# Konzepte objektorientierter Programmierung
## – Lecture 4 –

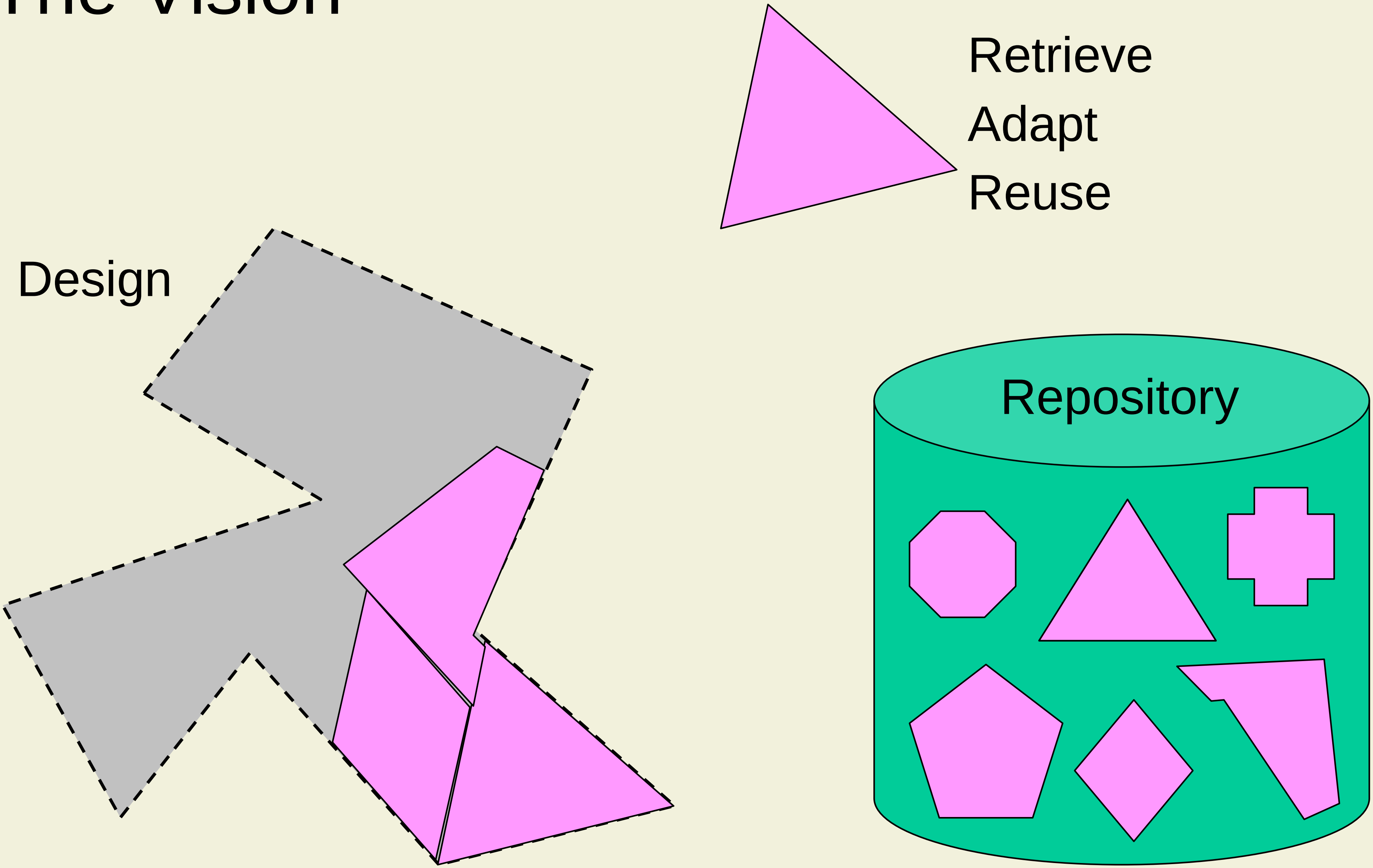
**Prof. Dr. Peter Müller**

Software Component Technology

Wintersemester 05/06

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# The Vision



Retrieve

Adapt

Reuse

Design

Repository

# Levels of Reuse

- Program parts
  - Code
  - Examples: String, LinkedList

Components (reuse in the small)

- Designs
  - Design patterns
  - Examples: Observer pattern, factory pattern
- Software architectures
  - Architectural patterns
  - Examples: Client-server, layered architecture

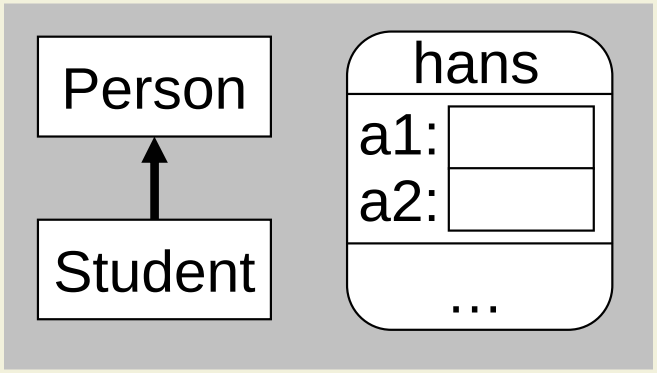
Frameworks (reuse in the large)

# Component

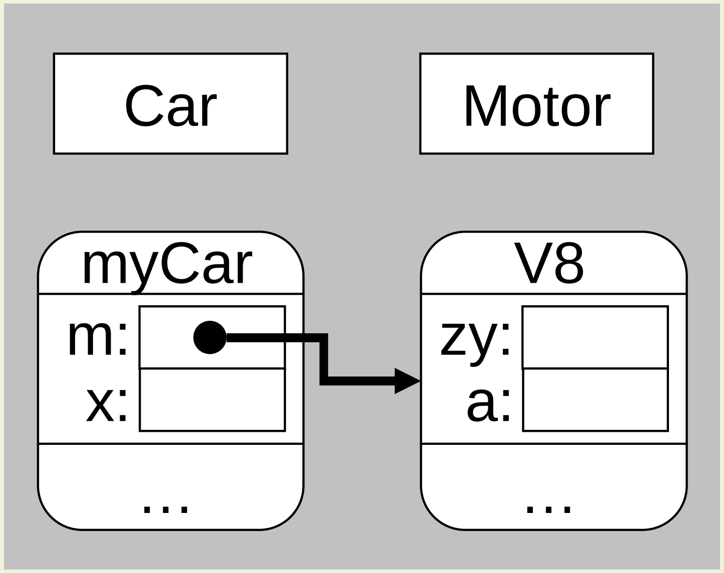
- Definition:
  *An object-oriented component is a group of one or more cooperating classes and interfaces that implement a common abstraction. Components can be reused without further specialization.*
- Examples
  - Simple classes such as String, BigInteger, etc.
  - Groups of classes such as DoublyLinkedList – Node – Iterator
  - But not: The Java Abstract Window Toolkit

# Main Forms of Reuse “in the Small”

- Inheritance
  - Subclassing establishes **“is-a” relation**
  - Enables subtype **polymorphism**
  - Only **one object** at runtime



