

***Software
Technology
Group***

TU Darmstadt | FB Informatik

Software Engineering Design

9. Frameworks

Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

Dipl.-Ing. Michael Haupt

Object-Oriented Frameworks

OO framework: a collection of cooperating classes that together define a generic or template solution to a family of domain specific requirements.

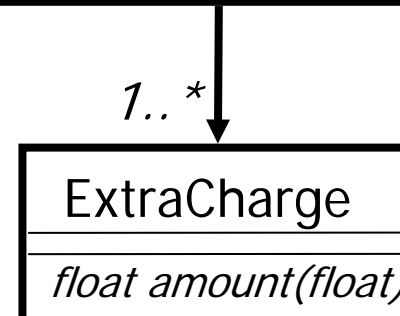
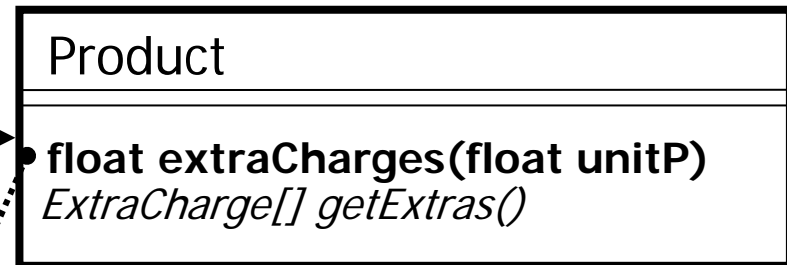
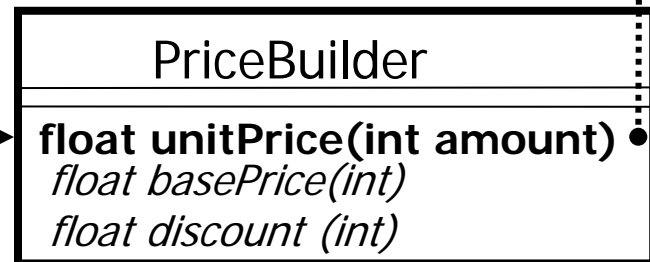
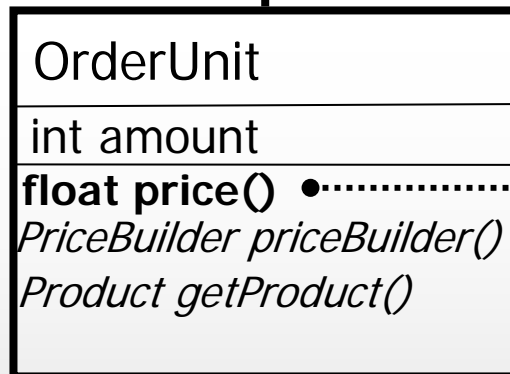
A framework is *the design for an application* or subsystem: A set of abstract classes and the way objects in those classes collaborate.

Ralph E. Johnson

Example: A Mini-Framework for Price Calculation in Order Management Domain

```
PriceBuilder pb = priceBuilder();
Product p      = getProduct();
float up       = pb.unitPrice(amount);
float extras   = p.extraCharges(up);
return unitPrice + extras;
```

```
return basePrice(amount) -
       discount(amount);
```



```
int extras;
for(ExtraCharge e : getExtras())
    extras += e.amount(unitP);
return extras;
```

Frameworks Embody Design Insight

- Dictate the overall structure of a family of applications by describing:
 - how the responsibilities are partitioned between various components, and
 - how these components interact with each other
- Without being too detailed about the concrete implementation of the components
 - make intensive use of the **template method** pattern for modeling “hot-spots” in a design

