Principles of Software Construction: Objects, Design and Concurrency

Design Case Study: GUI with Swing

Christian Kästner    Charlie Garrod

© 2012-14 C Garrod, C Kästner, J Aldrich, and W Scherlis
Design Goals - Summary

• 5 design goals
  ▪ Design for division of labor
  ▪ Design for understandability and maintenance
  ▪ Design for change
  ▪ Design for reuse
  ▪ Design for robustness

• 5 design strategies
  ▪ Explicit interfaces (clear boundaries)
  ▪ Information hiding (hide likely changes)
  ▪ Low coupling (reduce dependencies)
  ▪ High cohesion (one purpose per class)
  ▪ Low repr. gap (align requirements and impl.)

• 3 GRASP patterns
  ▪ Information Expert
  ▪ Creator
  ▪ Controller
Learning Goals

• Understanding event-based programming

• Understanding design patterns in GUIs
  ▪ Strategy pattern
  ▪ Observer pattern
  ▪ Composite pattern
  ▪ Decorator pattern
  ▪ Template method pattern
  ▪ Façade pattern
  ▪ Model-view-controller pattern
Event-based programming

- Style of programming where control-flow is driven by (usually external) events

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```java
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```
Events in GUIs

• User clicks a button, presses a key
• User selects an item from a list, an item from a menu, expands a tree
• Mouse hovers over a widget, focus changes
• Scrolling, mouse wheel turned
• Resizing a window, hiding a window
• Drag and drop

• A package arrives from a web service, connection drops, ...
• System shutdown, ...
Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
    Question q = question.next();
    System.out.println(q.toString());
    String answer = input.nextLine();
    q.respond(answer);
}
while (true) {
    if (isKeyDown("Alt+Q"))
        break;
    if (isKeyDown("F1"))
        openHelp();
    if (isMouseDown(10 ...))
        startMovingWindow();
    ...}

You have a royalty-free right to use, modify, reproduce, and distribute the sample applications and toolkits in any way you find useful, provided that you agree Microsoft has no warranty, obligations or liability any of the sample applications or toolkits.
Event-based GUIs

//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email = 
    emailField.getText();
emaillist.add(email);
//static public void main...  
JFrame window = ...  
window setDefaultCloseOperation(  
    WindowConstants.EXIT_ON_CLOSE);  
window.setVisible(true);

//on add-button click:  
String email = emailField.getText();  
emaillist.add(email);

//on delete-button click:  
int pos = emaillist.getSelectedItem();  
if (pos>=0) emaillist.delete(pos);
(Blocking) Interactions with Users

Game

Dealer

Player

newGame

addCards

addCards

getAction

action

[action==hit] addCard

blocking execution
Interactions with Users through Events

- Do not wait for user response, react to event
- Here: Two interactions to separate events:

```
Game
  \_ newGame
  \_ addCards

Dealer
  \_ addCards

Player
  \_ addCard

hit
```
Event-Based GUI Model

- The program has no main loop! Continues executing even after main method ends
- Program is idle after start until called for events
- The operating system / GUI framework processes all keyboard/mouse/... events
- Only few events are relevant for application
- Every widget may react to its own events
- Program(widgets) may register callbacks to react to specific events
- Program ends by calling specific API, not when the main method finishes executing
Reacting to events from GUI framework

- **Setup phase**
  - Describe how the GUI window should look
  - Use libraries for windows, widgets, and layout
  - Embed specialized code for later use
  - Register callbacks

- **Execution**
  - Framework gets events from OS
    - Raw events: mouse clicks, key presses, window becomes visible, etc.
  - Framework processes events
    - Click at 10,40: which widget?
    - Resize window: what to re-layout and redraw?
  - Triggers callback functions of corresponding widgets (if registered)
Example: RabbitWorld GUI

- ...hw2.lib.ui.WorldUI.main()
  - Creates a top-level window
  - Creates a WorldUI to go in it
  - Sets some parameters
  - Makes the window (and its contents) visible

- ...hw2.lib.ui.WorldPanel.paintComponent()
  - Called when the OS needs to show the WorldPanel (part of WorldUI)
    - Right after the window becomes visible
      - super.paintComponent() draws a background
      - ImageIcon.paintIcon(...) draws each item in the world

Let’s look at the code...
GUI Frameworks in Java

• AWT
  ▪ Native widgets, only basic components, dated

• Swing
  ▪ Java rendering, rich components

• SWT + JFace
  ▪ Mixture of native widgets and Java rendering; created for Eclipse for faster performance

• Others
  ▪ Apache Pivot, SwingX, JavaFX, ...