- Aggregation
  - Establishes **“has-a” relation**
  - **No subtyping** in general
  - **Two objects** at runtime



# Agenda for Today

## 4. Frameworks

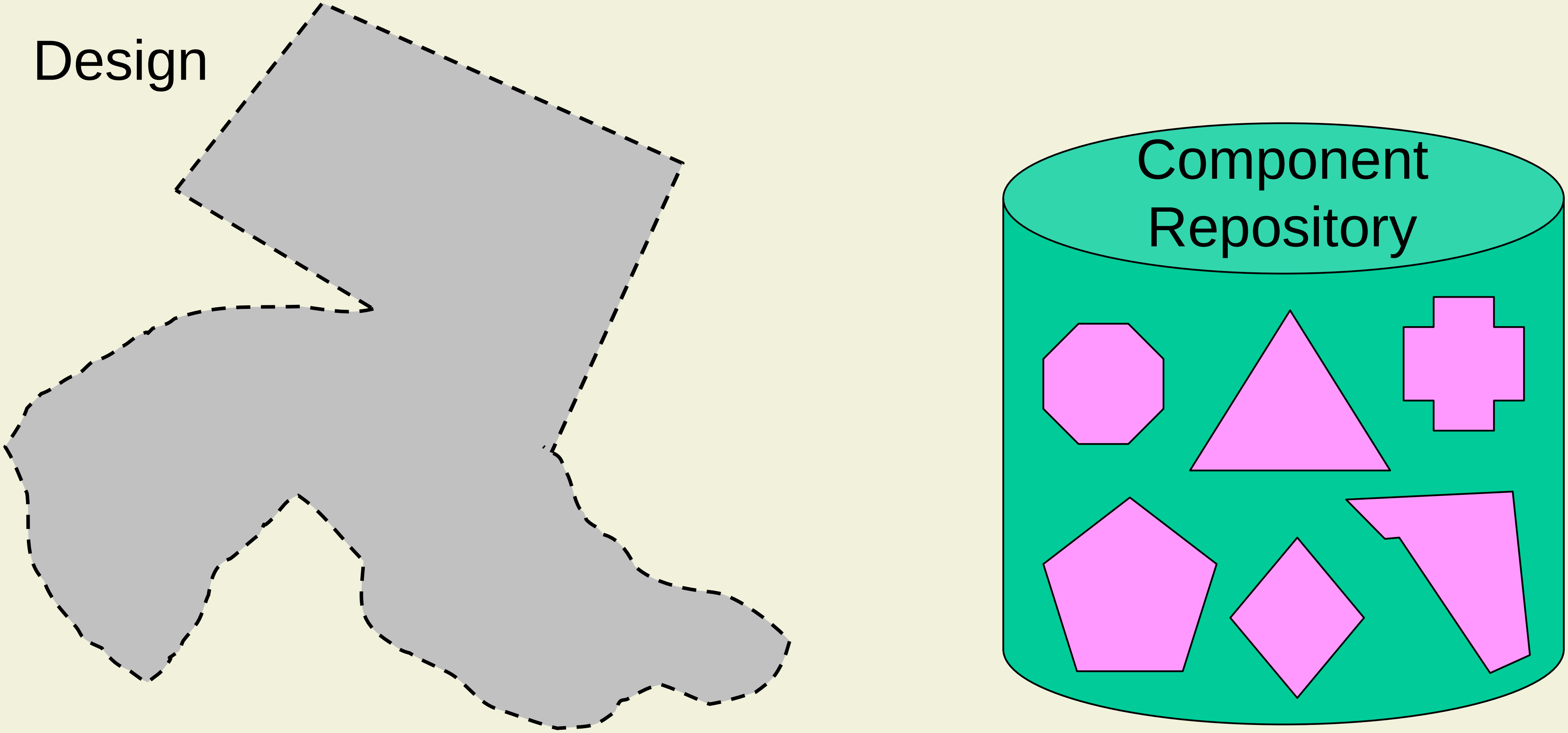
- 4.1 Introduction
- 4.2 Case Study: Java AWT
- 4.3 Events
- 4.4 Reuse in the Large

## Objectives

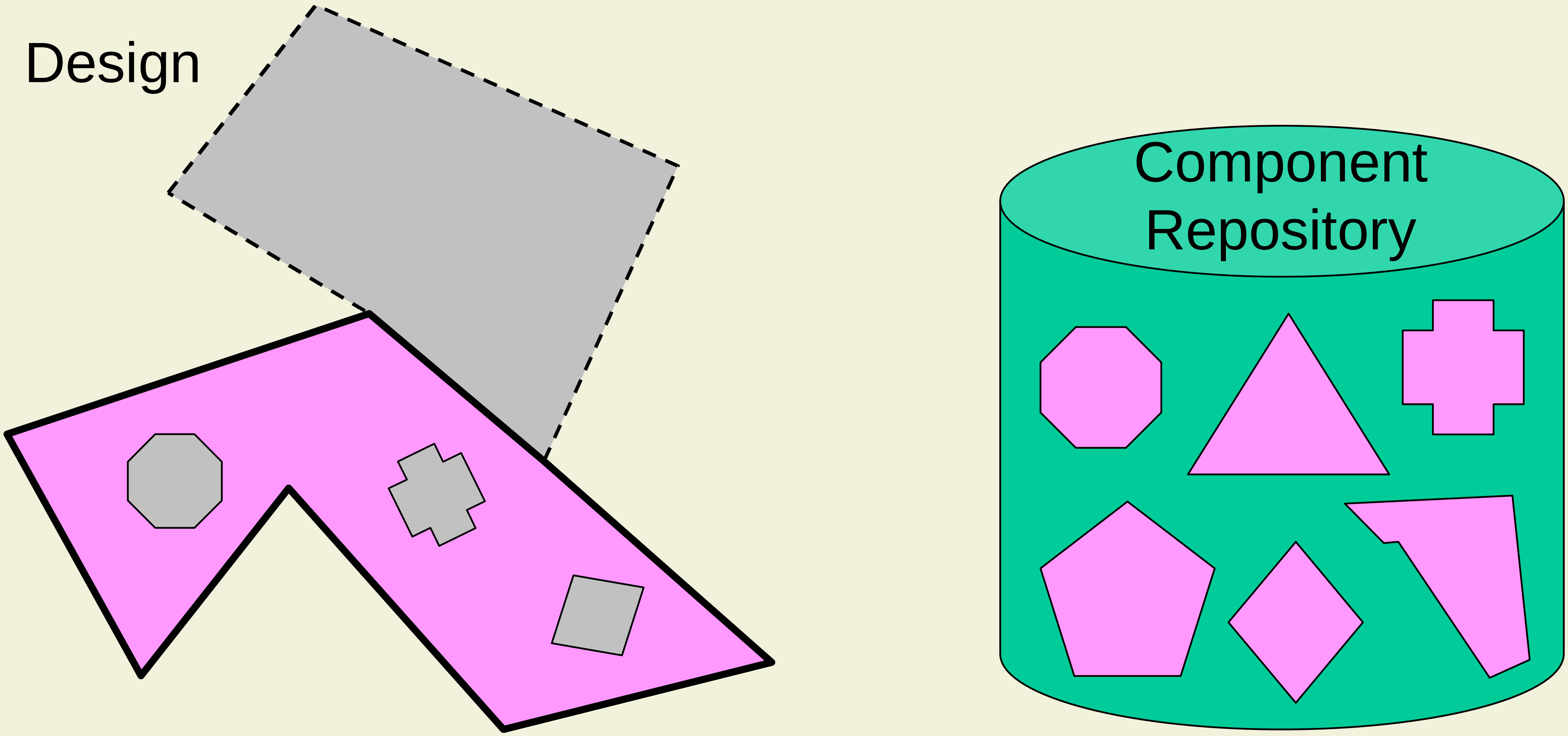
- Event-driven systems
- Reuse of design and architectural patterns

