Principles of Software Construction: Objects, Design and Concurrency

GUIs with Swing

15-214

toad

Spring 2013

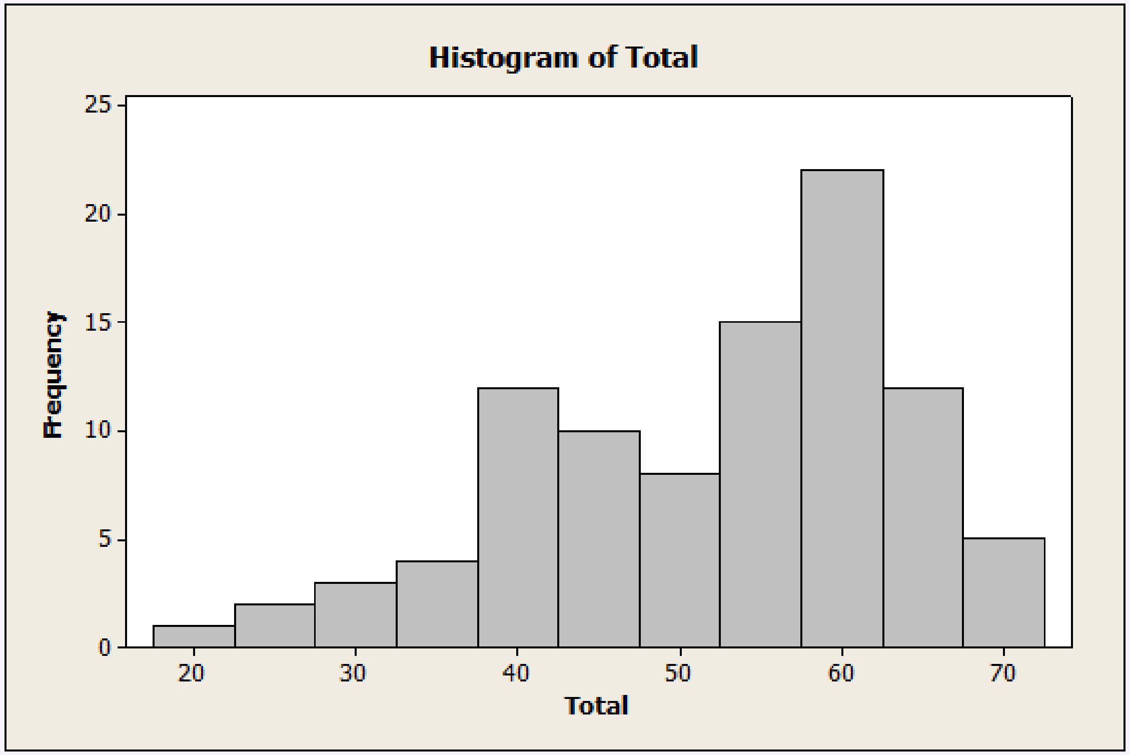
Christian Kästner Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Midterm results



- Homework 4a grading expected by tonight
  - Feedback on your design and design questions for 4b
- Homework 4b due Tuesday after spring break

Recap from last week

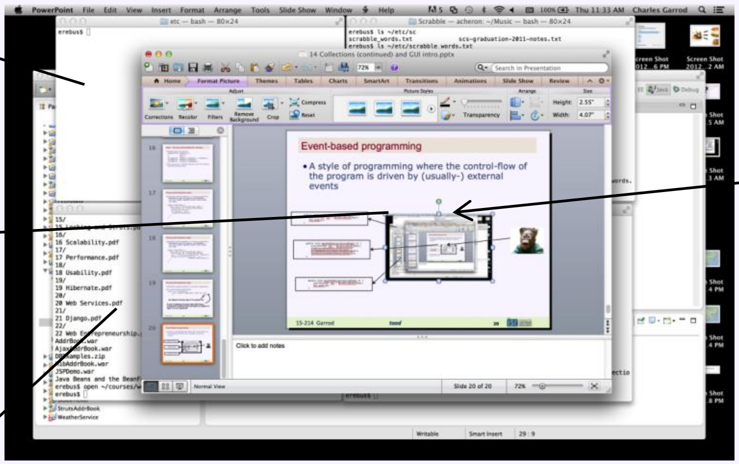
Event-based programming

- A style of programming where the control-flow of the program is driven by (usually-) external events