• Different in their specific designs, but similar overall strategies and concepts
Swing

JFrame

JPanel

JButton

JTextField

...
Swing has lots of widgets

- JLabel
- JButton
- JCheckBox
- JChoice
- JRadioButton
- JTextField
- JTextArea
- JList
- JScrollBar
- ... and more

- JFrame is the Swing Window
- JPanel (aka a pane) is the container to which you add your components (or other containers)
To create a simple Swing application

• Make a Window (a JFrame)
• Make a container (a JPanel)
  ▪ Put it in the window
• Add components (Buttons, Boxes, etc.) to the container
  ▪ Use layouts to control positioning
  ▪ Set up observers (a.k.a. listeners) to respond to events
  ▪ Optionally, write custom widgets with application-specific display logic
• Set up the window to display the container
• Then wait for events to arrive...
Reacting to Events
/ **Creating a Button**

```java
//static public void main...
JFrame window = ...

JPanel panel = new JPanel();
window.setContentPane(panel);

JButton button = new JButton("Click me");
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      System.out.println("Button clicked");
   }
});
panel.add(button);

window.setVisible(true);
```

- **Panel to hold the button**
- **Callback function**
- **Register callback function**
- **Callback function implements ActionListener interface**
Action Listeners

- Listeners are objects with callback functions
- Listeners can be registered with widgets
- Widgets call all registered widgets if event occurs

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

```java
class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
    int id;
    ...
}
```
**Action Listeners**

- Listeners are objects with callback functions
- Listeners can be registered with widgets
- Widgets call all registered widgets if event occurs

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}

class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
    int id;
    ...
}

class AbstractButton extends JComponent {
    private List<ActionListener> listeners;
    public void addActionListener(ActionListener l) {
        listeners.add(l);
    }
    protected void fireActionPerformed(ActionEvent e) {
        for (ActionListener l: listeners)
            l.actionPerformed(e);
    }
}
class MyButton extends JButton {
    public MyButton() { super(“Click me”); } 
    @Override
    protected void fireActionPerformed(ActionEvent e) {
        super.fireActionPerformed(e);
        System.out.println(“Button clicked”);
    }
}

//static public void main...
JFrame window = ...
JPanel panel = new JPanel();
window.setContentPane(panel);
panel.add(new MyButton());
window.setVisible(true);
Design Discussion

• Button implementation should be reusable
  ▪ but differ in button label
  ▪ and differ in event handling
  ▪ multiple independent clients might be interested in observing events
  ▪ basic button cannot know what to do

• Decoupling action of button from button implementation itself

• Inheritance simple form to specialize buttons (second example)

• Listeners are separate objects, fulfilling an interface (first example)
  ▪ multiple listeners possible
  ▪ multiple buttons can share same listener
JButton btnNewButton = new JButton("New button");

btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        counter.inc();
    }
});

btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "button clicked", "alert",
                                 JOptionPane.ERROR_MESSAGE);
    }
});
class MyButton extends JButton {
    Counter counter;
    public MyButton() {
        ... setTitle...
    }
    protected void fireActionPerformed(ActionEvent e) {
        counter.inc();
    }
}

JButton btnNewButton = new MyButton();
panel.add(btnNewButton);
public class ActionListenerPanel extends JPanel {
    implements ActionListener {
    
    public ActionListenerPanel() { setup(); }
    
    private void setup() {
        button1 = new JButton("a");
        button1.addActionListener(this);
        button2 = new JButton("b");
        button2.addActionListener(this);
        add(button1); add(button2);
    }
    
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==button1)
            display.setText(BUTTON_1);
        else if(e.getSource()==button1) ...
    }
    ...
}

public class ActionListenerPanel extends JPanel implements ActionListener {
    public ActionListenerPanel() { setup(); }
    private void setup() {
        button1 = new JButton("a");
        button1.addActionListener(this);
        button2 = new JButton("b");
        button2.addActionListener(this);
        add(button1); add(button2);
    }
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==button1)
            display.setText(BUTTON_1);
        else if...
    }
}

Cohesion?