# 4. Frameworks

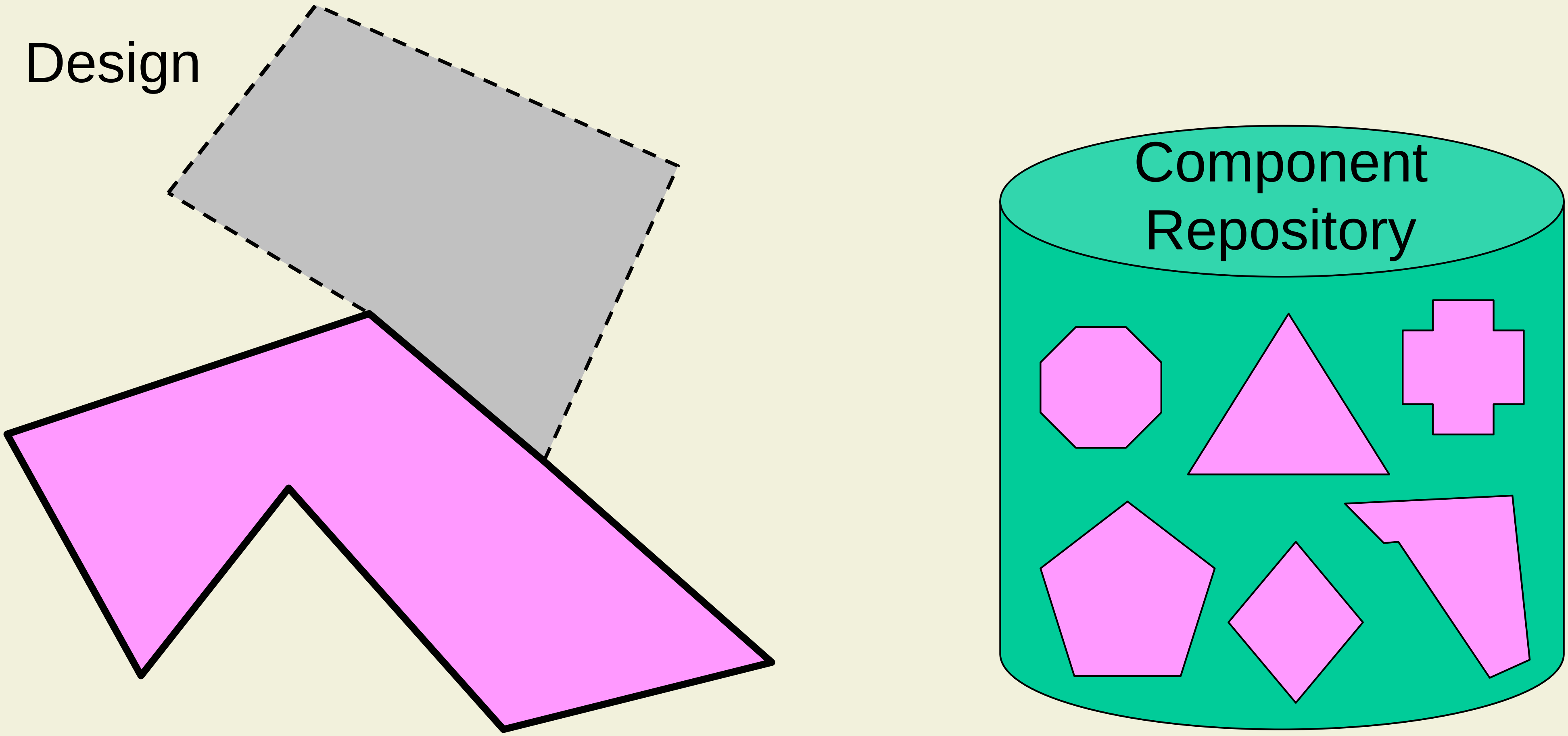
# Reuse in the Large



# Reuse in the Large



# Reuse in the Large



Design

Component Repository

# Framework

- Definition:
  *A Framework is an extendible and adaptable system of classes that provides a core functionality and appropriate components to perform a common superordinate task. The core functionality and the components must simplify variations of tasks of the application area drastically.*

- Frameworks are developed especially for reuse

# Examples and Counterexamples

- Examples
  - GUI frameworks such as Java AWT, Java Swing, etc.
  - Business frameworks such as IBM San Francisco
  - Component frameworks such as Enterprise Java Beans
- Counterexamples
  - Java's Exception hierarchy: Classes do not cooperate to perform a common task
  - java.util: Sets, lists, and iterators work closely together, but do not perform a superordinate task
  - A program: Programs are systems of classes, but not extendible and adaptable

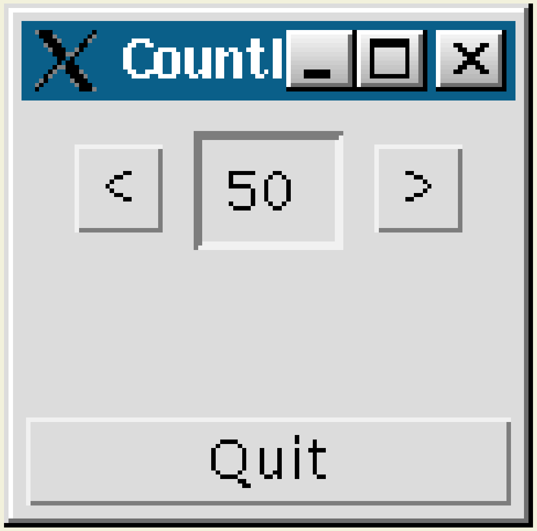
# Characteristics of Frameworks

- Frameworks provide a **core functionality** and abstract from details of a concrete application
- Frameworks can be **incomplete** such that they must be complemented before they are executable programs
- To use a framework, it is sufficient to understand its operation based on an **abstract model**, that is, its key objects and their communication.
- Frameworks often support a special **architectural style**, e.g., a layered architecture

