7. Program Frameworks

Overview:

7.1 Introduction to program frameworks

7.2 Program frameworks for User Interfaces:

- Architectural properties of GUIs
- Abstract Window Toolkit of Java

Many software systems

- have similar subtasks,
- use common communication mechanisms,
- share components,
- are based on the same architectural pattern.

For example, many software systems

- have a graphical user interface,
- communicate via remote procedure call,
- have applications that share components,
- are based on the three-tier architectural pattern.

If systems share properties, there is a chance to reuse

- architectural and design knowledge,
- implementation techniques,
- implementation parts, and
- communication infrastructure.

An important method to support reuse is the construction of frameworks that capture the shared properties.

7.1 Introduction to Program Frameworks

- Program frameworks are an important technique for software reuse.

- They are usually developed for certain tasks or domains.

- From a static point of view, a program framework is an *extensible* and *adaptable* system of *modules* or classes that implement the core functionality and communication mechanisms of a general task or domain.

Program frameworks are written in some programming language (e.g. C, C++, or Java).

Depending on the language, the elements of a program framework are

- procedures,
- modules,
- classes,
- interfaces,
- or similar programming elements.

In the following, we give answers to two questions:

1. What is the distinction between a program framework and an arbitrary set of library classes?

2. What is the distinction between program frameworks and other kinds of frameworks?

Both answers will help to understand the relation between architectural properties and program frameworks.

Program Frameworks and Library Classes

Explanation: (Program framework)

A program framework is a system of classes which:

- work together to realize the functionality of more complex parts of software systems
- are extensible and adaptable
- provide the core functionality to perform a common general task that is needed within different concrete systems (generic aspect)

Syntactically, an object-oriented program framework appears to the programmer as a set of (library) classes.

But:
Not every set of (library) classes is a program framework, e.g.:

- a set of independent, unrelated classes like the Java library classes String, Boolean, Long, Float, etc.
- classical application programming interfaces that allow to access services of the operating or network system

Remark:

Often, a program framework is based on an architectural pattern. E.g. many GUI toolkits are designed with the “model-view-controller pattern” in mind.

More powerful program frameworks are so-called *application frameworks*. They capture the generic architecture of families of applications. E.g. they support uniform user-guidance and common appearance of different applications in the family (cf. the article “ET++ – a Portable, Homogeneous Class Library and Application Framework”).

Other program frameworks are less general and realize generic architectures for certain application domains.

Discussion of Program Frameworks

Program frameworks capture generic architectures and generic implementations in form of (library) classes written in some programming language.

This has two *disadvantages*:

1. Program frameworks rely on whitebox reuse.
2. They are tied to the mechanisms and limitations of the underlying programming language.

In the following, we explain these disadvantages and discuss possible alternatives or extensions.

Whitebox Reuse

Applications that are developed with the help of a program framework are often constructed by specializing the classes of the framework.

Explanation: (Whitebox reuse)

Whitebox reuse means that the software developer has and needs to have access to the program parts to be reused.

Disadvantage of whitebox reuse:

The provider of the framework cannot change the implementation of the framework without affecting existing applications developed with it.

Possible solutions:

- Graybox approach:
  - develop and use better encapsulation and interface specification techniques

- Blackbox components:
  - capture the generic architecture that should be supported
  - are fully encapsulated and can only be accessed through a well-defined interface
  - have a hidden implementation
  - can be given in a binary format (programming language independent

Programming Language Limitations

Program frameworks share the inherent limitations of using one fixed language.

The construction of systems running on multiple platforms is limited by:

- Interprocess communication and communication with programs written in different languages.
- Support for configuration and deployment of system parts.
- Component compatibility and versioning.
- Support for persistent data in a programming language independent and portable way.

7.2 Program Framework for User Interfaces

Overview:

- Architectural Properties of Graphical User Interfaces
- The Abstract Window Toolkit of Java

Architectural Properties of Graphical User Interfaces

A graphical user interface should enable a user-friendly and flexible interaction between an application and its users.

Tasks of a GUI:

1. It should allow to control the application.
2. It should allow to enter data for the application.
3. It should enable to present the state of the application and the results of operations performed by the application in a graphical way.