Class responsibilities include (1) building the display, (2) wiring buttons and listeners, (3) mapping events to proper response, (4) perform proper response

Consider separating out event handling with different listeners
The *Template Method* design pattern

```
<<stereotype>>
AbstractClass
{abstract }

+TemplateMethod(){final }
PrimitiveOperation(){abstract }

TemplateMethod()
{
    PrimitiveOperation();
    ...
    ...
}

<<stereotype>>
ConcreteClass

PrimitiveOperation()
```
The **Template Method** design pattern

- **Applicability**
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations

- **Consequences**
  - **Code reuse**
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses

You may have used this in your virtual world
Template Method in JButton

- JButton has some behavior of how to handle events
  - eg drawing the button pressed while mouse down

- Some behavior remains undefined until later -> abstract method
  - In this case, default implementation of fireActionEvent already exists

- Template method provides specific extension point within larger shared computation
The Observer design pattern

```
«stereotype»
Subject
{abstract}
+Attach(observer: Observer): void
+Detach(observer: Observer): void
+Notify()

observers

Observers
{interface}
+Update(): abstract

ConcreteSubject

ConcreteObserver
+Update()

subject: State
+GetState(): State
+SetState(state: State): void
```

For all o in observers, o.Update();
The **Observer** design pattern

- **Applicability**
  - When an abstraction has two aspects, one dependent on the other, and you want to reuse each
  - When 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 others without knowing who they are

- **Consequences**
  - **Loose coupling** between subject and observer, enhancing **reuse**
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect
Swing has lots of event listener interfaces:

- ActionListener
- AdjustmentListener
- FocusListener
- ItemListener
- KeyListener

- MouseListener
- TreeExpansionListener
- TextListener
- WindowListener

...and on and on...
Consider Virtual World:
- World stores all items
- GUI shows items
- GUI triggers new behavior in world
- When should the GUI update the picture?
• What if we add a warning field to the GUI that alerts if rabbits are about to die out?
What if we add a warning field to the GUI that alerts if rabbits are about to die out?

All control within the GUI. What if the World changes for other reasons not triggered by the GUI?
Observer Pattern between Data Model and GUI

- Alternative Design: Let the World tell the GUI if something happened

Inflexible to add new object to update (always requires change to the World)

Coupling from World to specific GUI implementations
Observer Pattern between Data Model and GUI

- Observer Design: Let the world tell anybody interested about updates

Observer pattern decouples core implementation from GUI. Explicit interfaces, lower coupling, better reuse, easier to change and extend, ...
The Observer design pattern

```
Subject
{abstract}
+Attach(observer: Observer): void
+Detach(observer: Observer): void
+Notify()

Observer
{interface}
+Update(): abstract

ConcreteSubject
+subjectState: State
+GetState(): State
+SetState(state: State): void

ConcreteObserver
+Update()
```

Notify()
```
Layout of Widgets
Swing Layout Manager

see http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
A naïve Implementation

• Hard-code layout algorithms