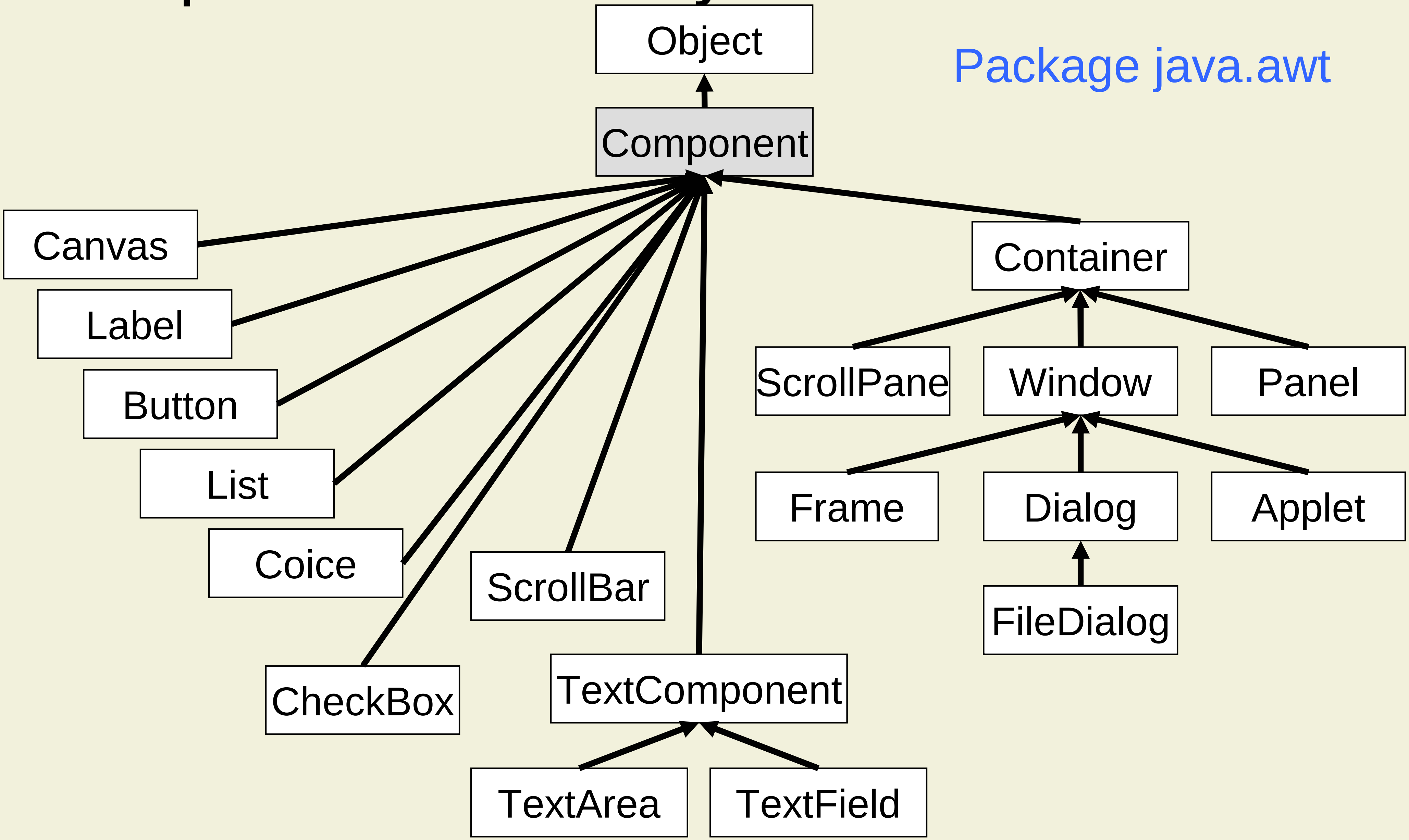
# 4. Frameworks

# AWT: Main Aspects of Abstract Model

- AWT: Abstract Window Toolkit
- Elements of the GUI are represented by **components**
- **Display** and **layout** of the components have to be specified
- Components receive **events** from the window system and propagate them to so-called **listeners**

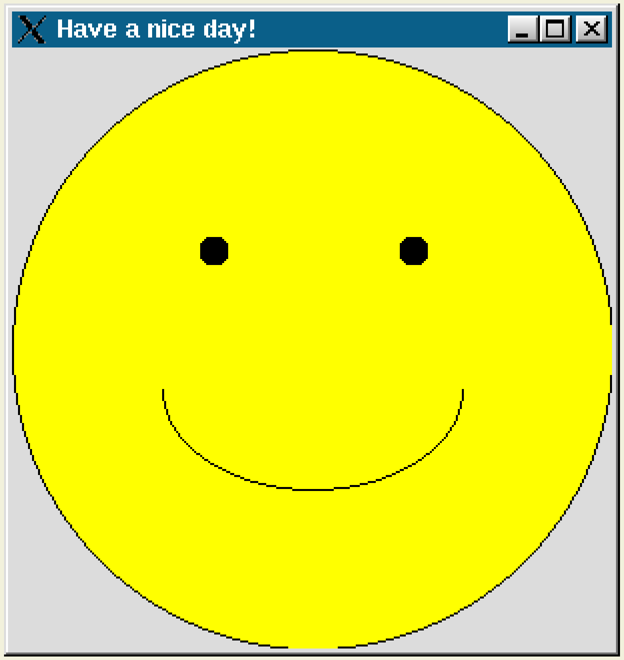


# Component Hierarchy



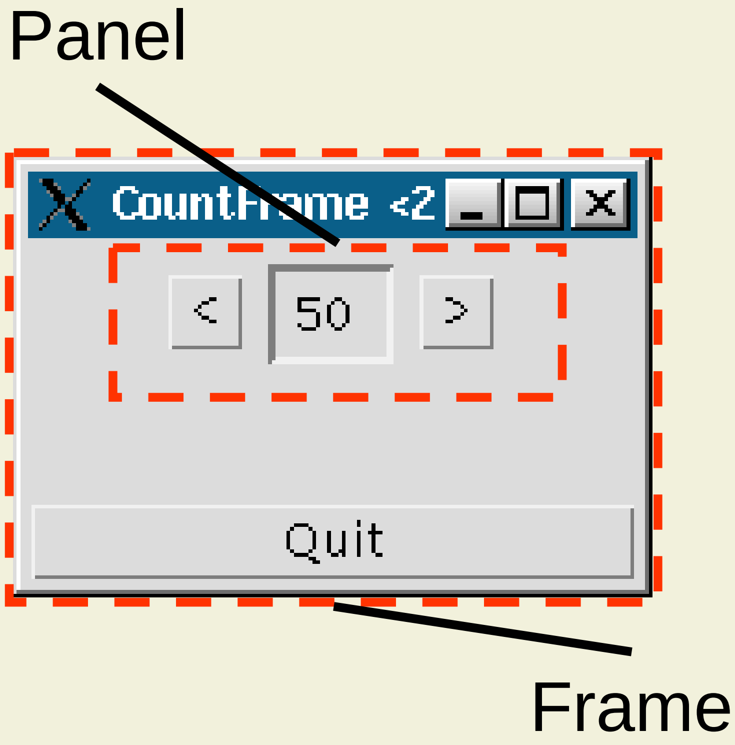
# Displaying Components

- Components are displayed in rectangular areas
- Appearance is pre-defined for each component
- Developers can set parameters such as fonts, colors, size, etc.
- Each component has a Graphics-object to perform drawing
- Method paint displays component and can be overridden

