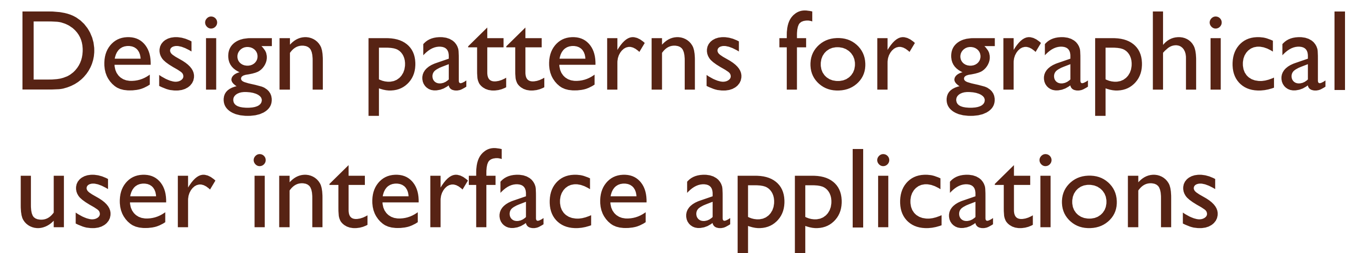


Prof.Asoc. Alda Kika
Department of Informatics
Faculty of Natural Sciences
University of Tirana

Outline

- Pattern Concept
- Design pattern in computer science
- Observer Pattern and Model View Controller Architecture
- Strategy
- Composite
- Decorator

Pattern Concept

- Christopher Alexander
- His 1979 book called "The Timeless Way of Building" that asks the question "Is quality objective?"
- When he found an example of a high quality design, he would compare that object to other objects of high quality and look for commonalties, especially if both objects were used to solve the same type of problem
- "… a three part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, … , and the rule which tells us how to create that thing, and when we must create it."

Pattern Concept

- The pattern provides an approach that can be used to achieve a high quality solution to its problem.
- Each pattern has:
  - A short name
  - A brief description of the context
  - A lengthy description of the problem
  - A prescription for the solution

Short passages Pattern

- Context

." . . . long, sterile corridors set the scene for everything bad about modern architecture.“

- Problem

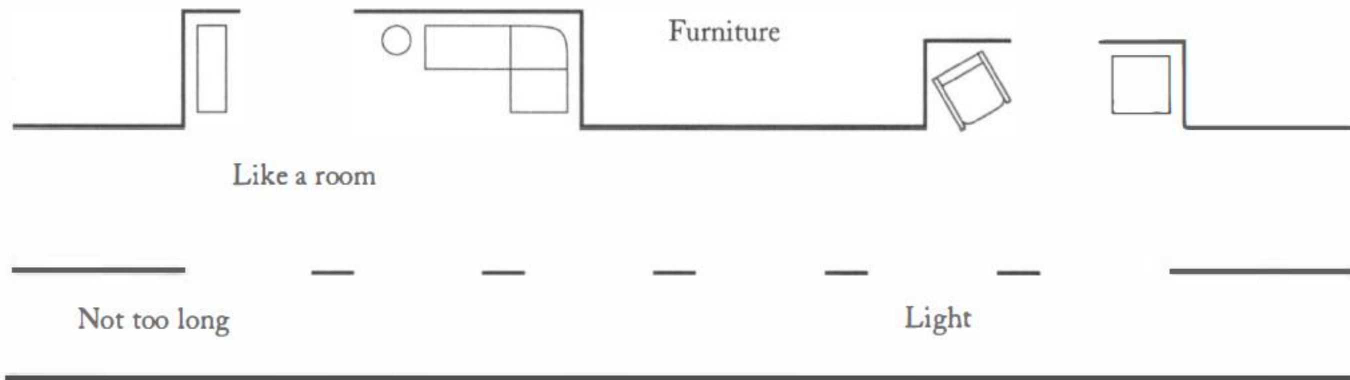
Alexander discusses issues of light and furniture. He cites research results about patient anxiety in hospital corridors. According to the research, corridors that are longer than 50 feet are perceived as uncomfortable.

Short passages Pattern

Short passages Pattern

- Solution

Keep passages short. Make them as much like rooms as possible, with carpets or wood on the floor, furniture, bookshelves, beautiful windows. Make them generous in shape and always give them plenty of light; the best corridors and passages of all are those that have windows along an entire wall.



Design patterns

A design pattern gives advice about a problem in software design.

A pattern is a description of a problem and its solution that you can apply to many programming situations.

- Design Patterns
- Model-View-Controller and Observer
- Strategy
- Composite
- Decorator

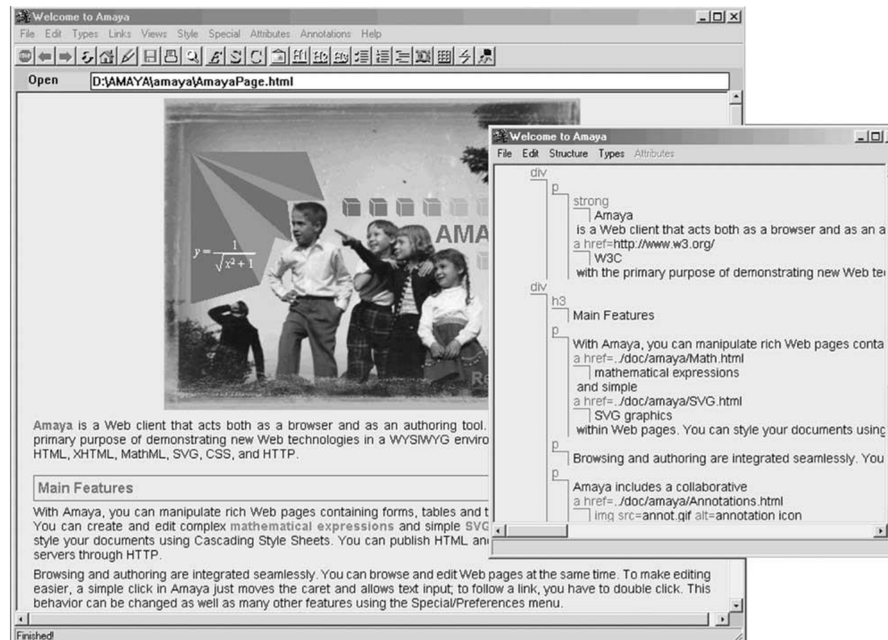
Design patterns

- For each design pattern
  - Prototypical example
  - Context, problem
  - Solution, decoupling/interface
  - UML diagram
- Patterns
  - Observer and MVC, strategy, composite, decorator

Observer Pattern

- Some programs show multiple editable views of the same data
  - "what you see is what you get" (WYSIWYG)

and a structural view of a document



Model/View/Controller

- Model: data structure, no visual representation
- Views: visual representations
- Controllers: user interaction

- Views/controllers update model
- Model tells views that data has changed
- Views redraw themselves

Model

- The Model's responsibilities
  - Provide access to the state of the system
  - Provide access to the system's functionality
  - Can notify the view(s) that its state has changed