```java
class JPanel {
    protected void doLayout() {
        switch(getLayoutType()) {
            case BOX_LAYOUT: adjustSizeBox(); break;
            case BORDER_LAYOUT: adjustSizeBorder(); break;
            ...
        }
    }
    private adjustSizeBox() { … }
}
```

• A new layout requires changing or overriding JPanel
Layout Manager

• A panel has a list of children

• Different layouts possible
  ▪ List of predefined layout strategies
  ▪ Own layouts possible

• Every widget has preferred size

• Delegate specific layout to a separate class implementing an explicit interface
  ▪ Use polymorphism for extensibility
abstract class Container { // JPanel is a Container
    private LayoutManager layoutMgr;

    public doLayout() {
        LayoutManager m = this.layoutMgr;
        if (m!=null)
            m.layoutContainer(this);
    }

    public Component[] getComponents() { ... }
}

interface LayoutManager {
    void layoutContainer(Container c);
    Dimension getMinimumLayoutSize(Container c);
    Dimension getPreferredSize(Container c);
}
Behavioral: Strategy
Behavioral: Strategy

- **Applicability**
  - Many classes differ in only their behavior
  - Client needs different variants of an algorithm

- **Consequences**
  - Code is more **extensible** with new strategies
    - Compare to conditionals
  - Separates algorithm from context
    - each can vary independently
    - **design for change and reuse; reduce coupling**
  - Adds objects and dynamism
    - code harder to understand
  - Common strategy interface
    - may not be needed for all Strategy implementations – may be extra overhead
void sort(int[] list, String order) {
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    }
    ...
}

void sort(int[] list, Comparator cmp) {
    boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    ...
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int I, int j) { return i<j; }
}

class DownComparator implements Comparator {
    boolean compare(int I, int j) { return i>j; }
}
Design Goals

- **Design to explicit interfaces**
  - Strategy: the algorithm interface

- **Design for change and information hiding**
  - Find what varies and encapsulate it
  - Allows adding alternative variations later

- **Design for reuse**
  - Strategy class may be reused in different contexts
  - Context class may be reused even if existing strategies don’t fit

- **Low coupling**
  - Decouple context class from strategy implementation internals

- **Side note: how do you implement the Strategy pattern in functional languages?**
Template Method vs Strategy vs Observer

- **Template method vs strategy pattern**
  - both support variations in larger common context
  - Template method uses inheritance + abstract method
  - Strategy uses interface and polymorphism (object composition)
  - strategy objects reusable across multiple classes; multiple strategy objects per class possible
  - Why is Layout in Swing using the Strategy pattern?

- **Strategy vs observer pattern**
  - both use a callback mechanism
  - Observer pattern supports multiple observers (0..n)
  - Update method in observer triggers update; rarely returns result
  - Strategy pattern supports exactly one strategy or an optional one if null is acceptable (0..1)
  - Strategy method represents a computation; may return a result
The Strategy Pattern to Paint Borders

- `contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));`
  - Border interface has “paintBorder”, “getBorderInsets” and “isBorderOpaque” methods

```java
//alternative design
class JPanel {
    protected void paintBorder(Graphics g) {
        switch(getBorderType()) {
            case LINE_BORDER: paintLineBorder(g); break;
            case ETCHED_BORDER: paintEtchedBorder(g); break;
            case TITLED_BORDER: paintTitledBorder(g); break;
            ...
        }
    }
}
```
The Strategy Pattern to Paint Borders

- `contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));`
  - Border interface has “paintBorder”, “getBorderInsets” and “isBorderOpaque” methods

```java
//alternative design
class JPanel {
    protected void paintBorder(Graphics g) {
        switch(getBorderType()) {
            case LINE_BORDER: paintLineBorder(g); break;
            case ETCHED_BORDER: paintEtchedBorder(g); break;
            case TITLED_BORDER: paintTitledBorder(g); break;
            //...}
    }
}