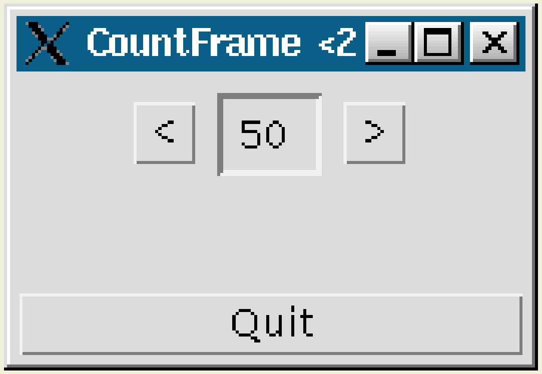


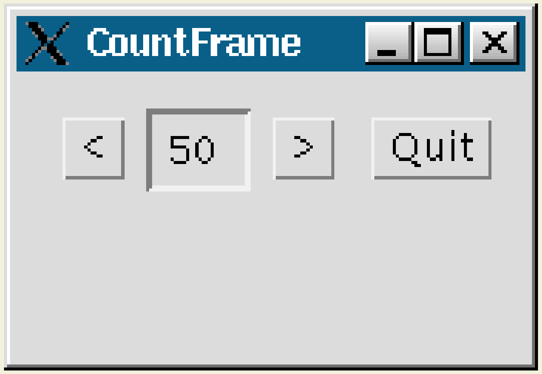
# Displaying Containers: Layout Managers

- Components can be grouped into containers
- Layout of components in one container is computed by a layout manager
- The layout manager can be set for each container



Border-Layout



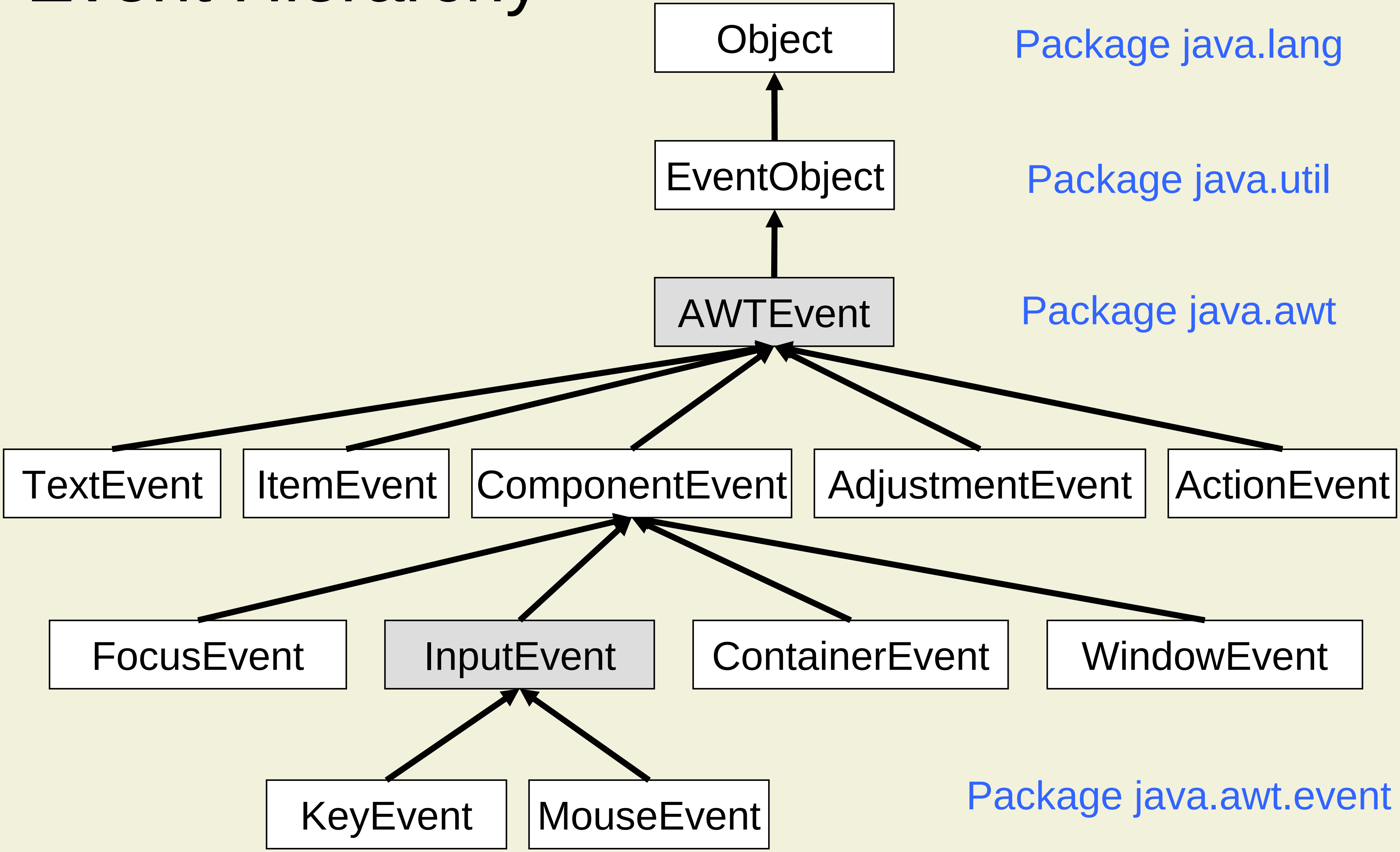


Flow-Layout

# Events in the AWT

- User actions **create events**, e.g., mouse clicks or key-strokes
- The window system **assigns a component** to each event (the so-called **event source**)
- Low-level events
  - Grouped into event types: MouseEvent, KeyEvent, etc.
  - mousePressed, mouseReleased, mouseClicked, etc.
- Semantic events
  - Combinations of low-level events
  - actionPerformed, textValueChanged, etc.