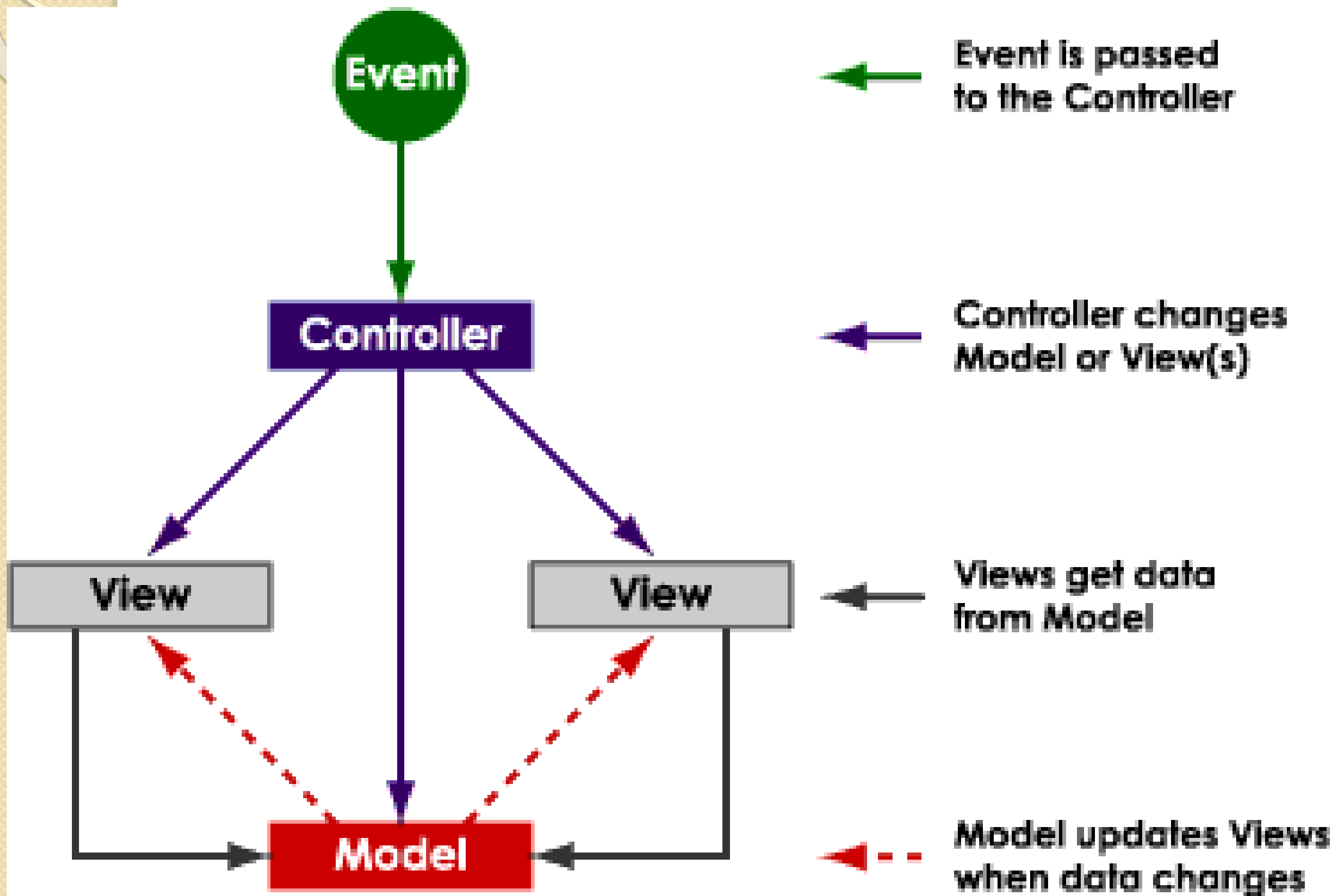
View

- The view's responsibilities
  - Display the state of the model to the user
- At some point, the model must registers the views so the model can notify the observers that its state has changed

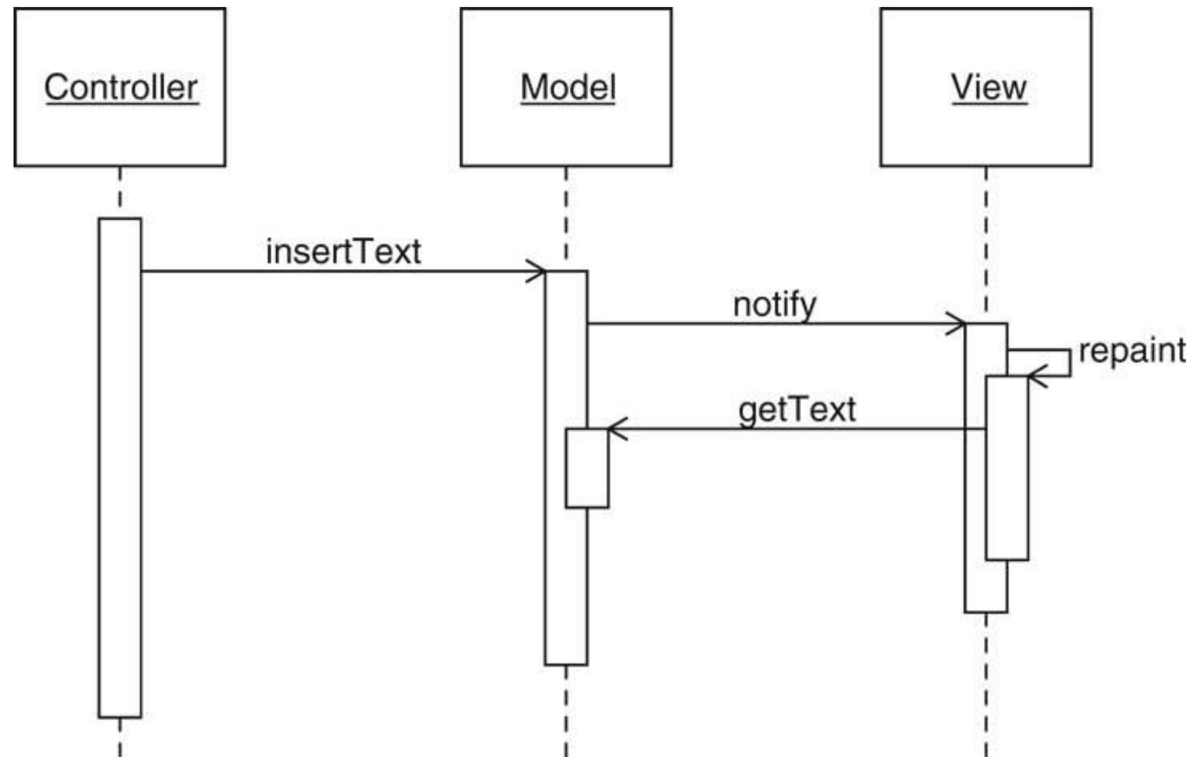
Controller

- The controller's responsibilities
  - Accept user input
    - Button clicks, key presses, mouse movements, slider bar changes
  - Send messages to the model, which may in turn notify it observers
  - Send appropriate messages to the view

from http://www.enode.com/x/markup/tutorial/mvc.html)



Model/View/Controller



Button / Listener

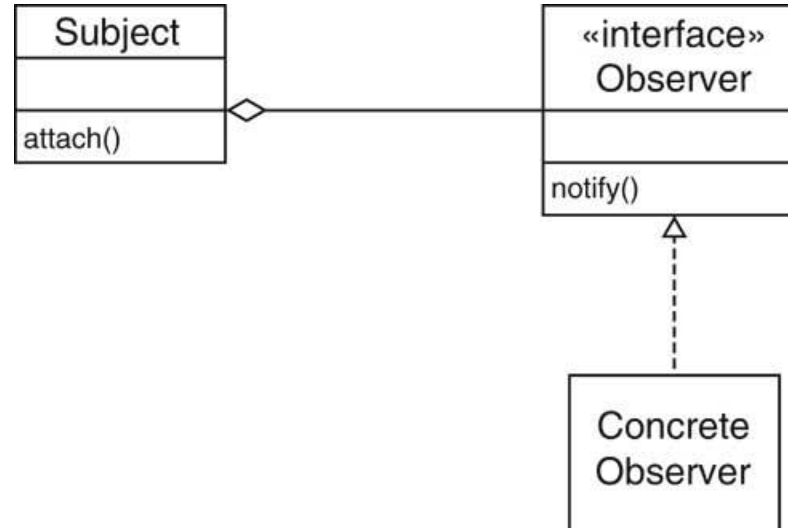
```
helloButton.addActionListener(new
  ActionListener() {
    public void actionPerformed(ActionEvent
  event) {
      textField.setText("Hello, World");
    }
  }
);
```

Observer Pattern

- **Context**
  - An object (which we'll call the subject) is the source of events (such as "my data has changed").
  - One or more objects (called the observers) want to know when an event occurs.