Frameworks Embody Design Insight

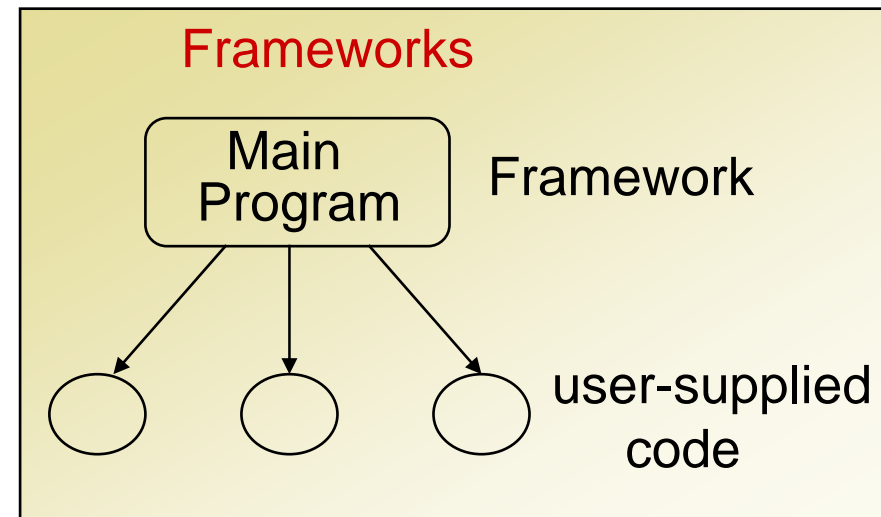
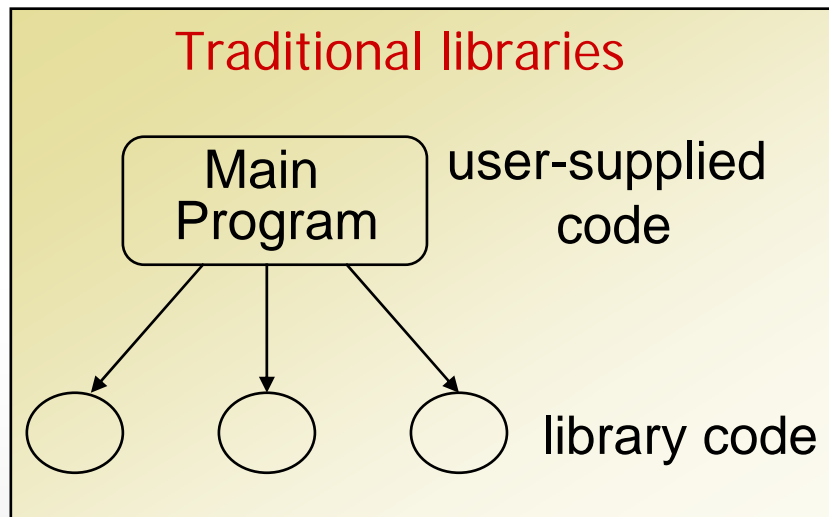
- Framework defines
 - the model of some domain (or aspect thereof)
 - the abstract design of this model as a set of interfaces
 - the space of possible runtime object configurations
 - the object coupling on an abstract level (interfaces only)
 - the allowed control flow between objects
 - the distribution of responsibilities between classes
 - possible implementations and the constraint thereon by means of a partial (abstract) implementation
 - generic reusable functionality
 - a reuse skeleton and a reuse contract

Instantiating Object-Oriented Frameworks

- Application-specific details filled in during **instantiation (customization)** of the framework by refining **“hot-spot” methods**
 - application designer needs only concentrate on the specifics of the application at hand
 - previous design decisions embodied in the structure of the framework need not be reexamined, nor does the code provided by the framework need to be rewritten

Hollywood Principle: Don't Call Us, We Will Call You

- *Inversion of control* → the framework plays the role of a main program in coordinating and sequencing application activity



- Flow of **control** is **dictated by** the **framework** and is the same for all applications

Using Object-Oriented Frameworks

- **Client**
 - clients can readily use the framework
 - clients can customize the framework to their needs
 - in any case: complex implementation issues are hidden from them
- **Implementation reuse**
 - reuse of readily instantiable classes as is
 - reuse of abstract implementations by inheritance
- **Design reuse**
 - a framework represents the domain concepts adequately
 - is flexible and adaptable to accommodate specific client requirements