A program framework for GUIs has to provide:

- GUI components (e.g. buttons, text fields, ...),
- techniques to show and arrange the GUI components on the screen (e.g. to layout several buttons in an area of a window),
- a mechanism to link operations of the application kernel to the actions performed on the GUI (e.g. the mouse click on a button has to the operation that should be performed in reaction to the click).
- an execution model for simultaneous user actions.

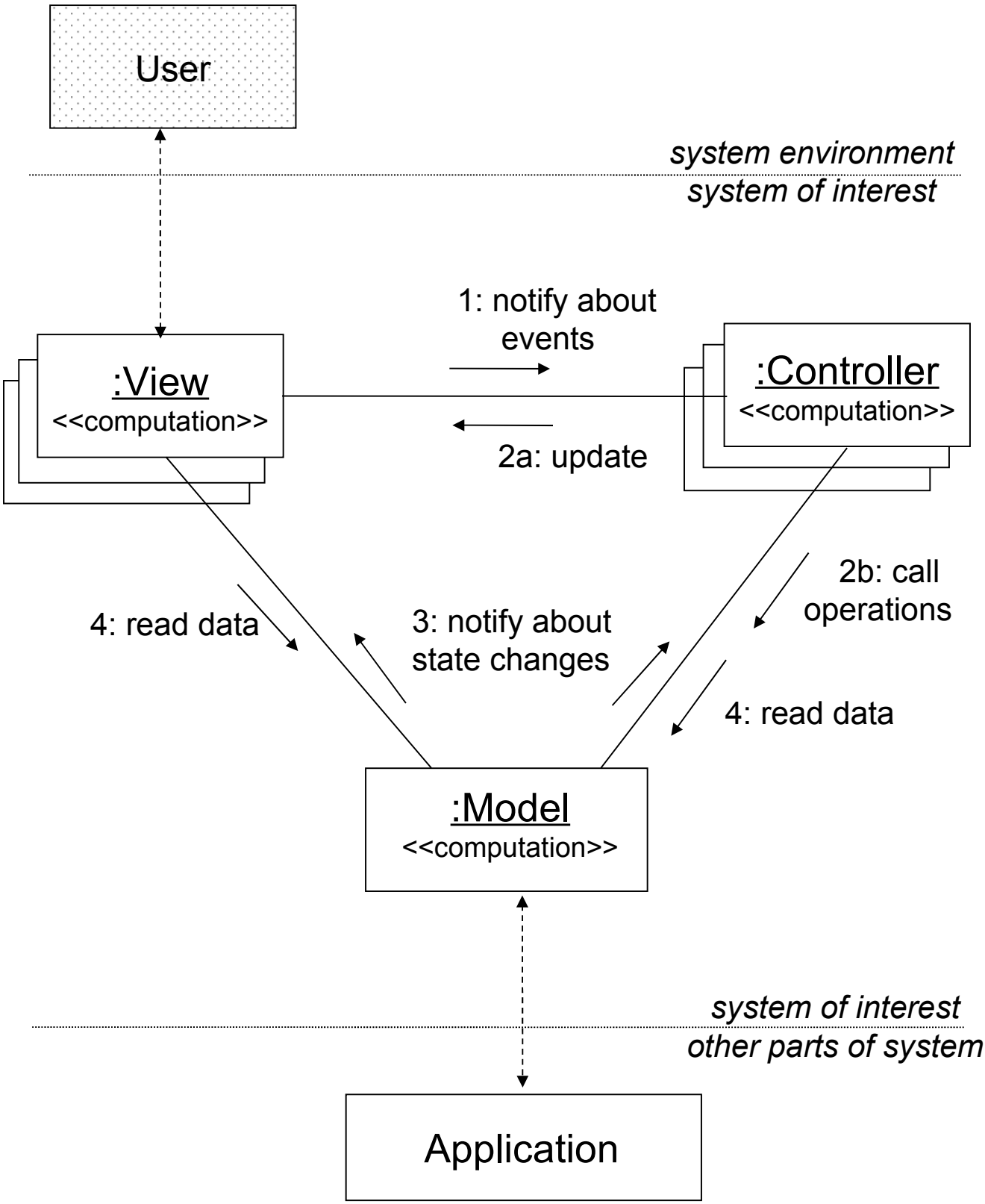
MVC-Pattern

The classical pattern underlying graphical user interfaces is the model-view-controller pattern or MVC-pattern.
It distinguishes three kinds of components:

- The *model* represents the underlying application kernel and captures the state of the GUI. It provides the interface to the core functionality of the application.