- Solution
  - Define an observer interface type. Observer classes must implement this interface type.
  - The subject maintains a collection of observer objects.
  - The subject class supplies methods for attaching observers.
  - Whenever an event occurs, the subject notifies all observers.

Observer Pattern



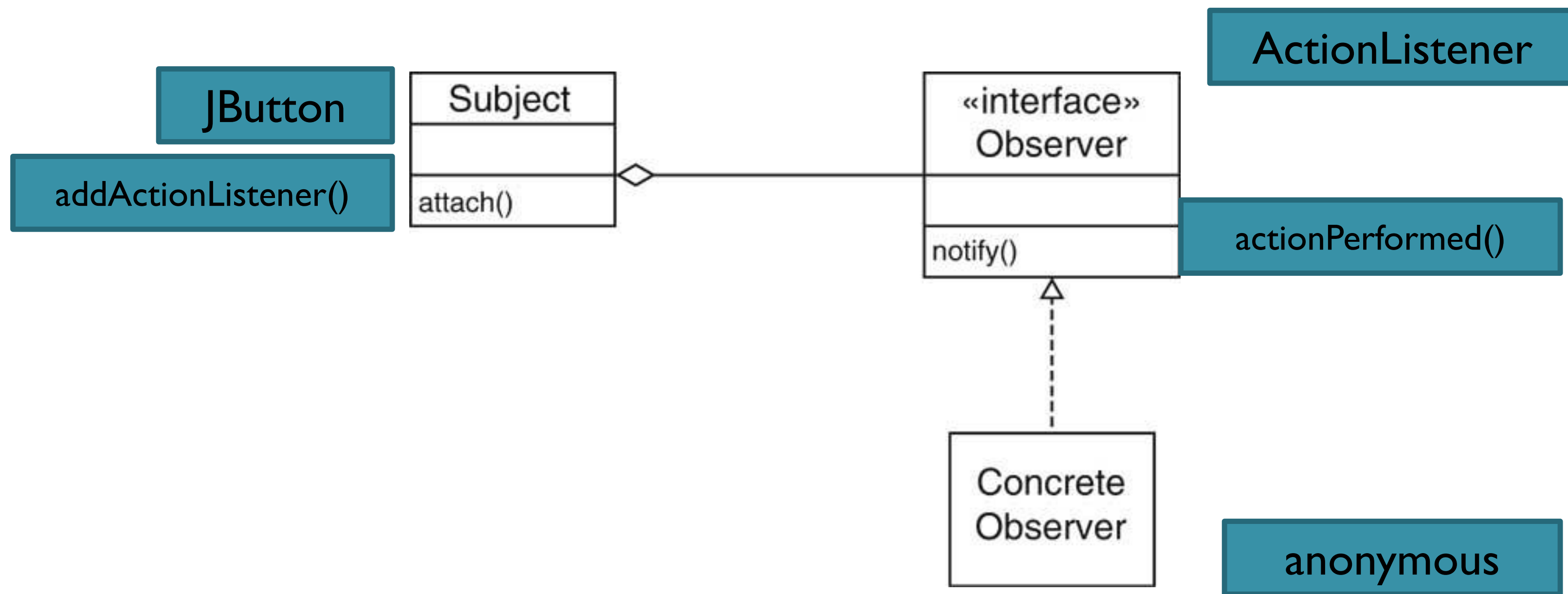
Observer Pattern

- Model notifies views when something interesting happens
- Button notifies action listeners when something interesting happens

- Views attach themselves to model in order to be notified
- Action listeners attach themselves to button in order to be notified

- Generalize: *Observers* attach themselves to *subject*

Observer Pattern



Observer Pattern

Name in Design Pattern	Actual Name
Subject	JButton
Observer	ActionListener
ConcreteObserver	The class that implements the ActionListener interface type
attach()	addActionListener
notify()	actionPerformed

Observable Class and Observer Interface

- Java implementation of observer design pattern
  - Class java.util.Observable
    - This class represents a model in MVC, or the subject in the Observer design pattern
    - Class **Observable** provides method ***addObserver***() which takes a **Observer** object as argument
  - Interface Observer
    - Interface **Observer** represents the view in MVC and enables loose coupling between an **Observable** object and its **Observer**s
    - **Observable** object notifies each registered objects whenever it changes
    - The **Observer** can be an instance of any class that implements interface **Observer**

Account Manager

Account 1

Amount: | Deposit | Withdraw

Balance: $1,000.00 | -$5,000 | $0 | +$5,000

Account 2

Amount: | Deposit | Withdraw

Balance: $3,000.00 | -$5,000 | $0 | +$5,000

Assets

Account 1
Account 2

Account Manager

Account 1

Amount: 1500.00 | Deposit | Withdraw

Balance: $2,500.00 | -$5,000 | $0 | +$5,000

Account 2

Amount: | Deposit | Withdraw

Balance: $3,000.00 | -$5,000 | $0 | +$5,000

Assets

Account 1
Account 2

Account Manager

Account 1

Amount: 1500.00 | Deposit | Withdraw

Balance: $2,500.00 | -$5,000 | $0 | +$5,000

Account 2

Amount: 4623.12 | Deposit | Withdraw

Balance: ($1,623.12) | -$5,000 | $0 | +$5,000

Assets

Account 1
Account 2

Account Manager

Account 1

Amount: 3210.93 | Deposit | Withdraw

Balance: ($710.93) | -$5,000 | $0 | +$5,000

Account 2

Amount: 4623.12 | Deposit | Withdraw

Balance: ($1,623.12) | -$5,000 | $0 | +$5,000

Assets

Account 1
Account 2

AccountManager Application

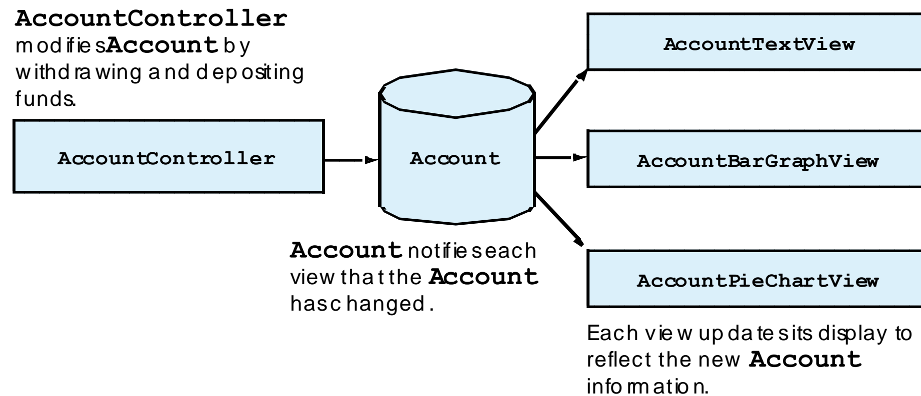


Fig. 3.3 **AccountManager** application MVC architecture.

