.

### A new design language

■ Design patterns give you a new 'dictionary' of software design

- e.g. from *DPs Explained*
- A carpenter doesn't ask a colleague to "make the joint by cutting straight down, then back up at an angle of 45 degrees, then straight back down again, followed by going back up the other way at 45 degrees, and then straight down again, and then...etc."



- They would say "use a dovetail joint"