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

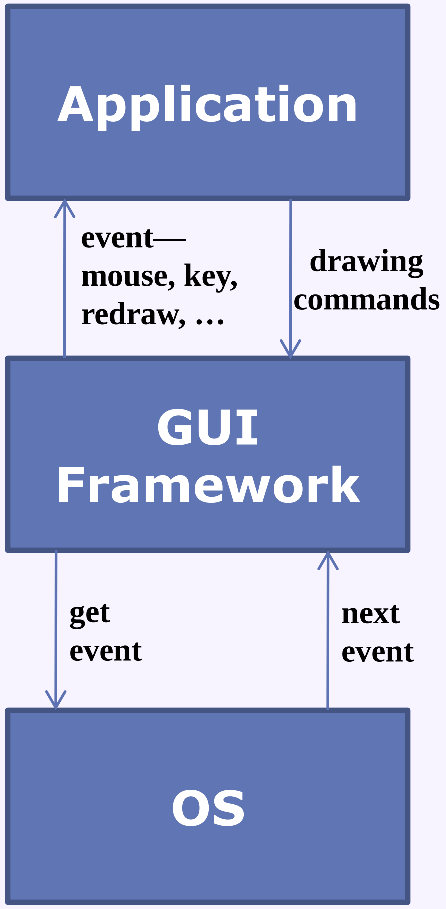
```
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

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


PowerPoint File Edit View Insert Format Arrange Tools Slide Show Window Help
Event-based programming
Slide 20 of 20
Click to add notes

Reacting to events - from framework

- Setup phase
  - Describe how the GUI window should look
  - Use libraries for windows, widgets, and layout
  - Embed specialized code for later use
- Customization (provided during setup)
  - New widgets that display themselves in custom ways
  - How to react to events
- Execution
  - Framework gets events from OS
    - Mouse clicks, key presses, window becomes visible, etc.
  - Framework triggers application code in response
    - The customization described above



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, …

Swing

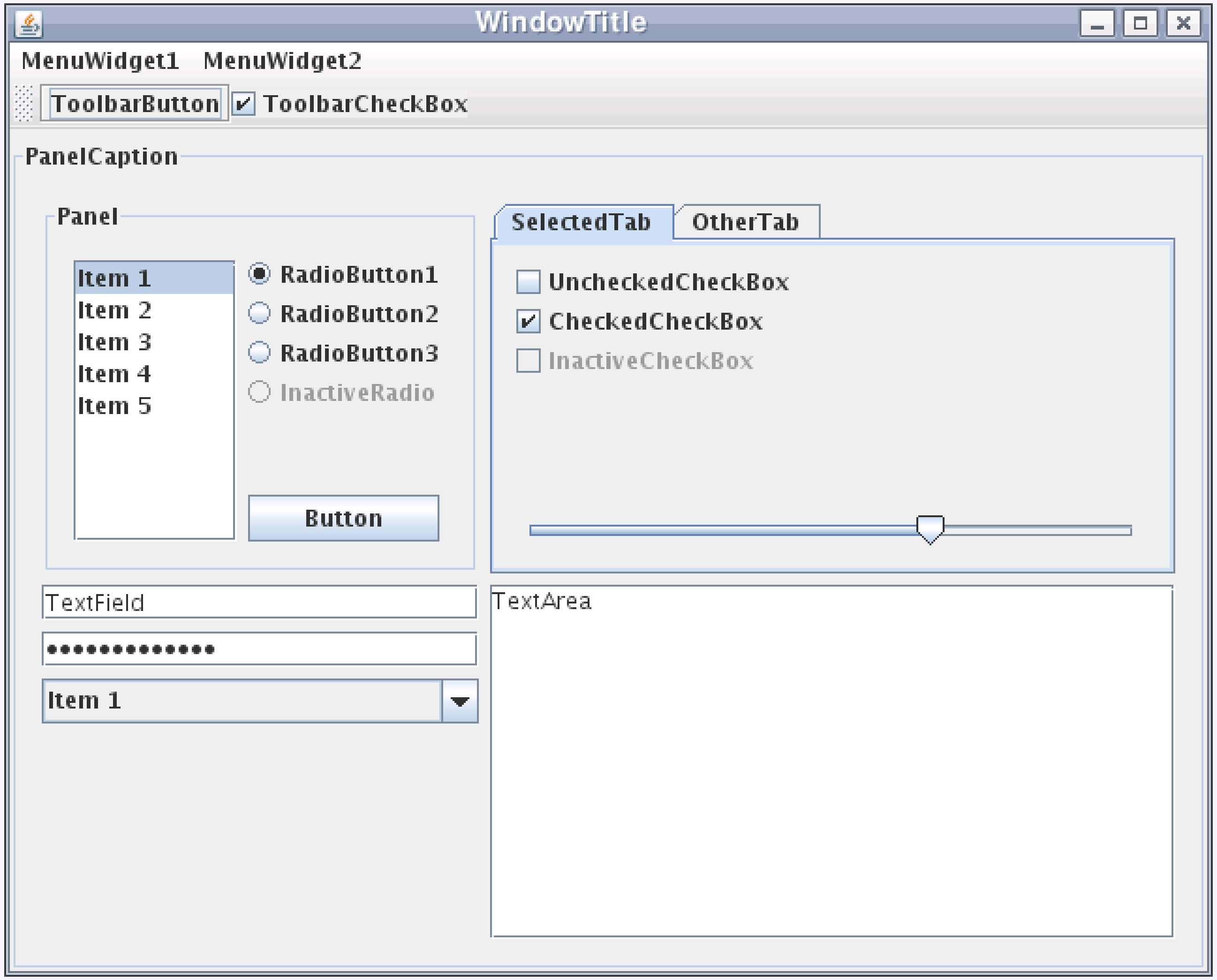
JFrame

JPanel

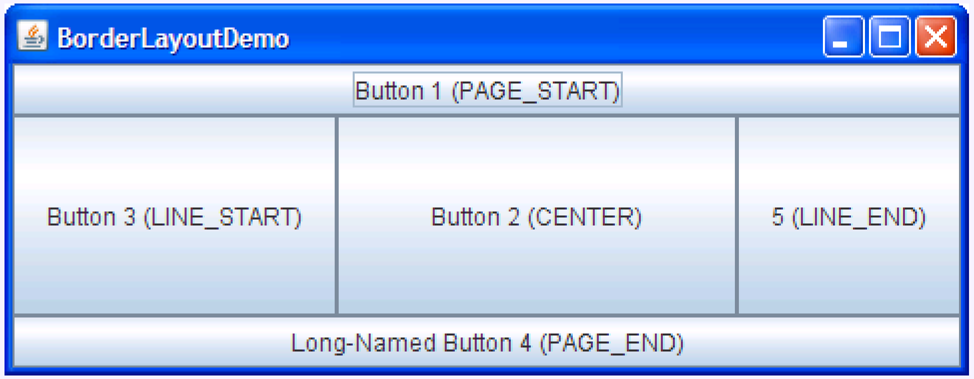
JButton

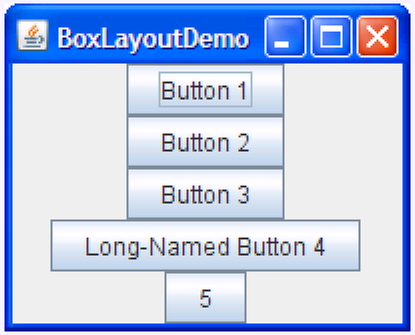