AccountManager Application

- Uses MVC architecture, class **Observable**, and interface **Observer** to implement an **AccountManager** application for managing bank account information
- Major components
  - Class **AccountController**
    - Accepts user input as dollar amounts entered in a **JTextField**
    - The user selects a **JButton** to withdraw or deposit the given amount
    - Modifies the **Account** to execute the transaction
  - Class **Account**
    - Is an Observable object that acts as the application's model
    - When the **AccountController** performs the withdrawal or deposit, the Account notifies each view that the Account information has changed
      - **AccountTextView**, **AccountBarView**, **AccountPieChartView**
    - Each view updates its display with the modified **Account** information

Account Class

- Represents a bank account in the **AccountManager** application
- Class **Account** extends class **Observable** and acts as a model in the application
  - Method ***setChanged***() of class **Observable** set model's ***changed*** flag
  - Method ***notifyObservers***() of class **Observable** notify all **Account Observers** (Views) of the change.
  - An **Observable** object must invoke method ***setChanged***() before invoking method ***notifyObservers***()
  - Method ***notifyObservers***() invokes method ***update***() of interface **Observer** for each registered **Observer**

Class AbstractAccountView

- Application **AccountManager** presents **Account** information to the user through three views
  - **AccountTextView**, **AccountBarGraphView**, **AccountPieChartView**
- Each view presents a different visual representation of the **Account** information
- **AbstractAccountView** is an abstract base class for these **Account** views and provides common functionality
  - Such as registering as an Account observer
- Implements interface **Observer**, which allows each AbstractAccountView subclass to register as an **Observer** of an **Account**
- Extends **JPanel** because it provide graphical presentations

Class AccountBarGraphView

- Extends **AbstractAccountView** to provide a bar-graph view of **Account** data
- Method ***paintComponen***t() draws a bar graph for the current **Account** balance
- Method ***updateDisplay***() invokes method ***repaint***() to update the bar graph's display
- ***AbstractAccountView*** method ***update***() invokes method ***updateDisplay***() each time the **Account** model notifies the view of a change in the model

Class AbstractAccountView

- Method ***updateDisplay()*** is marked **abstract**, requiring each **AbstractAccountView** subclass to provide an appropriate implementation for displaying the **Account** information
- ***update()*** invokes ***updateDisplay***() each time an Account notifies the **AbstarctAccountView** of a change
- Interface **Observer** defines method update, which takes as an argument a reference to an **Observable** instance that issued the **update**
- An **Observable** object issues an **update** by invoking ***notifyObservers***() of class **Observable**
- notifyObservers() invoke update() for each registered Observer
- An Observer can listen for updates from multiple Observable objects

Class AssetPieChartView

- **AssetPieChartView** provides a pie-char view of multiple asset **Account**s
- **AssetPieChartView** shows the percentage of total assets held in each **Account** as wedges in the pie chart
- Defines method ***addAccount***() which, adds an **Account** to the List of **Account** shown in the pie chart
- ***removeAccount***() removes an **Account** from the pie chart

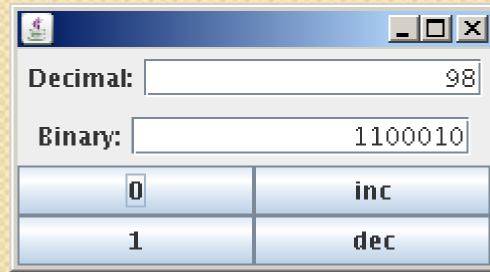
Class AccountController

- **AccountController** implements the controller in the MVC architecture
- **AccountController** provides a user interface for modifying **Account** data
- **AccountController** extends **JPanel** because it provides a set of GUI components for depositing and withdrawing **Account** funds

Class AccountManager

- **AccountManager** is an application that uses MVC to manage **Account** information
- Method ***createAccountPanel***() creates a **JPanel** containing an **AccountController**, **AccountTextView** and **AccountBarGraphView** for the given Account.

QUIZ



Display Components

Button Components

Which interpretations are reasonable?

1. None
2. a
3. b
4. c
5. a+b
6. a+c
7. b+c
8. a+b+c
9. I don't know

a) MVC:

Frame is Model

Display Components are Views

Buttons are Controllers

b) MVC:

Model is not visible, but contains a number

Display Components are Views

Listeners attached to Buttons are Controllers

c) Observer:

A Button is a Subject

A Listener attached to a button is an Observer

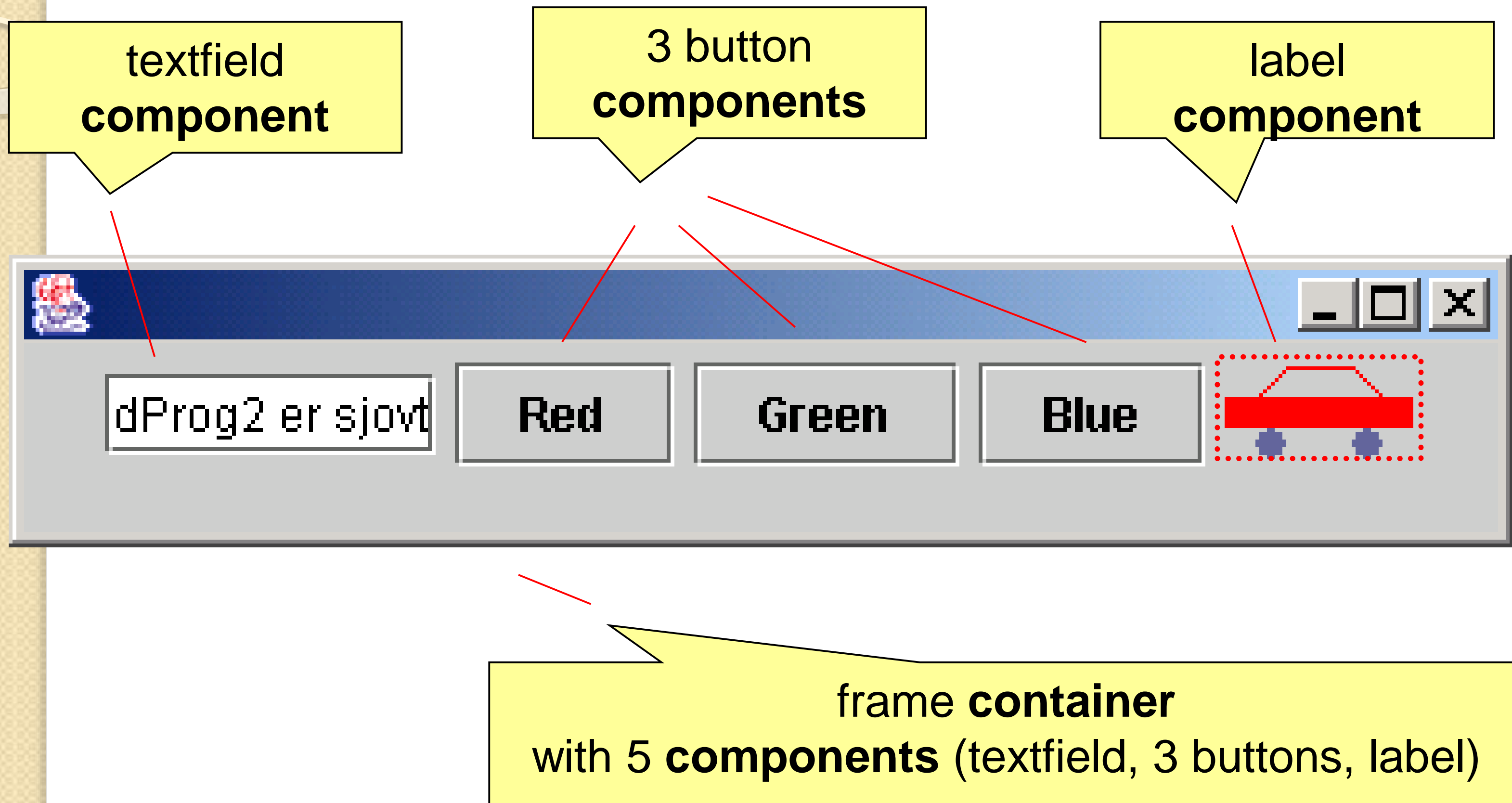
Design patterns

- Design Patterns
- Model-View-Controller and Observer
- **Strategy**
- Composite
- Decorator

Strategy Pattern and Layout Managers

- To build user interface we insert user interface components into containers. It is not a good idea to specify pixel position for each component.
  - The user chooses a different "look and feel".
  - The program gets translated into a different language.
- A layout manager arranges the components in a container.