Examples of Real OO Frameworks

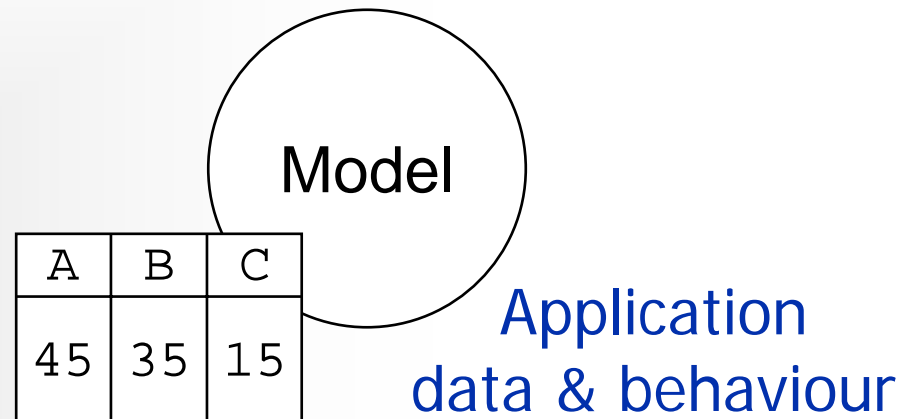
Frameworks made the GUI Revolution possible

- MVC
- Macintosh Toolbox
- MacApp
- Interviews/ET++
- OWL/MFC
- AWT, Swing, SWT, etc.