//actual JComponent implementation
protected void paintBorder(Graphics g) {
    Border border = getBorder();
    if (border != null)
        border.paintBorder(this, g, 0, 0, getWidth(), getHeight());
```
Nesting Containers

BorderLayout()
GridLayout(1,0)
Panel 1 Panel 2
GridLayout(1,0)
Panel 3 Panel 4 Panel 5 Panel 6

OK
Reminder: Composite Design Pattern

- **Applicability**
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects

- **Consequences**
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components
Drawing Widgets
public void paint(Graphics g)

Invoked by Swing to draw components. Applications should not invoke paint directly, but should instead use the repaint method to schedule the component for redrawing.

This method actually delegates the work of painting to three protected methods: paintComponent, paintBorder, and paintChildren. They're called in the order listed to ensure that children appear on top of component itself. Generally speaking, the component and its children should not paint in the insets area allocated to the border. Subclasses can just override this method, as always. A subclass that just wants to specialize the UI (look and feel) delegate's paint method should just override paintComponent.

Overrides:
   paint in class Container

Parameters:
   g - the Graphics context in which to paint

See Also:
   paintComponent(java.awt.Graphics),
   paintBorder(java.awt.Graphics), paintChildren(java.awt.Graphics),
   getComponentGraphics(java.awt.Graphics), repaint(long, int, int,
   int, int)
Reminder: Template Method

- **Applicability**
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations

- **Consequences**
  - Code reuse
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses
GUI design issues

• **Interfaces vs. inheritance**
  - Inherit from JPanel with custom drawing functionality
  - Implement the ActionListener interface, register with button
  - Why this difference?

• **Models and views**
GUI design: Interfaces vs. inheritance

• Inherit from JPanel with custom drawing functionality
  ▪ Subclass “is a” special kind of Panel
  ▪ The subclass interacts closely with the JPanel – e.g. the subclass calls back with super
  ▪ Accesses protected functionality otherwise not exposed
  ▪ The way you draw the subclass doesn’t change as the program executes

• Implement the ActionListener interface, register with button
  ▪ The action to perform isn’t really a special kind of button; it’s just a way of reacting to the button. So it makes sense to be a separate object.
  ▪ The ActionListener is decoupled from the button. Once the listener is invoked, it doesn’t call anything on the Button anymore.
  ▪ We may want to change the action performed on a button press—so once again it makes sense for it to be a separate object

• See also Command pattern later
Separating Core and GUI
Core vs. GUI

• Core Implementation: Application logic
  - computing a result
  - updating some data

• GUI
  - Graphical representation of applications data
  - User interactions; starting point for actions

• Design Guideline: Separate core from GUI
  - Able to provide alternative GUIs
  - Able to test core separately
  - Keep GUI slim (no additional tests needed?)
  - Multithreading: keep GUI reactive also during long computations
  - -> Design for change, design for reuse, design for division of labor; low coupling, high cohesion
World

GUI

Warning

step

update(data)

update(data)

update

update(data)

update
Metapattern: Model-View-Controller (MVC)

- Manage inputs from user: mouse, keyboard, menu, etc.
- Manage display of information on the screen
- Manage data related to the application domain
Model-View-Controller (MVC)

Passive model

Active model

Separating Application Core and GUI

- Reduce coupling

- Create core of the application working and testable without a GUI
  - Use Observer pattern to communicate information from Core (Model) to GUI (View)
  - Use Controller (Façade) to perform operations on core
  - May run in separate threads (worker thread vs GUI thread) to avoid blocking, see SwingWorker

![Diagram of core and GUI separation]

GUI Tests → Core

GUI

Core Tests → Core

GUI
The Façade Design Pattern

- See GRASP Controller
- Applicability
  - You want to provide a simple interface to a complex subsystem
  - You want to decouple clients from the implementation of a subsystem
  - You want to layer your subsystems
- Consequences
  - It shields clients from the complexity of the subsystem, making it easier to use
  - **Decouples** the subsystem and its clients, making each easier to change
  - Clients that need to can still access subsystem classes
  - -> **Explicit interfaces, low coupling, information hiding, design for reuse, design for understandability, ...**
Filling Lists and Tables
JList and JTree

- Lists and trees highly flexible (reusable)
- Can change rendering of cells
- Can change source of data to display

```java
//simple use
String[] items = { "a", "b", "c" };
JList list = new JList(items);
```
The ListModel

- Allows list widget (view) to react to changes in the model

```java
//with a ListModel
ListModel model = new DefaultListModel();
model.addElement("a");
JList list = new JList(model);
```

```java
interface ListModel<T> {
    int getSize();
    T getElementAt(int index);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```
The ListModel

- Allows list widget (view) to react to changes in the model

```java
//with a ListModel
ListModel model = new DefaultListModel();
model.addElement("a");
JList list = new JList(model);
```

```java
interface ListModel<T> {
    int getSize();

    interface ListDataListener extends EventListener {
        void intervalAdded(...);
        void intervalRemoved(...);
        void contentsChanged(...);
    }
}
```
Scenario