JTextField

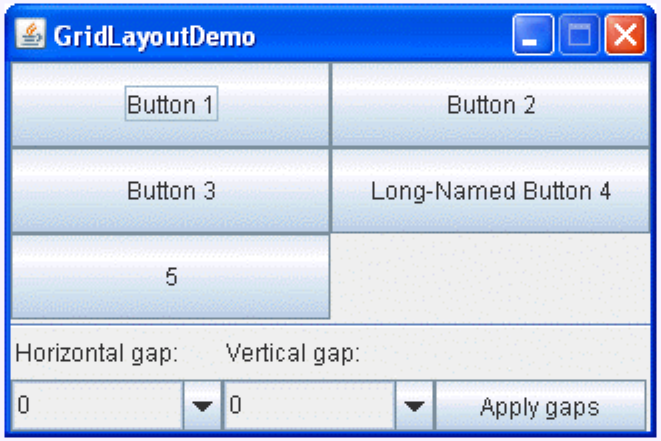
…

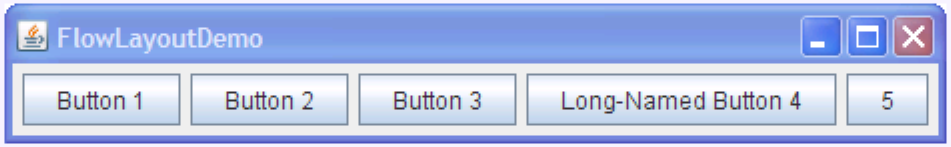


Swing Layout Manager









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

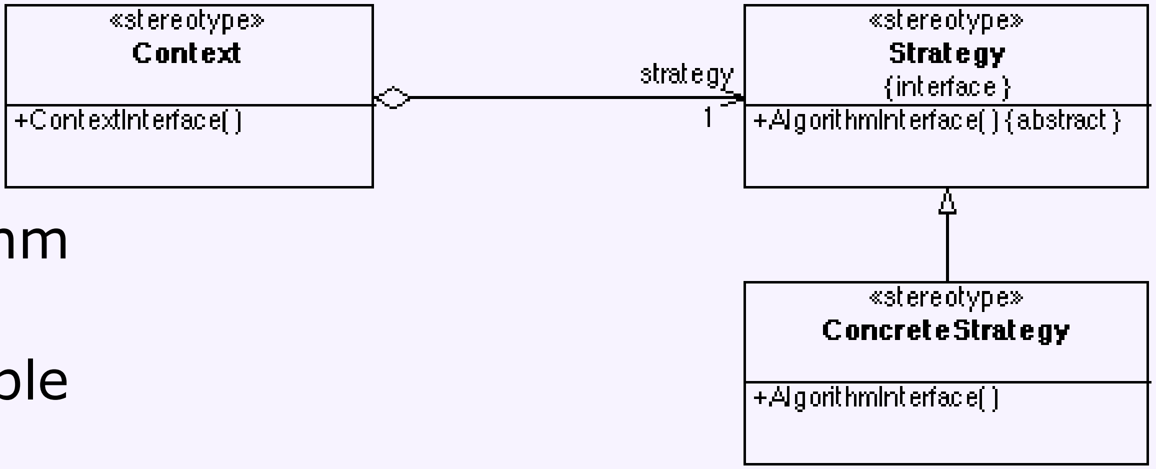
Find the pattern…

- contentPane.setLayout(new BorderLayout(0,0));

- contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));

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
  - Adds objects and dynamism
    - code harder to understand
  - Common strategy interface
    - may not be needed for all Strategy implementations – may be extra overhead



Design Pattern in Swing

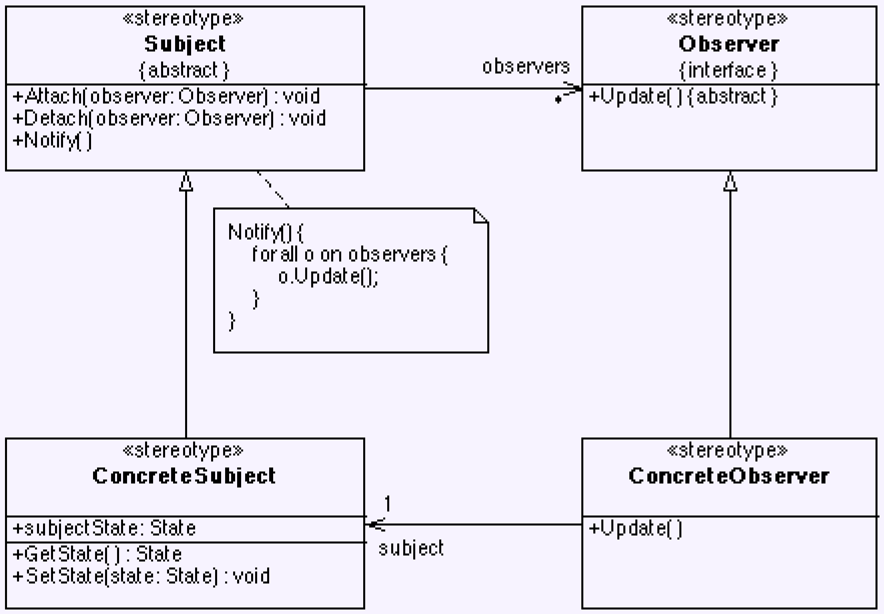
- Observer Pattern
- Strategy Pattern
- Composite Pattern
- Command Patterns (Actions)
- Model View Controller Pattern
- Controller / Facade

Swing has lots of event listener interfaces:

- ActionListener
- AdjustmentListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- TreeExpansionListener
- TextListener
- WindowListener
- …and on and on…

Reminder: 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