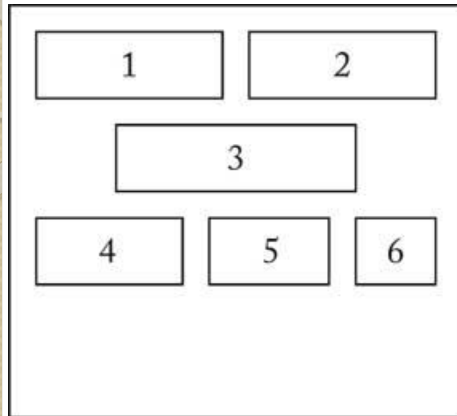
GUI: components and containers



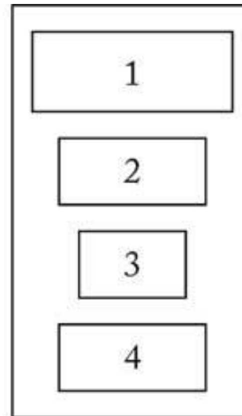
Layout Managers

- User interfaces is made up of *components*
- Components are placed in *containers*
- Container needs to arrange components
- Swing doesn't use pixel coordinates
- Advantages:
  - Can switch "look and feel"
  - Can internationalize strings
- Layout manager controls arrangement

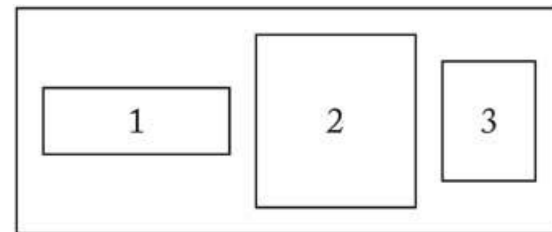
Layout Managers



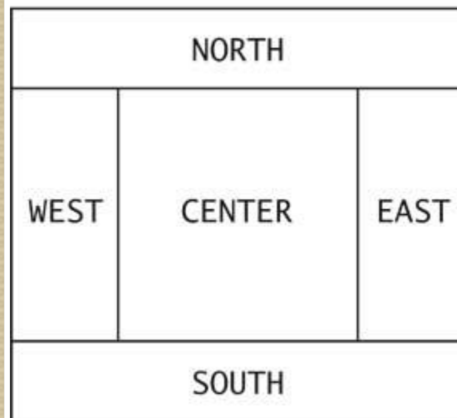
FlowLayout



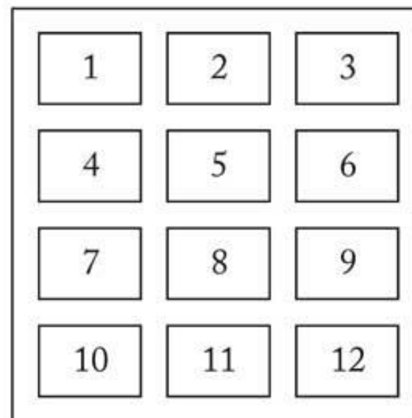
BoxLayout (vertical)



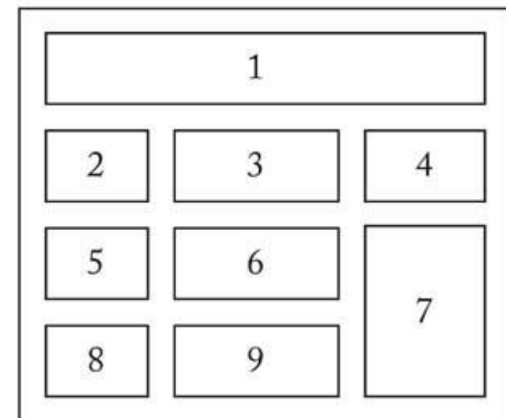
BoxLayout (horizontal)



BorderLayout

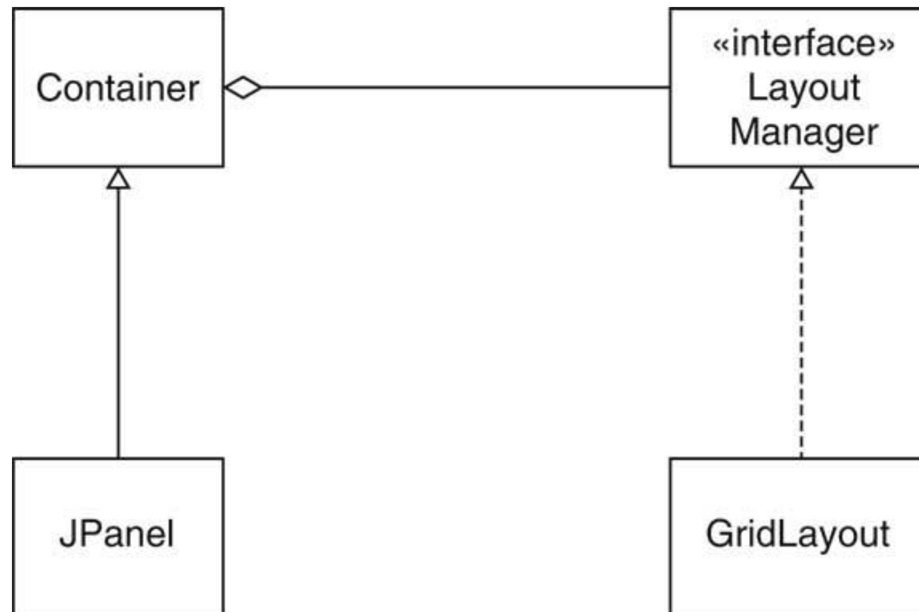


GridLayout



GridBagLayout

Layout Managers



Example: Calculator

Code in Java

```
JPanel keypadPanel = new JPanel();
keypadPanel.setLayout(new BorderLayout());
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
// ...
keypadPanel.add(buttonPanel,
BorderLayout.CENTER);
 JTextField display = new JTextField();
keypadPanel.add(display, BorderLayout.NORTH);
```

Strategy Pattern

- The STRATEGY pattern applies whenever you want to allow a client to supply an algorithm. The pattern tells us to place the essential steps of the algorithm in a strategy interface type. By supplying objects of different classes that implement the strategy interface type, the algorithm can be varied.
  - Pluggable strategy for layout management
  - Layout manager object responsible for executing concrete strategy
  - Generalizes to Strategy Design Pattern

Strategy Pattern

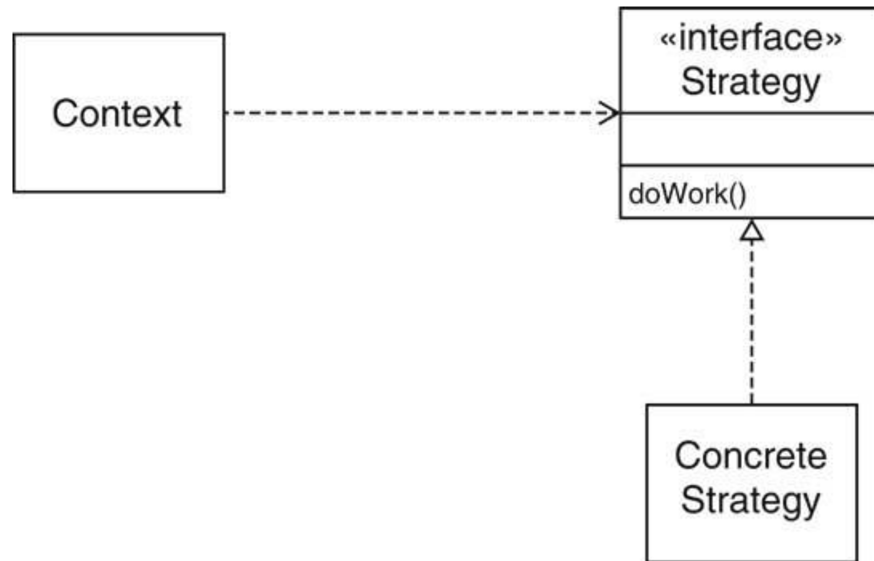
Context

- A class can benefit from different variants for an algorithm
- Clients sometimes want to replace standard algorithms with custom versions