- Assume we want to show all items of the virtual world in a list and update the items

```java
//design 1
class World implements ListModel<String> {
    List<Item> items ...

    int getSize() { return items.size(); }
    String getElementAt(int index) {
        items.get(index).getName();
    }
    void addListDataListener(ListDataListener l) { ... }
    protected void fireListUpdated() { ... }
}
```
Scenario

- Assume we want to show all items of the virtual world in a list and update the items

```java
//design 2
class World {
    DefaultListModel<Item> items ...

    public getListModel() { return items; }
    public Iterable<Item> getItems() {
        return items.elements();
    }
}
```
Scenario

• Assume we want to show all items of the virtual world in a list and update the items

//design 3
class WorldAdapter implements ListModel<String> {
  private final World world;
  public WorldAdapter(World w) {world = w;}

  int getSize() { return count(world.getItems()); }
  String getElementAt(int index) {
    find(world.getItems(), index);
  }
  void addListDataListener(ListDataListener l) {...}
...}
Scenario

- Assume we want to show all items of the virtual world in a list and update the items.
The Adapter Design Pattern

```
<<stereotype>>
Client

<<stereotype>>
Target
{abstract}
+Request() {abstract}

<<stereotype>>
Adapter

+Request()

```

```
Request()
{
  adaptee.SpecificRequest()
}

<<stereotype>>
Adaptee

+SpecificRequest()
```
The *Adapter* Design Pattern

**Applicability**
- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don’t necessarily have compatible interfaces
- You need to use several subclasses, but it’s impractical to adapt their interface by subclassing each one

**Consequences**
- Exposes the functionality of an object in another form
- Unifies the interfaces of multiple incompatible adaptee objects
- Lets a single adapter work with multiple adaptees in a hierarchy
- -> **Low coupling, high cohesion**
Other Scenarios for Adapters

- You have an application that processes data with an Iterator. Methods are:
  - `boolean hasNext();`
  - `Object next();`

- You need to read that data from a database using JDBC. Methods are:
  - `boolean next();`
  - `Object getObject(int column);`

- You might have to get the information from other sources in the future.
Façade vs. Adapter

• Motivation
  ▪ Façade: simplify the interface
  ▪ Adapter: match an existing interface

• Adapter: interface is given
  ▪ Not typically true in Façade

• Adapter: polymorphic
  ▪ Dispatch dynamically to multiple implementations
  ▪ Façade: typically choose the implementation statically
Design Goals

• Design to explicit interfaces
  ▪ Façade – a new interface for a library
  ▪ Adapter – design application to a common interface, adapt other libraries to that

• Favor composition over inheritance
  ▪ Façade – library is composed within Façade
  ▪ Adapter – adapter object interposed between client and implementation

• Design for change with information hiding
  ▪ Both Façade and Adapter – shields variations in the implementation from the client

• Design for reuse
  ▪ Façade provides a simple reusable interface to subsystem
  ▪ Adapter allows to reuse objects without fitting interface

...
Custom Renderer

- Renderer of list items and table cells exchangable
- Interface TableCellRenderer
- Strategy design pattern, again
Undoable Actions
Actions in GUIs

- We want to make actions accessible in multiple places
  - menu, context menu, keyboard shortcut, ...
  - When disabling an action, all places should be disabled

- We may want to undo actions

- Separate action execution from presentation

- Delay, queue, or log actions
  - eg macro recording, progress bars, executing remotely, transactions, wizards
Actions in Swing

- ActionListener
  - actionPerformed()

«interface»

- Action
  - getValue(in key)
  - putValue(in key, in value)
  - isEnabled()
  - isEnabled()
  - addPropertyChangeListener()

Keys:
- Name
- Icon
- Short description
- Long description
- Key combinations
- Selected

AbstractButton
- +getAction()
- +setAction()

* 1

JButton

JMenuItem

AbstractAction
- enabled
- ...