```
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);
        }
});

panel.add(btnNewButton, BorderLayout.SOUTH);
```

```
class MyButton extends JButton {
        Counter counter;
        public MyButton() {
                … setTitle…
        }
        protected void fireActionPerformed(ActionEvent event) {
                counter.inc();
        }
}

JButton btnNewButton = new MyButton();
panel.add(btnNewButton, BorderLayout.SOUTH);
```

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 issues

- 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()
    - 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
- Models and views

Command Pattern implementable

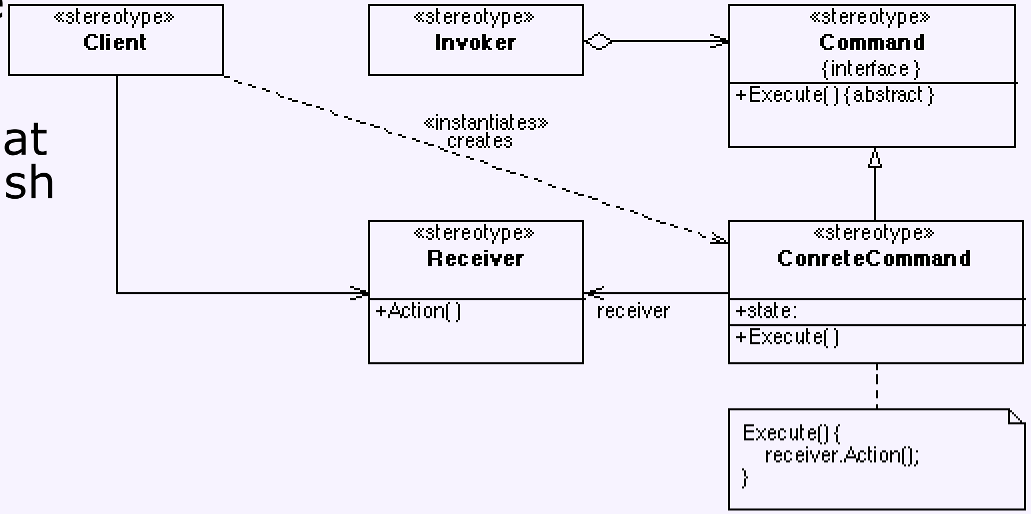
- AbstractAction can serve as command pattern