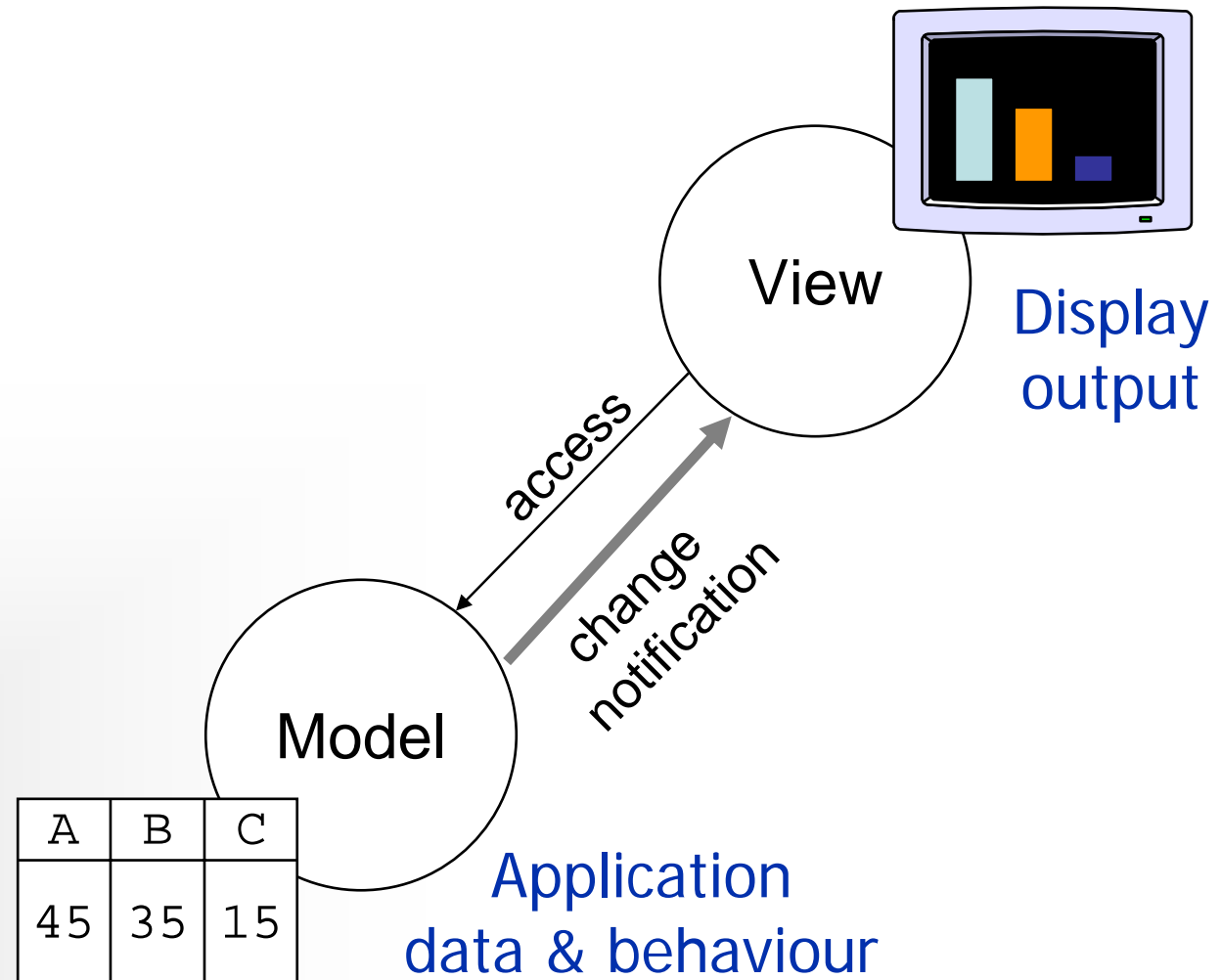
Common concept: Model-View-Controller

- Three clearly separated parts
- Parts that represent the **model** of the underlying **application domain**
 - accounts, portfolios, stock performance data
- The way the model is **presented** to the user
 - balance list, performance graphs
- The way the user **interacts** with it
 - buttons, menus, direct manipulation