- The *views* display the model or part of it on the screen. Thereby the model becomes accessible to the user.

- The controllers control the interaction between users and the model.


User
system environment
system of interest
1: notify about events
:View
<<computation>>
:Controller
<<computation>>
2a: update
2b: call operations
4: read data
3: notify about state changes
4: read data
:Model
<<computation>>
system of interest
other parts of system
Application

The MVC-pattern is not fixed.

Variations can e.g. affect

- the relation between views and controllers:
  (a) Is there always a one-to-one mapping?
  (b) Can there be a controller that is associated with several views?
- or pertain to the execution model.

For the purpose of this chapter, two things are important:

1. The construction of GUIs involves
   - the management of views
   - the management of the consistency between models and views,
   - and a mechanism to handle the possibly concurrent events

2. It should have become clear how the pattern structures the core functionality of a GUI into different system components.

The Abstract Window Toolkit of Java

The Abstract Window Toolkit, AWT for short, is the basic program framework that Java provides for the implementation of GUIs (the Swing toolkit is an extension of it).

It consists of more than a hundred classes.

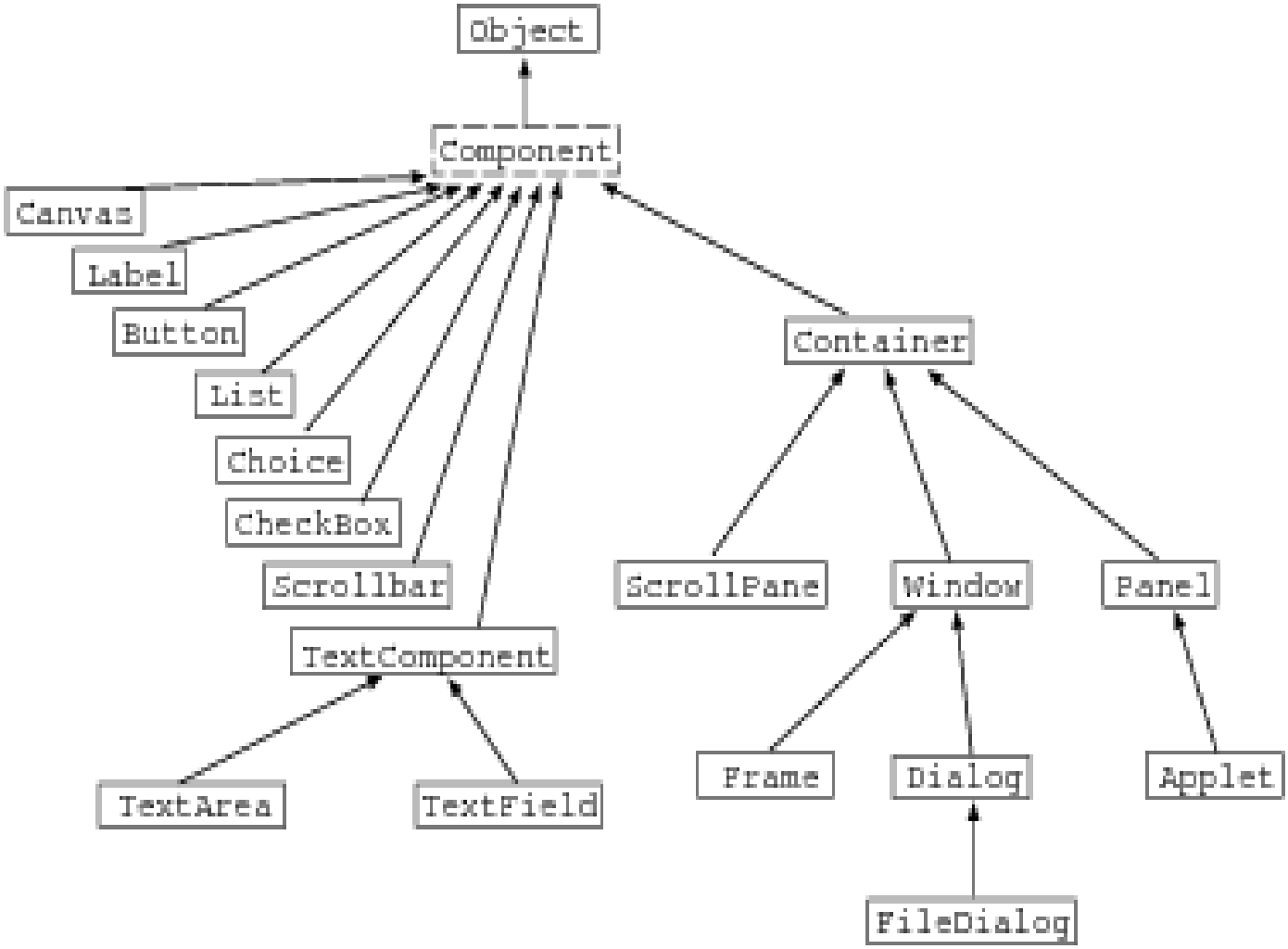
We give an overview of the AWT and discuss its relation to the MVC-pattern and in what respects it supports reuse.