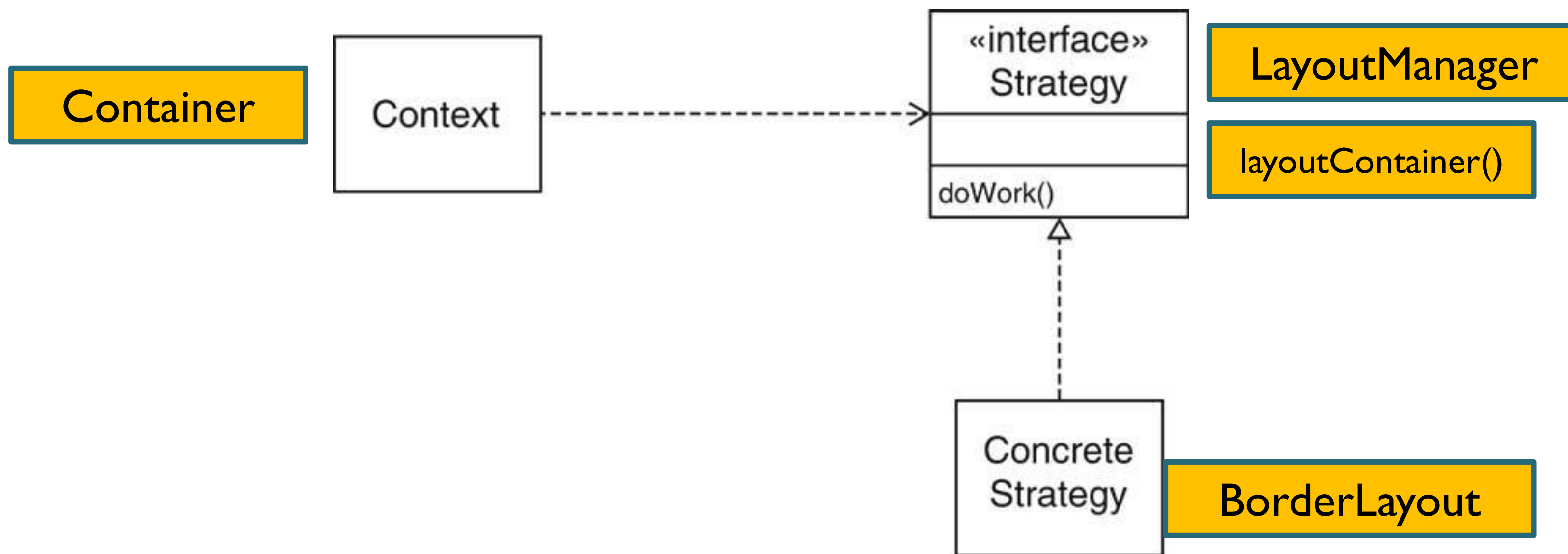
Solution

- Define an interface type that is an abstraction for the algorithm
- Actual strategy classes realize this interface type.
- Clients can supply strategy objects
- Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object

Strategy Pattern

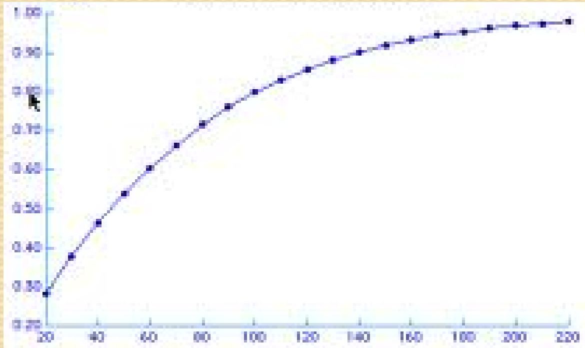


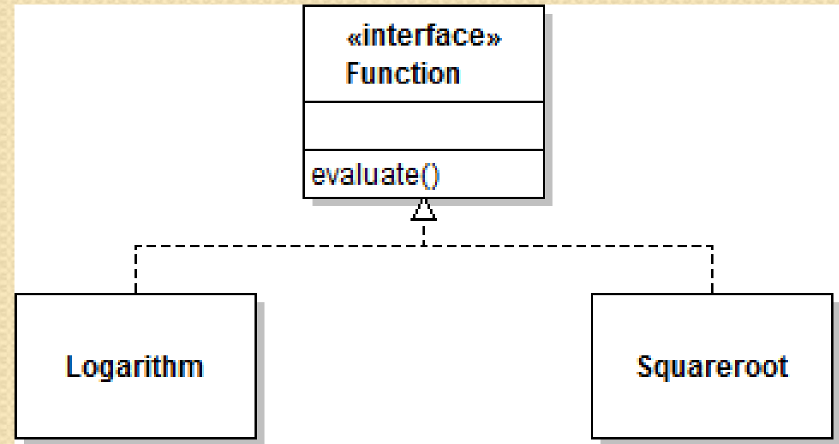
Strategy Pattern



LayoutManager (Java)

```
public interface LayoutManager
{
Dimension minimumLayoutSize (Container parent) ;
Dimension preferredLayoutSize (Container parent);
void layoutContainer (Container parent) ;
void addLayoutComponent(String name, Component
comp) ;
void removeLayoutComponent (Component comp) ;
}
```





QUIZ

Which versions of Graph class use **strategy** pattern?

1. None
2. A
3. B
4. A,B
5. I don't know

A

```
public class Graph {
  private Function f = new Logarithm();
  public draw()
    { plot(1,f.evaluate(1)); … }
}
```

B

```
public class Graph {
  private Function f;
  public Graph(Function fun) { f=fun; }
  public draw()
    { plot(1,f.evaluate(1)); … }
}
```

Design patterns

- Design Patterns
- Iterator
- Model-View-Controller and Observer
- Strategy
- **Composite**
- Decorator

Containers and Components

- Technical problem
  - Containers collect GUI components
  - Sometimes we want to add a container to another container (JPanel to a JFrame)
  - `Container` should *be* a `Component`
- Solution
  - Composite design pattern

The COMPOSITE pattern teaches how to combine several objects into an object that has the same behavior as its parts.

Composite Pattern

Context

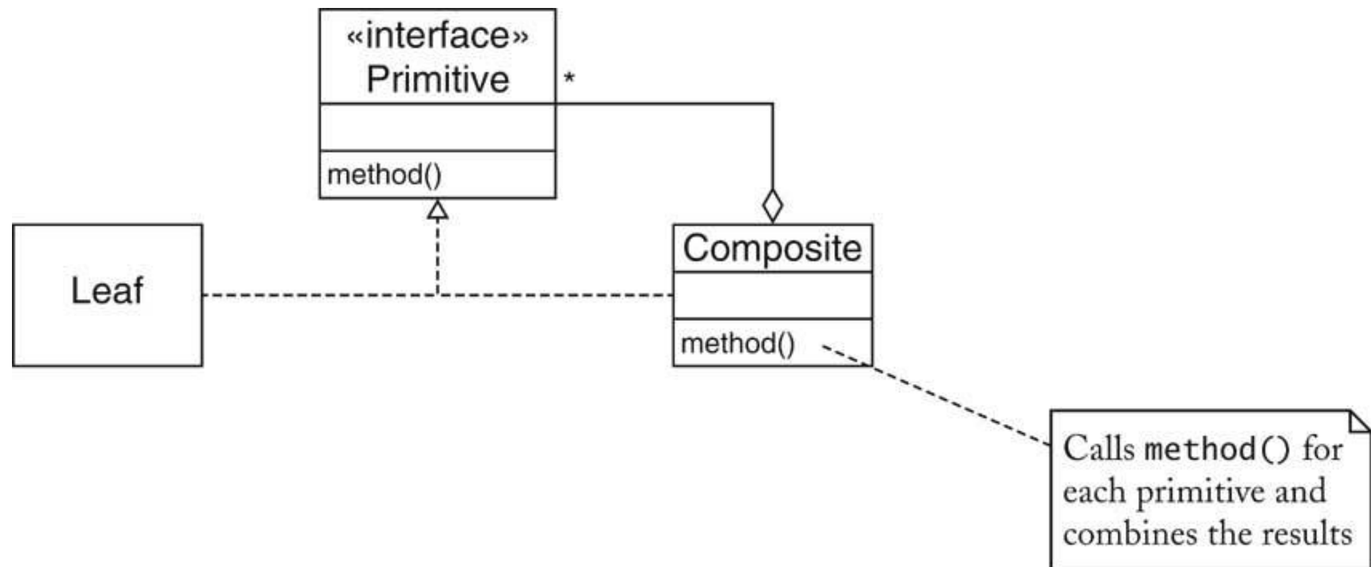
- Primitive objects can be combined to composite objects
- Clients treat a composite object as a primitive object