```
Action openAction = new AbstractAction("open...") {
          public void actionPerformed(ActionEvent e) {…}
}
fileMenu.add(openAction);
```

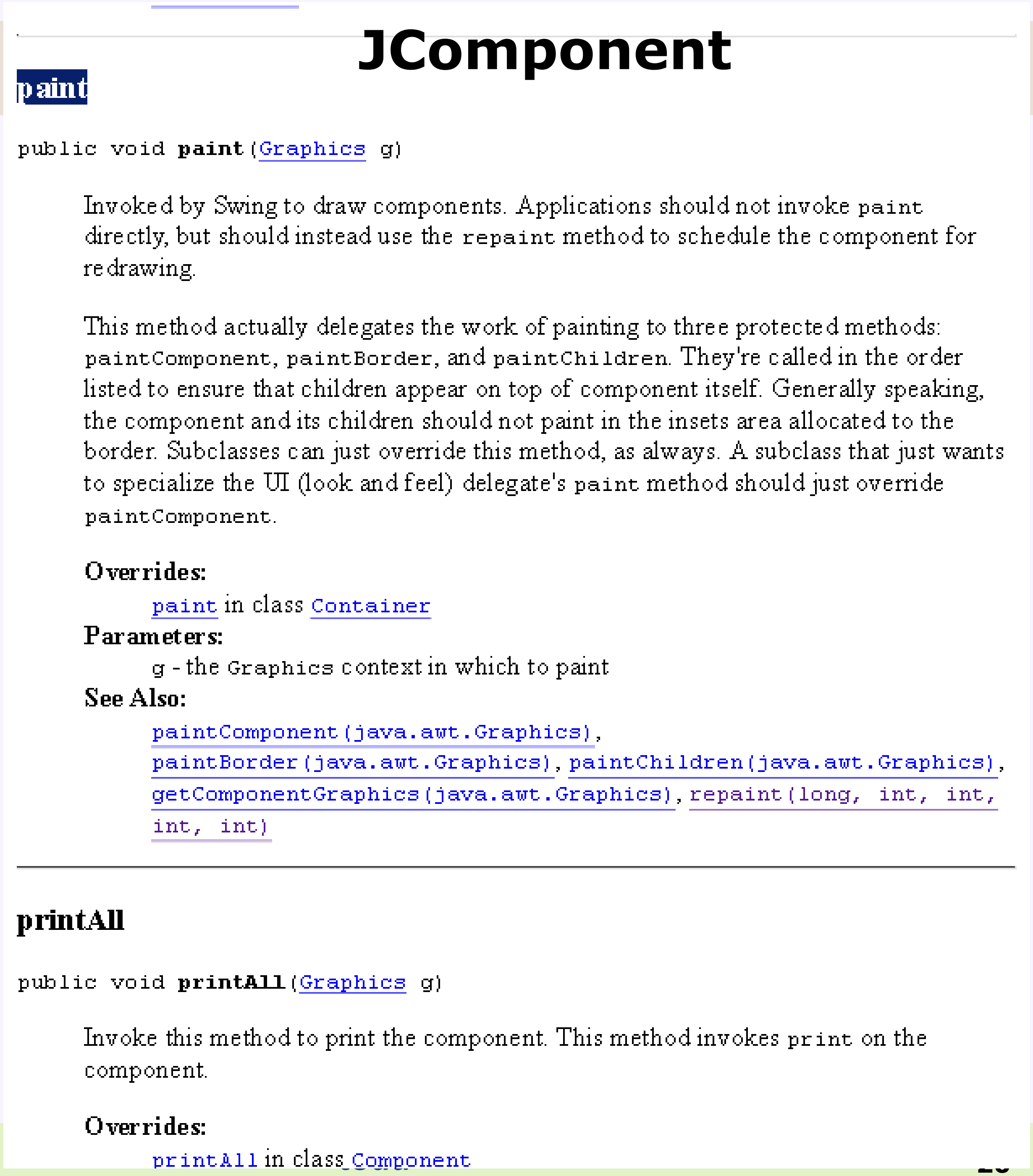
- see also javax.swing.undo package

Reminder: Command

- 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



JComponent



paint

```
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)`

---

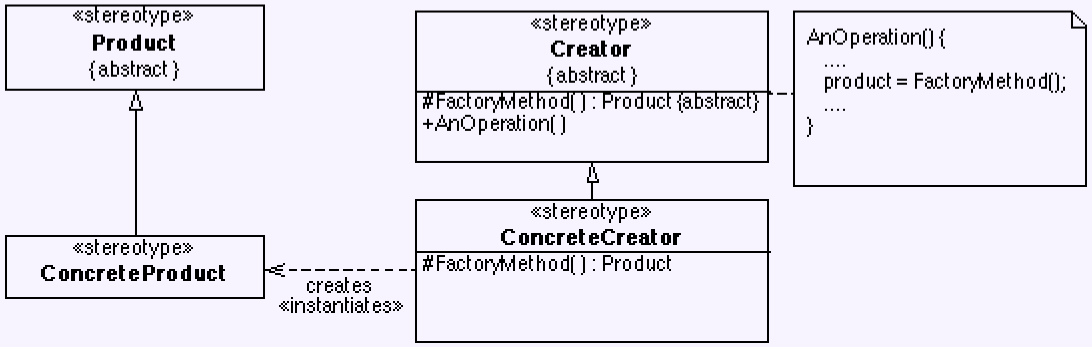
printAll

```
public void printAll(Graphics g)
```

Invoke this method to print the component. This method invokes `print` on the component.

Overrides:
`printAll` in class `Component`

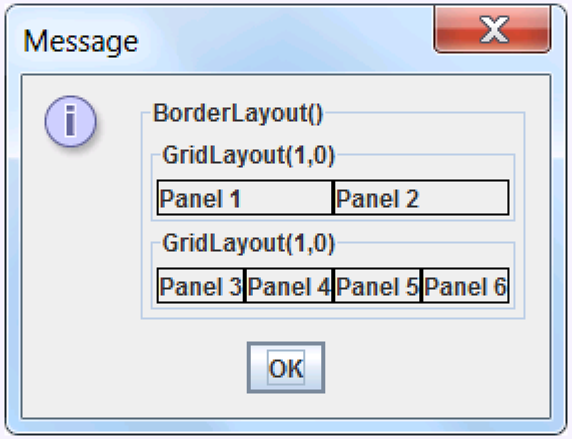
Reminder: Factory Method



```
AnOperation() {
  ....
  product = FactoryMethod();
  ....
}
```