Overview of the AWT

Like other GUI toolkits, the AWT is organized around a class hierarchy of GUI components.

GUI components are used to

- construct views,
- support event-handling,
- provide a mechanism to register so-called listeners.


Object
Component
Canvas
Label
Button
List
Choice
CheckBox
Scrollbar
TextComponent
TextArea
TextField
Container
ScrollPane
Window
Panel
Frame
Dialog
Applet
FileDialog

GUI components that are instances of class Container or its subclasses are called container, all others are called elementary components.

Characteristics of *containers*:

- GUI components can be added to and removed from a container.
- If a GUI component CP is contained in a container CT, then the representation of CP on the screen is contained in the representation of CT.
- The containment relation forms a hierarchy.

Characteristics of *windows*:

- Containers that are not included in other containers are called windows.
- Windows can be handled by the window manager of the platform on which the GUI is running, e.g. they can be moved, resized, and iconified.
- The main window of an application is called a *frame*; all other windows have to be registered with this frame to coordinate the event-handling.

The programmer has essentially three mechanisms to adapt the graphical representation of the GUI components.

1. Depending on the component type, she can customize some of its attributes, e.g. the title of a window or the label on a button.
2. She can override the method that draws the GUI component on the screen (only for special situations).
3. For each container, she can determine a layout manager.

The AWT provides a mechanism for event-handling similar to the MVC-pattern:

- Events occur at some GUI component CP.
- To get notified about the events occurring at CP, one has to register a listener object at CP.
- If an event of sort S occurs at CP, CP notifies all listener objects of type S that are registered at CP.
- The listener objects correspond to the controllers.
- The programmer of a GUI can provide the method that is executed when a listener is notified about an event.

Part of the flexibility of the AWT event-handling mechanism comes from the fact that

1. a listener object can be registered at more than one GUI component and

2. several listener objects can be registered at the same GUI component.

Shades of Reuse

If a programmer uses the AWT to implement a graphical user interface, he does not have to know

- how to refresh windows,
- how to render GUI components on the screen,
- how to make containers react to resize events,
- how to associate events with GUI components, and
- how to integrate all these features in a smooth way.

The knowledge about these aspects is captured in the architecture and implementation of the AWT.

Simple GUIs can be built from the classes of the AWT almost without specialization.

Simple construction technique:

1. Construct the views from the GUI components.
2. Connect the GUI components to the application kernel by implementing listener classes.
3. Register the listener objects with the GUI components.

In this case, the components, the graphics classes, and the event-handling mechanism are used in a *blackbox* fashion.