Solution

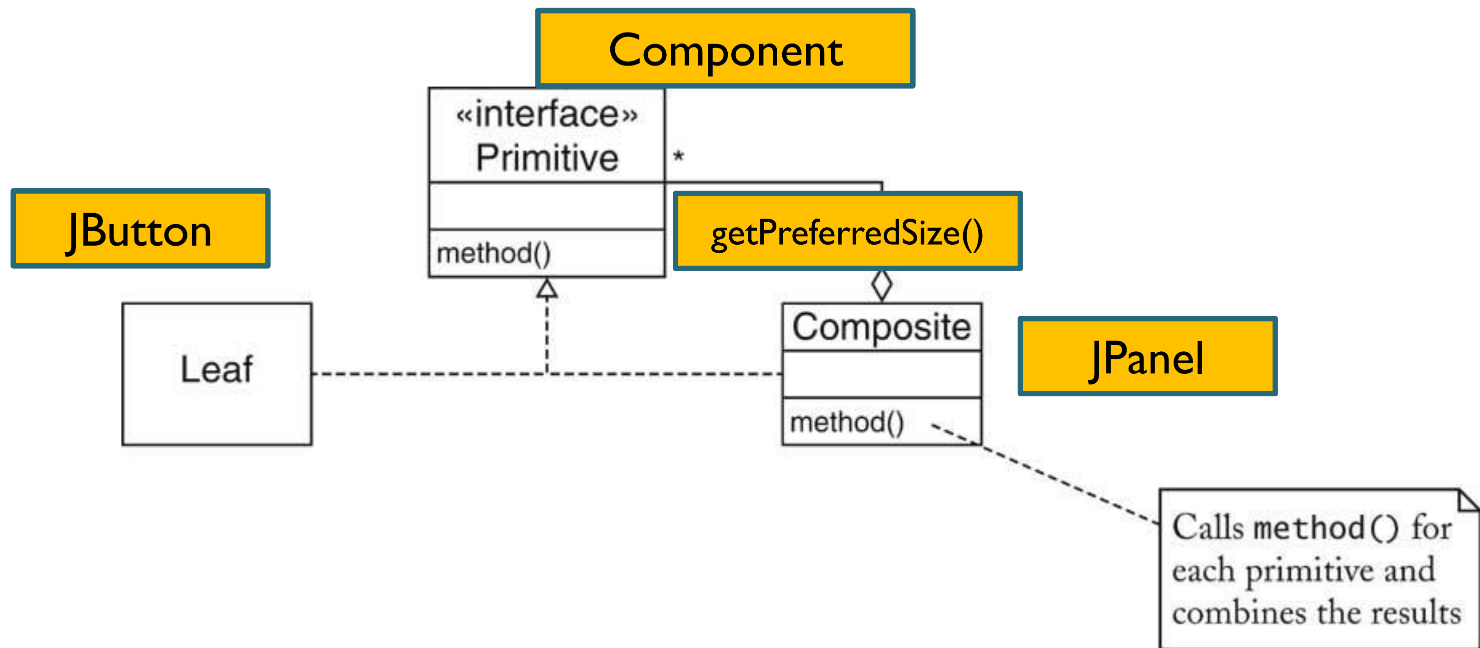
- Define an interface type that is an abstraction for the primitive objects
- Composite object collects primitive objects

- Composite and primitive classes implement same interface type.
- When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results

Composite Pattern



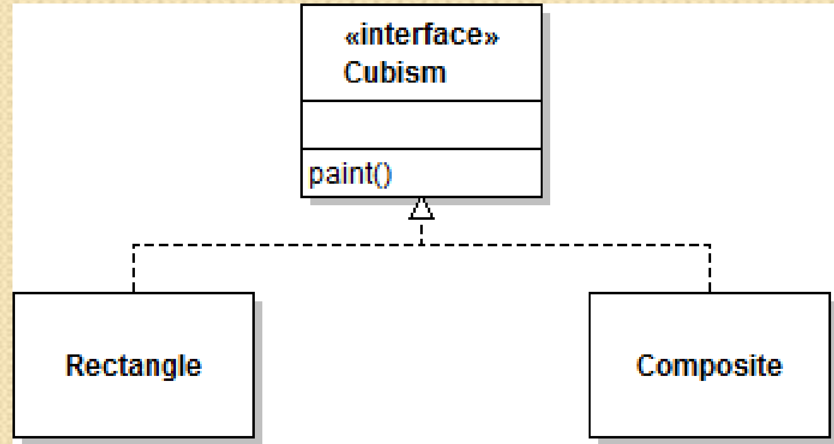
Composite Pattern



QUIZ

Does UML and code for paint methods realize **composite** design pattern?

1. Yes, both UML and code
2. Only UML
3. Only code
4. Neither UML nor code
5. I don't know

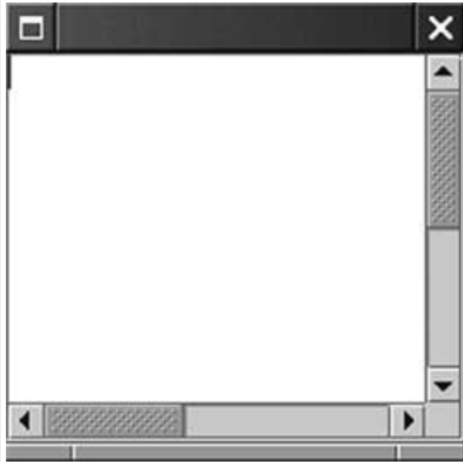


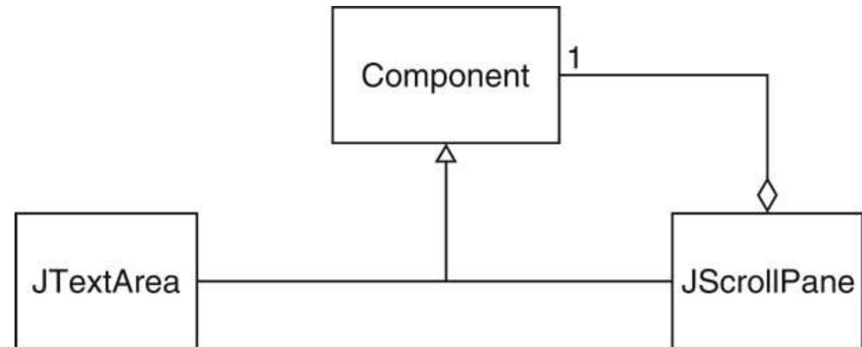

```
class Rectangle:
public void paint(Graphics2D g) {
  g.draw(. . .);
}

class Composite:
private ArrayList<Rectangle> cubes;
public void paint(Graphics2D g) {
  for (Rectangle r: cubes)
    r.paint(g);
}
```

Design patterns

- Design Patterns
- Iterator
- Model-View-Controller and Observer
- Strategy
- Composite
- Decorator

Scroll Bars





- Scroll bars can be attached to components
- Approach #1: Component class can turn on scroll bars
- Approach #2: Scroll bars can surround component
  `JScrollPane pane = new JScrollPane(component);`
- Swing uses approach #2
- `JScrollPane` is again a component