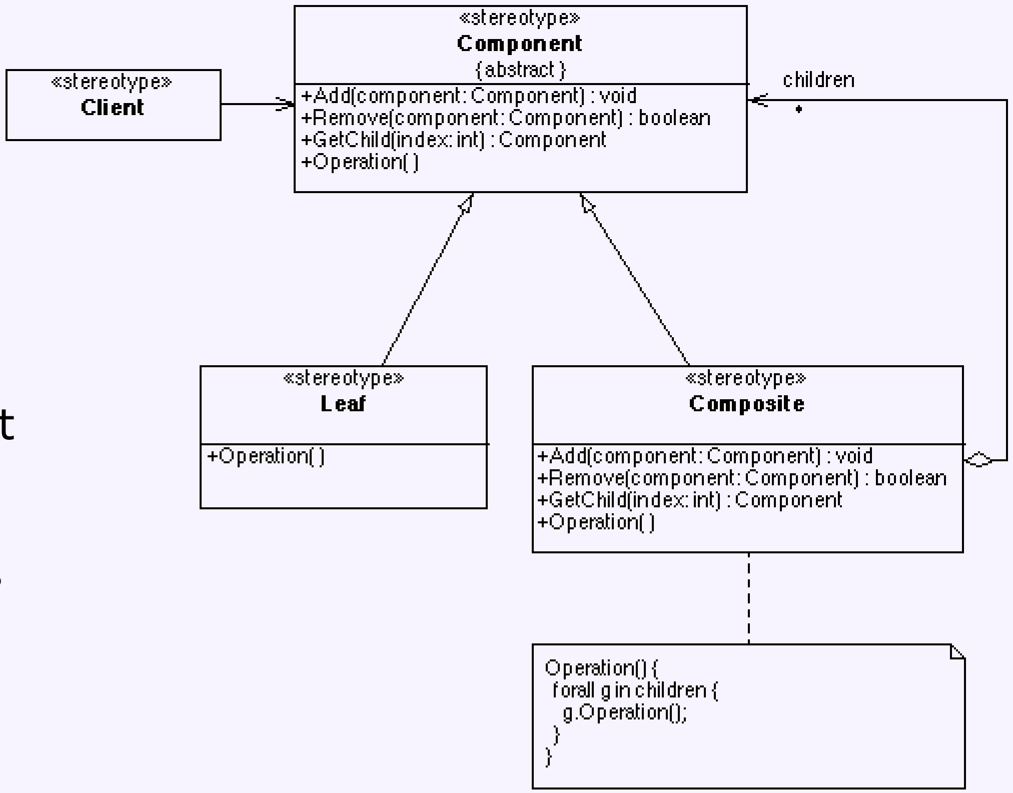
- Applicability
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
- Consequences
  - Provides hooks for subclasses to customize creation behavior
  - Connects parallel class hierarchies

Nesting Containers



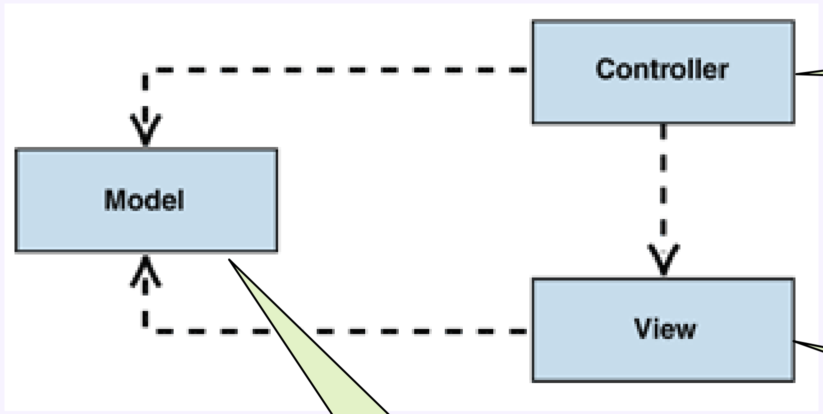
Reminder: Composite

- 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


«stereotype»
Client
«stereotype»
Component
{abstract}
+Add(component: Component) : void
+Remove(component: Component) : boolean
+GetChild(index: int) : Component
+Operation( )
children
*
«stereotype»
Leaf
+Operation( )
«stereotype»
Composite
+Add(component: Component) : void
+Remove(component: Component) : boolean
+GetChild(index: int) : Component
+Operation( )

```
Operation() {
 forall g in children {
   g.Operation();
  }
}
```

Model-View-Controller (MVC)



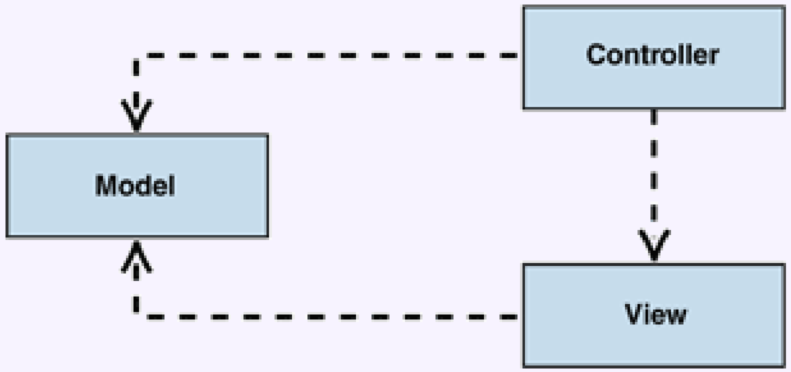
Manage inputs from user: mouse, keyboard, menu, etc.