# Event Hierarchy



Package java.lang

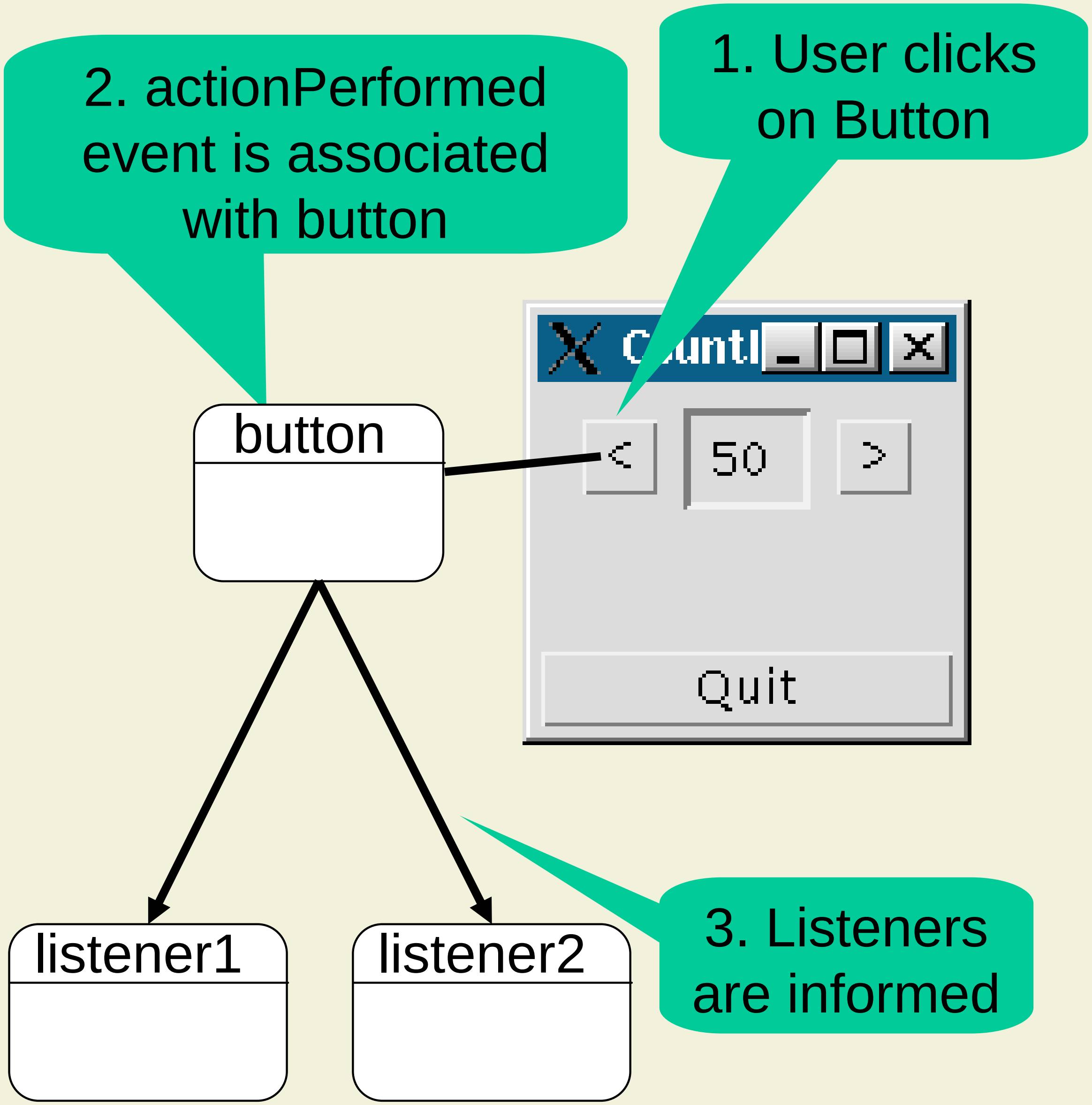
Package java.util

Package java.awt

Package java.awt.event

# Event Control: Basic Concept in the AWT

- Objects can register at a component as observer (listener) for one or several event types
- Upon occurrence of an event, the event source informs all registered objects by invoking a method


1. User clicks on Button
2. actionPerformed event is associated with button
button
<
50
>
Quit
listener1
listener2
3. Listeners are informed

# AWT as Framework

- The AWT consists of **12 packages** and **more than 100 classes**
  - Substantial reuse in the small
  - Inheritance (Component has more than 100 methods)
  - Aggregation (layout manager, fonts, etc.)
- Classes cooperate closely, that is, form a **system**
  - Mutually recursive types
- **Common superordinate task** is the development of GUIs
- System is **extensible** and **adaptable**

# Adaptations

- The AWT applies the standard OO-concepts to enable adaptations
- Specialization (Inheritance)
  - Overriding paint method to change presentation
- Polymorphism (Aggregation)
  - Exchanging layout managers of containers
- Parameterization
  - Changing the state of Component-objects, e.g., the size
- Event-Communication
  - Connecting application logic and GUI via listeners