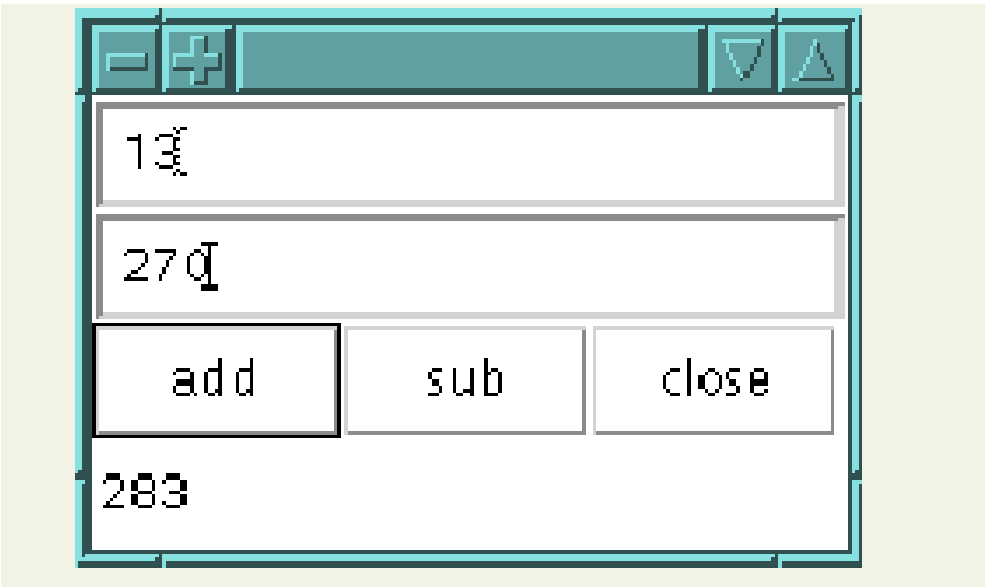
Specialization and inheritance, i.e., techniques based on whitebox reuse, are needed for extending the core functionality of the AWT.

Typical cases are

- the realization of specialized or compound GUI components
- and the refinement the event mechanism.

Example: (GUI not following MVC):

As an example application, we consider a simple calculator for adding and subtracting integers.



In this example, the application and its GUI are implemented within one class mixing aspects of view, controller, and model.

The object of class CalculatorFrame represents the view and is used as a listener for the three buttons.

The method actionPerformed that handles the button events also includes the application code.

```
package framework;

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

public class CalculatorFrame
  extends  Frame
  implements ActionListener
{
  TextField tfop1, tfop2;
  Label lbres;

  CalculatorFrame() {... } // see below

  public void actionPerformed(
     ActionEvent e ) {...} // see below

  public static void main(String argv[])
{
     new CalculatorFrame();
  }
}
```

```
CalculatorFrame(){
  setLayout( new GridLayout(4,1) );
  // input fields
  add( tfop1 = new TextField("",15) );
  tfop1.setEditable( true );
  add( tfop2 = new TextField("",15) );
  // buttons to trigger operations
  Panel p = new Panel(
                new GridLayout(1,3) );
  Button bt;
  p.add( bt = new Button("add") );
  bt.addActionListener(this);
  p.add( bt = new Button("sub") );
  bt.addActionListener(this);
  p.add( bt = new Button("close") );
  bt.addActionListener(this);
  add( p );
  // label to show the result
  add( lbres = new Label("") );
  setSize( 200, 120 );
  setVisible( true );
}
```

```
public void actionPerformed(
     ActionEvent e )
{
 try {
  String buttonlabel =
    ((Button) e.getSource()).getLabel();

  if( buttonlabel.equals("close") )
    System.exit( 0 );

  int op1 =
     Integer.parseInt( tfop1.getText() );

  int op2 =
     Integer.parseInt( tfop2.getText() );

  if( buttonlabel.equals("add") ){
    lbres.setText(
           String.valueOf(op1 + op2) );
  } else if( buttonlabel.equals("sub") ){
    lbres.setText(
           String.valueOf(op1 - op2) );
  }
 } catch( NumberFormatException nfe ){}
}
```

The example should show that the AWT does not prescribe the special architecture of the GUIs constructed by it.

The AWT allows to merge application, model, controller, and view into one class.

This flexibility can be misused:

- For small applications, this flexibility can be helpful.
- For larger applications or for applications where an evolution is probable, such unstructured implementations can become unmanageable.

Summarizing Discussion

In summary, a program framework like the AWT can fix some architectural decisions and leave other aspects of architectural design open.

Program frameworks can support more powerful architectural patterns, in particular patterns for whole application systems.