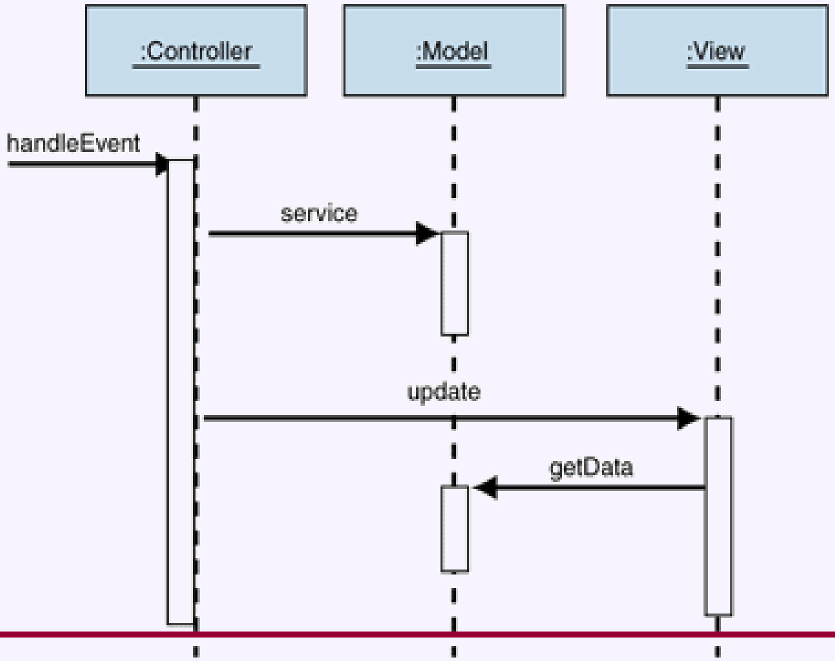
Manage display of information on the screen

Manage data related to the application domain

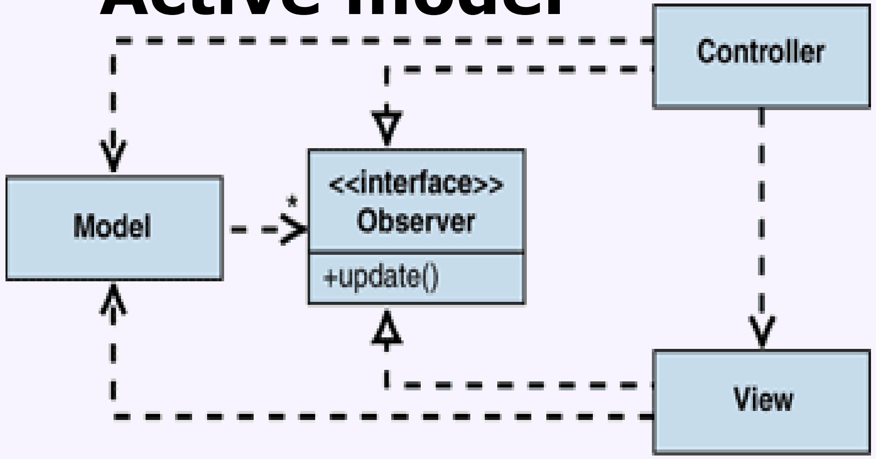
Model-View-Controller (MVC)