# 4. Frameworks

4.1 Introduction

4.2 Case Study: Java AWT

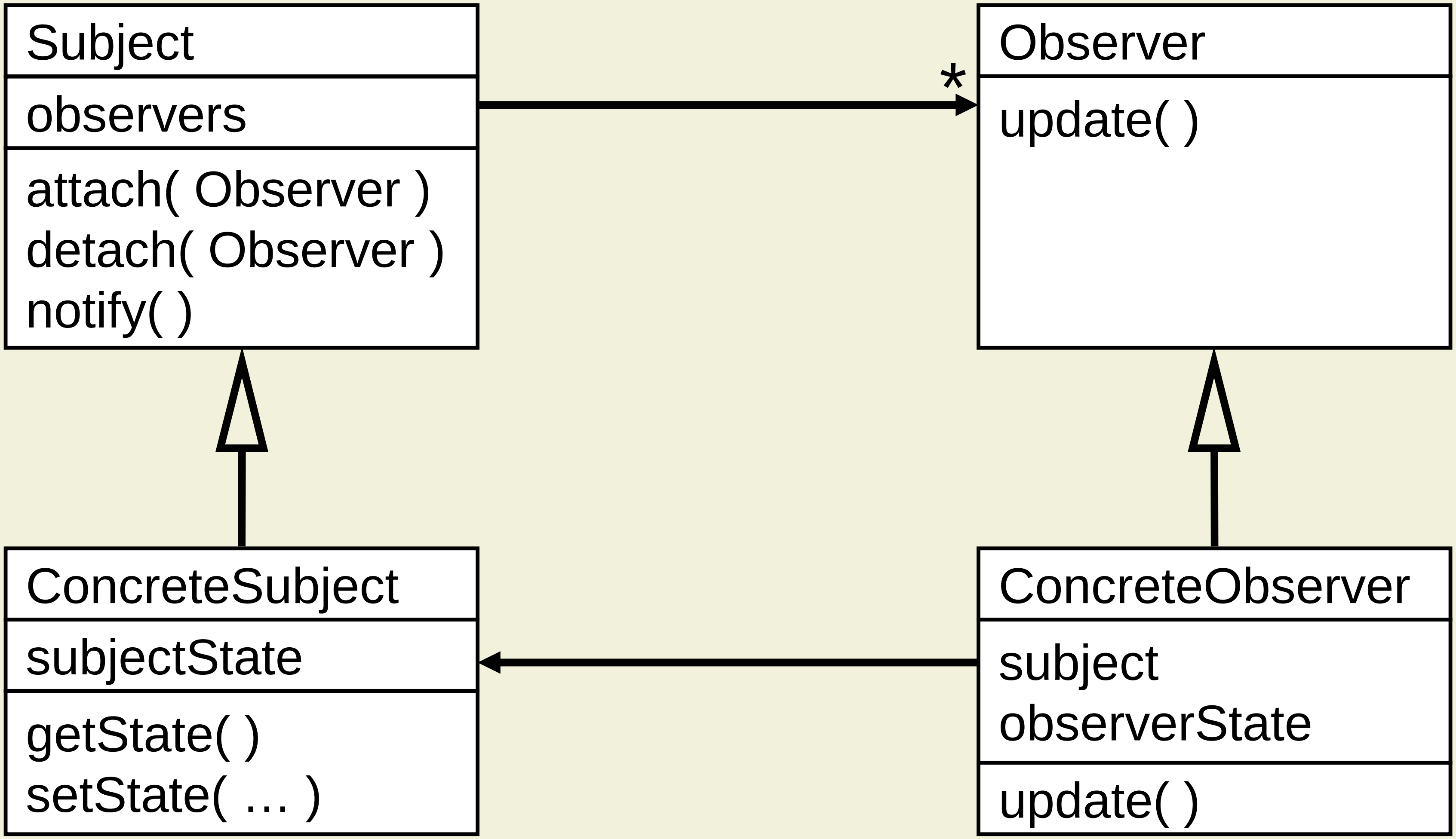
**4.3 Events**

4.4 Reuse in the Large

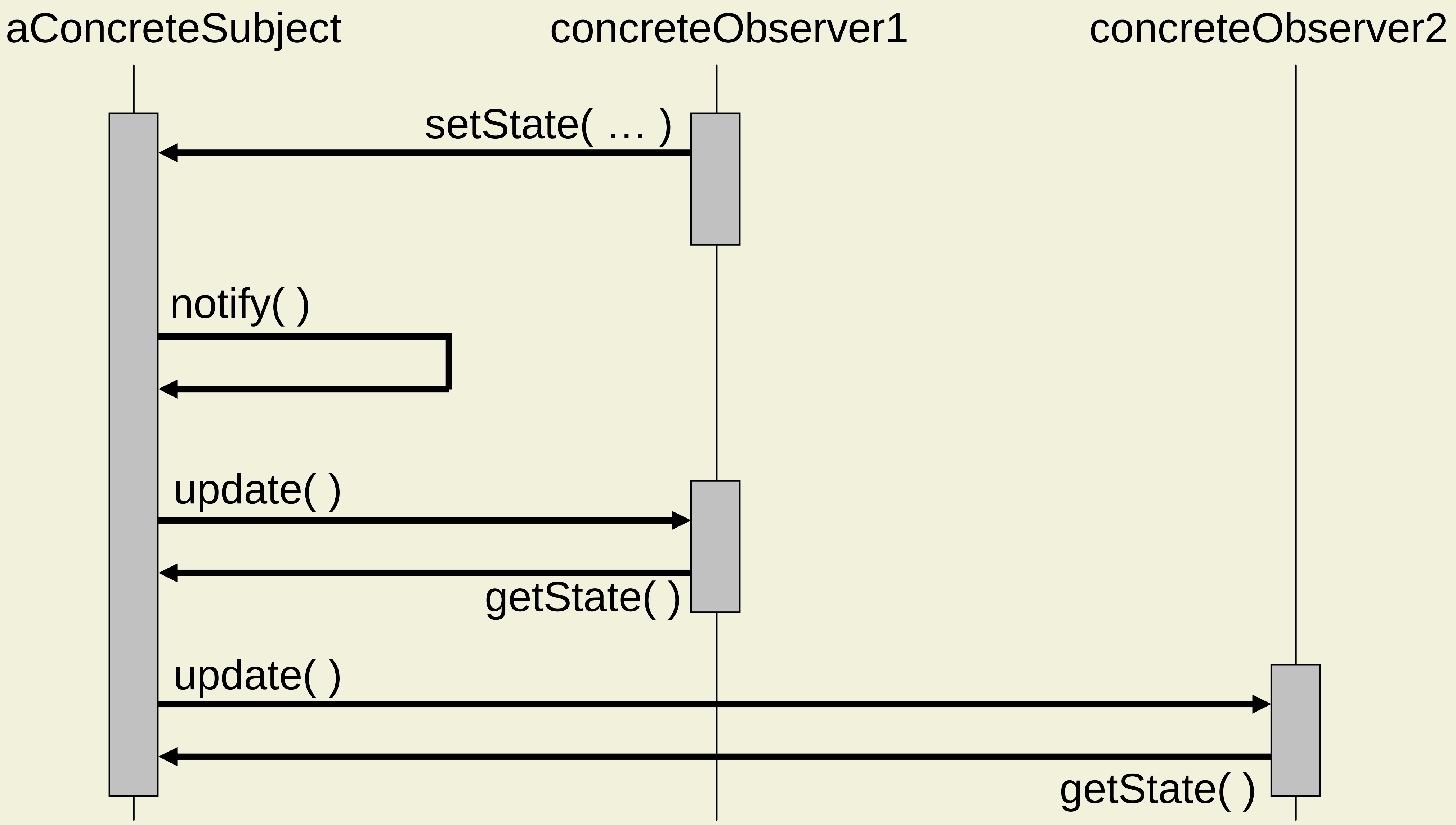
# References to Methods

- Event-handling
  - Event source invokes registered listener method
  - Event source needs list of listener methods
- Common OO-Solution
  - Use naming convention for listener methods
  - Define interface with the listener method
  - Listener objects implement the interface
  - Event source maintains list of listener objects (polymorphism)
  - Events are dispatched by invoking the pre-define method on each listener object

# Observer Pattern



# Collaborations



# Observers in the AWT: Listeners

- Requirements
  - Different types of events
  - Selective registration for event type
  - Events are not initiated by observers

- Event types are associated with method signatures
- Observers (listeners) have to implement corresponding interface

```
public interface ActionListener
                        extends EventListener {
  public void
        actionPerformed( ActionEvent e );
}
```

# Subjects in the AWT: Compo

Components maintain lists of listeners for each event type

**class** Button:

```
… void addActionListener( ActionListener l ) {
  actionListener = AWTEventMulticaster.add( actionListener, l );
  newEventsOnly = true;
}
```

Events are triggered by the window system

```
protected void processActionEvent( ActionEvent e ) {
  if ( actionListener != null )
    actionListener.actionPerformed( e );
}
```

Event is dispatched to each registered listener

# Event-Handling: Other Solutions

- Smalltalk
  - Built-in functionality (inherited from class Object)
  - Each object maintains list of dependent objects
  - Each object has methods changed, update, broadcast
- Eiffel
  - Powerful agent mechanism
  - References to methods can be passed as arguments
  - No need for naming conventions and Observer interface
  - EVENT library for events that have state

# Characteristics of Event Communication

- Triggering an event causes **implicit invocation**
- Event sources in general do not know
  - **Which objects** will be affected
  - **In which order** events are dispatched
  - **What processing** will occur as a result of an event

```
class Subject {
  Observer[ ] observers;
  …

  // requires true
  // ensures true
  void notify( ) {
    foreach o ∊ observers
      o.update( );
  }
}
```

# 4. Frameworks

4.1 Introduction

4.2 Case Study: Java AWT

4.3 Events

**4.4 Reuse in the Large**

- **Design Patterns**
- Architectural Patterns

# Reuse in the Large: Example

- Software development environment
- Components
  - Debugger: Reusable library component
  - Editor: Newly developed or reused
- Collaboration
  - When the debugger reaches a breakpoint, the editor shows the corresponding part of the source code

# Solution with Aggregation

- Debugger **"knows its" editor** (i.e., has a reference to editor)
- Editors have to **implement** a certain **interface**
- Debugger invokes **appropriate method** of editor

```
interface Editor {
  void showContext( … );
}
```

```
class Debugger {
  Editor editor;
  …
  void processBreakPoint( … ) {
    …
    editor.showContext( … );
  }
}
```

```
class Emacs implements Editor {
  void showContext( … ) { … }
}
```

# Adaptation: Add StackViewer

- New requirement: Stack trace should be displayed when breakpoint is reached
- Debugger can be adapted by **subclassing** and **overriding** method processBreakPoint

```
class StackViewer {
  …
  void showStackTrace( … )
    { … }
}
```

```
class MyDebugger
                extends Debugger {
  StackViewer sv;
  …
  void processBreakPoint( … ) {
    super.processBreakPoint( … );
    sv.showStackTrace( … );
  }
}
```