Decorator Pattern

Context

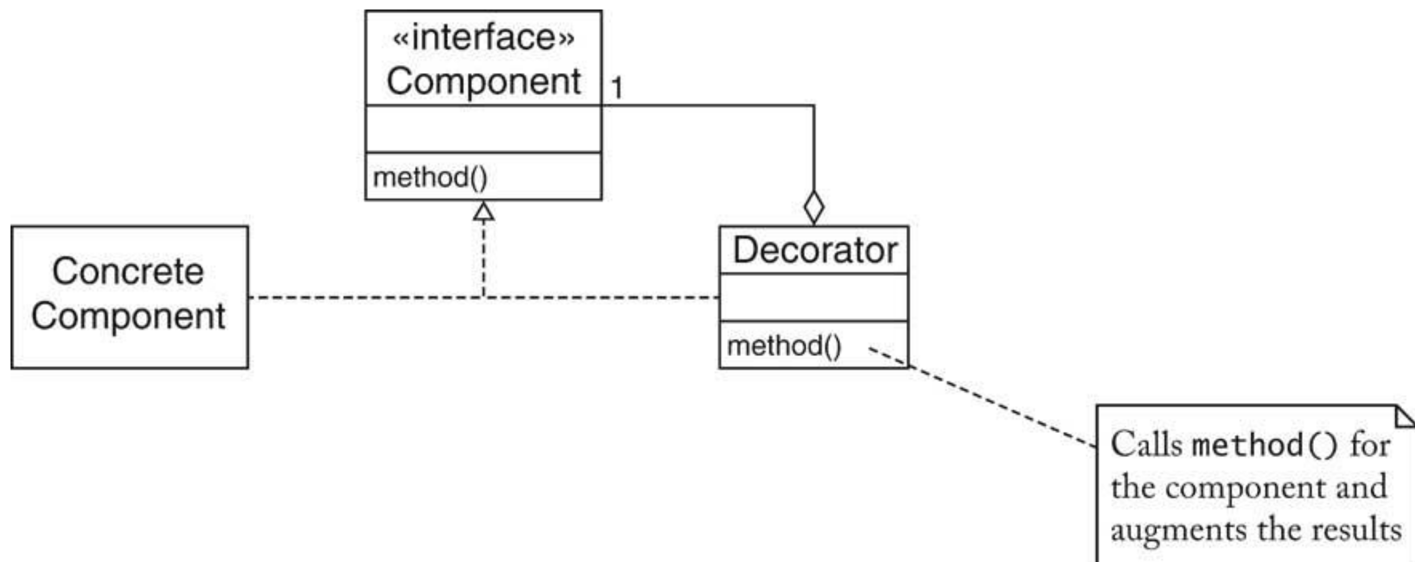
- Component objects can be decorated (visually or behaviorally enhanced)
- The decorated object can be used in the same way as the undecorated object
- The component class does not want to take on the responsibility of the decoration
- There may be an open-ended set of possible decorations

Solution

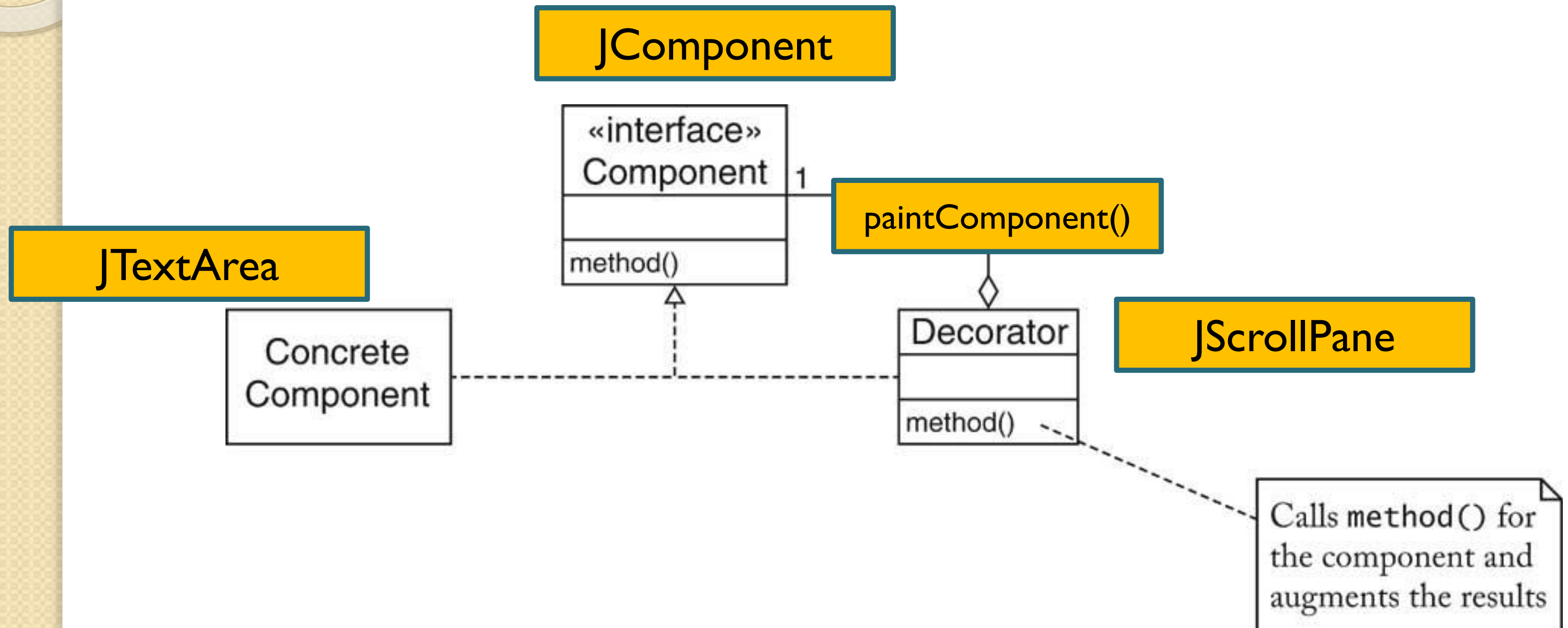
- Define an interface type that is an abstraction for the component
- Concrete component classes realize this interface type.
- Decorator classes also realize this interface type.
- A decorator object manages the component object that it decorates
- When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.

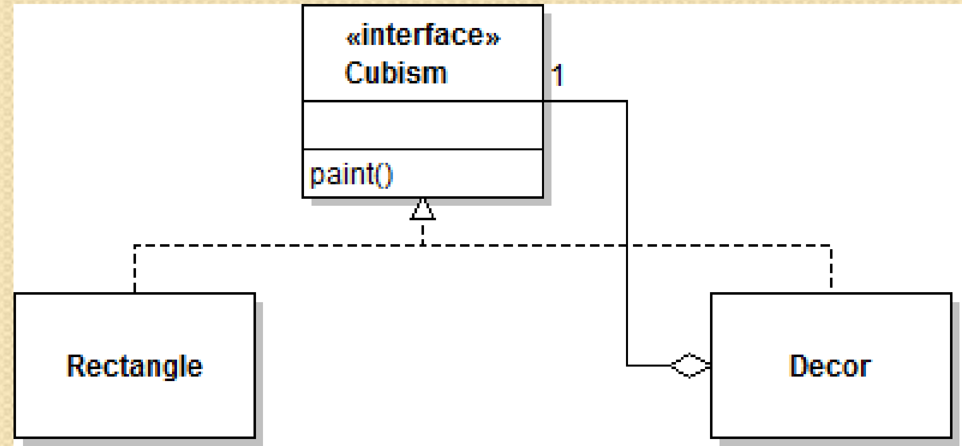
Decorator Pattern

The DECORATOR pattern teaches how to form a class that adds functionality to another class while keeping its interface.



Decorator Pattern





QUIZ

Does UML and code for paint methods realize **decorator** design pattern?

1. Yes, both UML and code
2. Only UML
3. Only code
4. Neither UML nor code
5. I don't know

```
class Rectangle:
public void paint(Graphics2D g) {
  g.draw(. . .);
}
```

```
class Decor:
private Cubism cube;
public void paint(Graphics2D g) {
  g.draw(new Rectangle2D(. . .));
  g.draw(... cube ...);
}
```