+ ...()
Action vs. ActionListener

- **Action is self-describing (text, shortcut, icon, ...)**
  - Can be used in many places
  - e.g. log of executed commands, queue, undo list

- **Action can have state (enabled, selected, ...)**

- **Actions are synchronized in all cases where they are used**
  - with observer pattern: PropertyChangeListener
Implementing a Wizard

- Every step produces some execution
- Execution is delayed until user clicks finish
- Collect objects representing executions in a list
- Execute all at the end
- -> Design for change, division of labor, and reuse; low coupling

```
interface ExecuteLater {
    void execute();
}
```
Implementing a Wizard

- Every step produces some execution

```java
class SetCompanyName implements ExecuteLater {
    private final String name;
    private final Registry registry;
    SetCompanyName(String n, Registry r) {
        name = n; registry = r;
    }
    void execute() {
        registry.writeKey(..., name);
    }
}
```

reusability; low coupling
The *Command* Design Pattern
The **Command** Design Pattern

- **Applicability**
  - Parameterize objects by an action to perform
  - Specify, queue and execute requests at different times
  - Support undo
  - Support logging changes that can be reapplied after a crash
  - Structure a system around high-level operations built out of primitives

- **Consequences**
  - Decouples the object that invokes the operation from the one that performs it
  - Since commands are objects they can be explicitly manipulated
  - Can group commands into composite commands
  - Easy to add new commands without changing existing code
Common Commands in Java

- **javax.swing.Action**
  - see above

- **java.lang.Runnable**
  - Used as an explicit operation that can be passed to threads, workers, timers and other objects or delayed or remote execution
  - see FutureTask
Commands in the Virtual World

- Represent common actions
  - Reusing implementation
  - Validity checking of moves (e.g., MoveCommand can only be called on moveable items; rabbits may only move within moving range)

- Could be queued and checked separately

- Separate creation of command from its execution:
  - AI: Command getNextAction(ArenaWorld) vs Command: void execute(World)
  - Both can throw different kinds of exceptions
Undoable Actions

- Remember undoable changes: Store effect with information how to undo it in a list
  - Delete command with removed text and location
  - Paste command with location of added text
  - Swing: UndoableEdit interface

- Provide an undo and redo implementation for each undoable change

- Queue undoable changes in a history; provide user interface actions for undo and redo
notice the composite pattern

The GUI Threading Architecture

main() thread:
- Create window
- Set up callbacks
- Show window (thread ends)

GUI Thread:
- Loop forever:
  - Get system event
  - Invoke callback
- Callback code:
  - Compute fibonacci
  - (UI is unresponsive)
  - Show result
The GUI Threading Architecture

main() thread

GUI thread

Worker thread

Create window
Set up callbacks
Show window (thread ends)

Loop forever:
Get system event
Invoke callback

Callback code:
create SwingWorker
start it executing

Worker thread execution:
invoke doInBackground()
compute fibonacci
store result in SwingWorker
signal to UI that we are done

SwingWorker
result : Long

create SwingWorker
start it executing

create SwingWorker
start it executing

create SwingWorker
start it executing

create SwingWorker
start it executing
The GUI Threading Architecture

main() thread

Create window
Set up callbacks
Show window (thread ends)

GUI thread

Loop forever:
  Get system event
  Invoke callback

Worker thread

Worker thread execution:
  invoke doInBackground()
  compute fibonacci
  store result in SwingWorker
  signal to UI that we are done

Invoke SwingWorker.done()
  get() result from SwingWorker
  show result in the UI

SwingWorker
  result : Long
Summary

- GUIs are full of design pattern
  - Strategy Pattern
  - Template Method Pattern
  - Composite Pattern
  - Observer Pattern
  - Façade Pattern
  - Adapter Pattern
  - Command Pattern
  - Model-View-Controller Metapattern

- Swing for Java GUIs

- Separation of GUI and Core
Design Challenge

• In the Point of Sales terminal the total of a sale should be computed and coupons should be taken into account

• Coupons and other special actions may change over time: design for change

• Design 1-3 solutions that share common computations, but is extensible for new coupons or other promotions