# Solution with Event-Control

- Debugger has a **generic list of observers**
- Debugger **triggers event** when breakpoint is reached
- Observers decide how to handle this event (**no control by debugger**)

```
class Debugger extends Subject {
  …
  void processBreakPoint( … ) {
    …
    notify( … );
  }
}
```

```
class Emacs
                implements Observer {
  void showContext( … ) { … }
  void update ( … ) {
    showContext( … );
  }
}
```

# Adaptation: Add StackViewer

- New requirement: Stack trace should be displayed when breakpoint is reached
- StackViewer is just another observer
- **Debugger** does **not** have to be **adapted**

```
class StackViewer
        implements Observer {
  …
  void showStackTrace( … )
   { … }

  void update ( … ) {
   showStackTrace( … );
  }
}
```

# Aggregation vs. Event Communication

## Aggregation

- Caller has **full control over computation**
- Caller **knows order** of invocations
- Reasoning about **correctness** is easier (contracts)

## Event-Control

- Strong support for **reuse in the large**: **Components can be introduced** by simple registration
- Support for **evolution**: **Component can be replaced by other components** without affecting interfaces of other components

# 4. Frameworks

4.1 Introduction

4.2 Case Study: Java AWT

4.3 Events

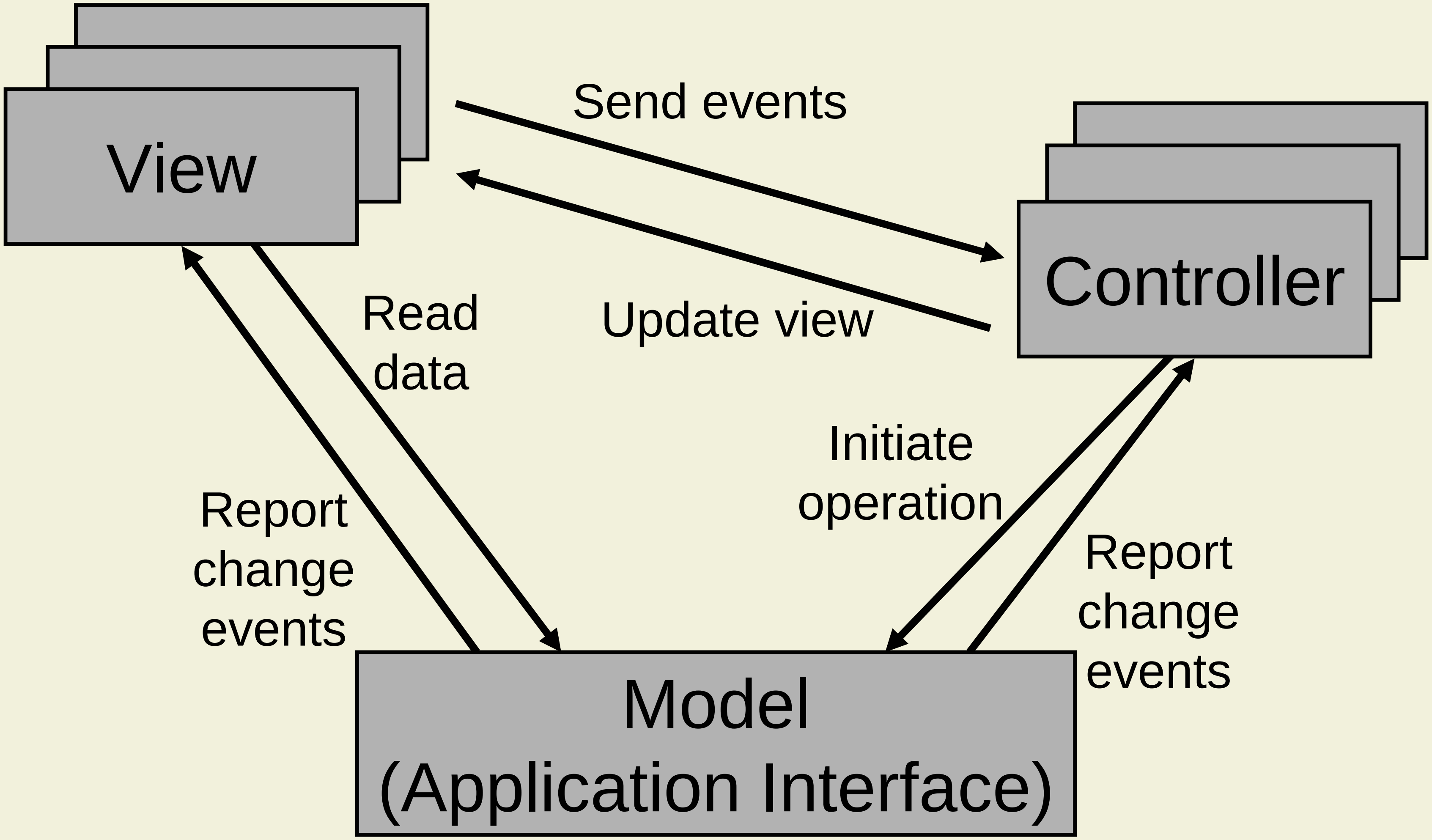
**4.4 Reuse in the Large**

- Design Patterns
- **Architectural Patterns**

# Software Architecture

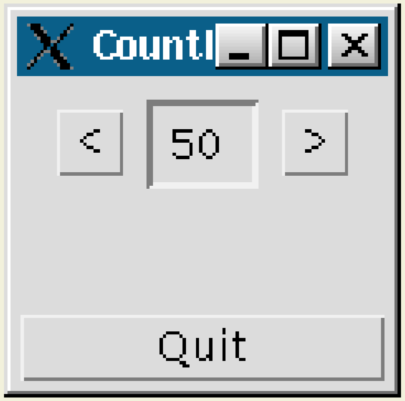
- Definition:
  *The architecture of a software system defines that system in terms of computational components and interactions among those components.*
  [Shaw, Garlan: Software Architecture]
- Components: Clients and servers, databases, filters, layers in a hierarchical system, etc.
- Interactions: Procedure call, shared variable access, client-server protocols, event multicast, etc.

# Model-View-Controller Architecture



# AWT supports MVC

- Model: Application interface
- View: AWT-components
- Controller: Listeners



```
class Counter {
  int value = 0;
  void increment( )     { value++; }
  void decrement( )     { value--; }
  String getValue( ) { return value; }
}
```

```
class Button extends Component {
  …
}
```

```
class incrListener
        implements ActionListener {
  Counter counter;
  Textfield tf;

  void actionPerformed
                  ( ActionEvent e ) {
    model.increment( );
    tf.setText( counter.getValue( ) );
  }
}
```

# Reuse with Frameworks: Summary

- Build on component reuse
  - E.g., component hierarchy in AWT
- Support reuse in the large
  - Designs (e.g., observer pattern in AWT)
  - Architectural patterns (e.g., MVC architecture in AWT)
  - Often through event communication
- Adaptation
  - Specialization
  - Polymorphism
  - Parameterization
  - Event-Communication