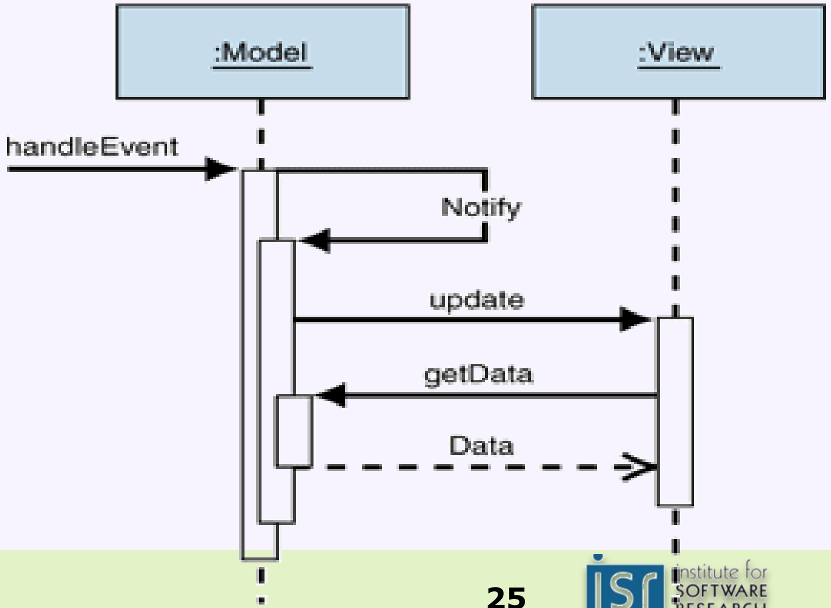
Passive model





Active model

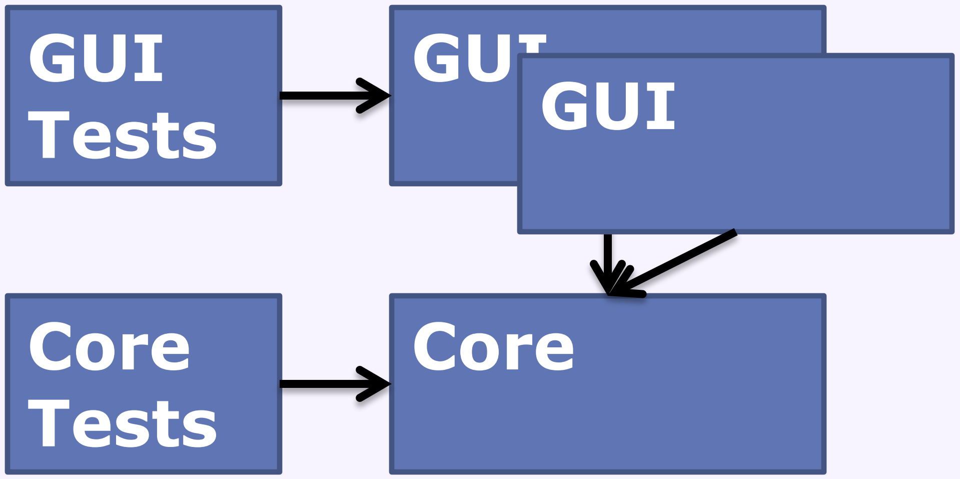


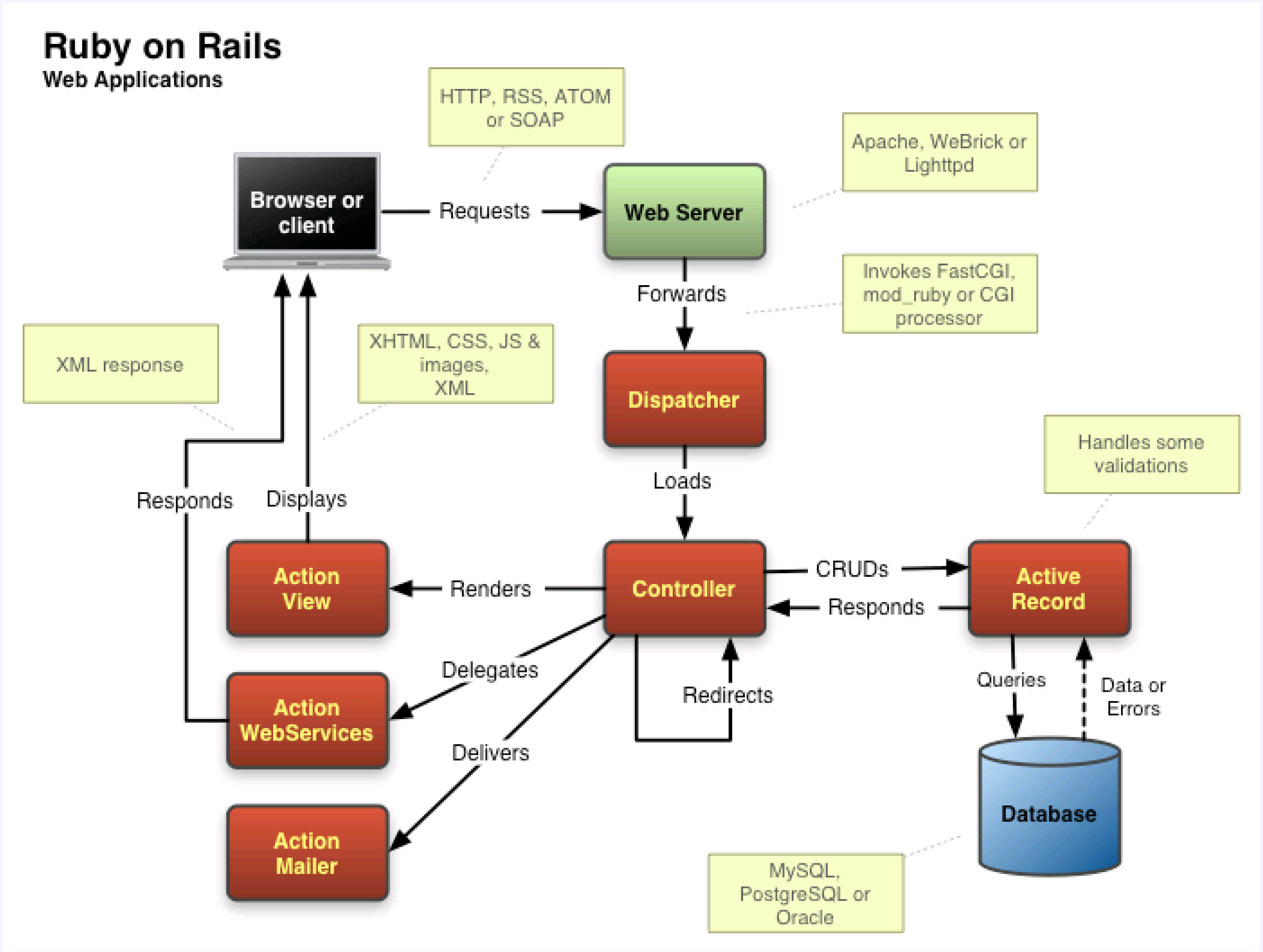


http://msdn.microsoft.com/en-us/library/ff649643.aspx

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




Ruby on Rails
Web Applications
HTTP, RSS, ATOM or SOAP
Apache, WeBrick or Lighttpd
Browser or client
Requests
Web Server
Forwards
Invokes FastCGI, mod_ruby or CGI processor
XML response
XHTML, CSS, JS & images, XML
Dispatcher
Loads
Handles some validations
Responds
Displays
Action View
Renders
Controller
CRUDs
Responds
Active Record
Delegates
Action WebServices
Delivers
Redirects
Queries
Data or Errors
Database
Action Mailer
MySQL, PostgreSQL or Oracle

Summary

- GUIs are full of design pattern
- Swing for Java GUIs
- Separation of GUI and Core