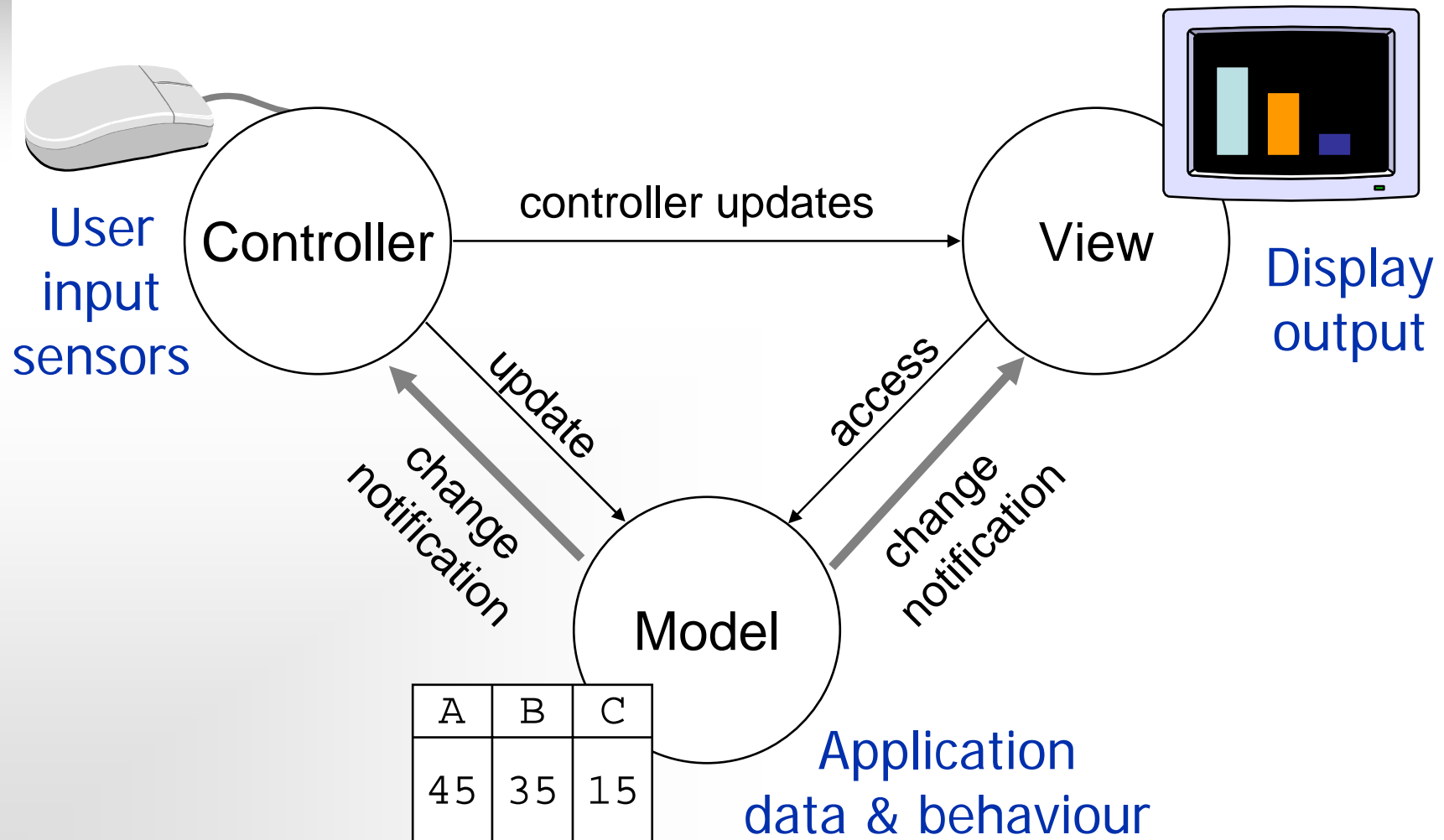
Model-View-Controller



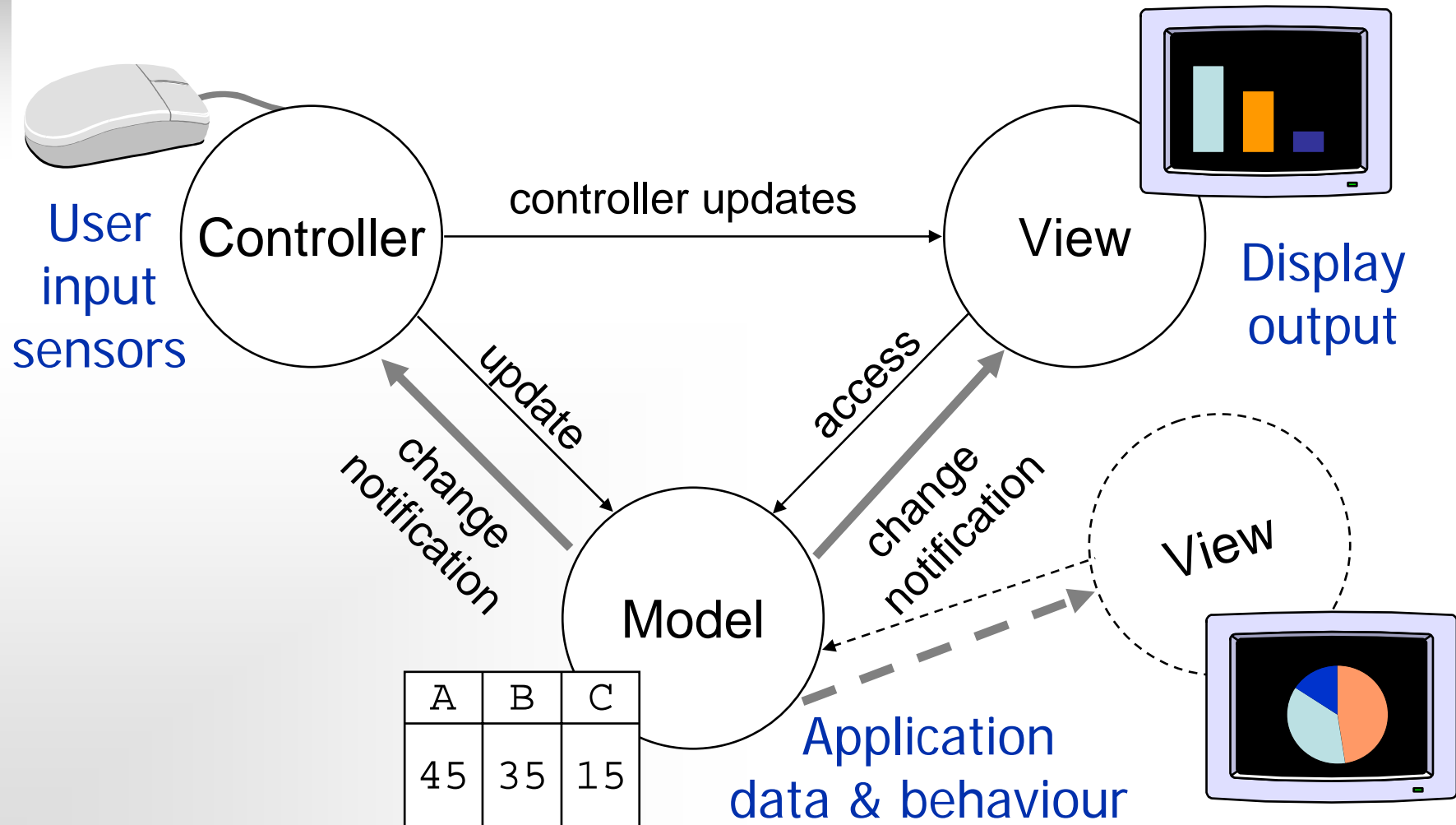
Model-View-Controller



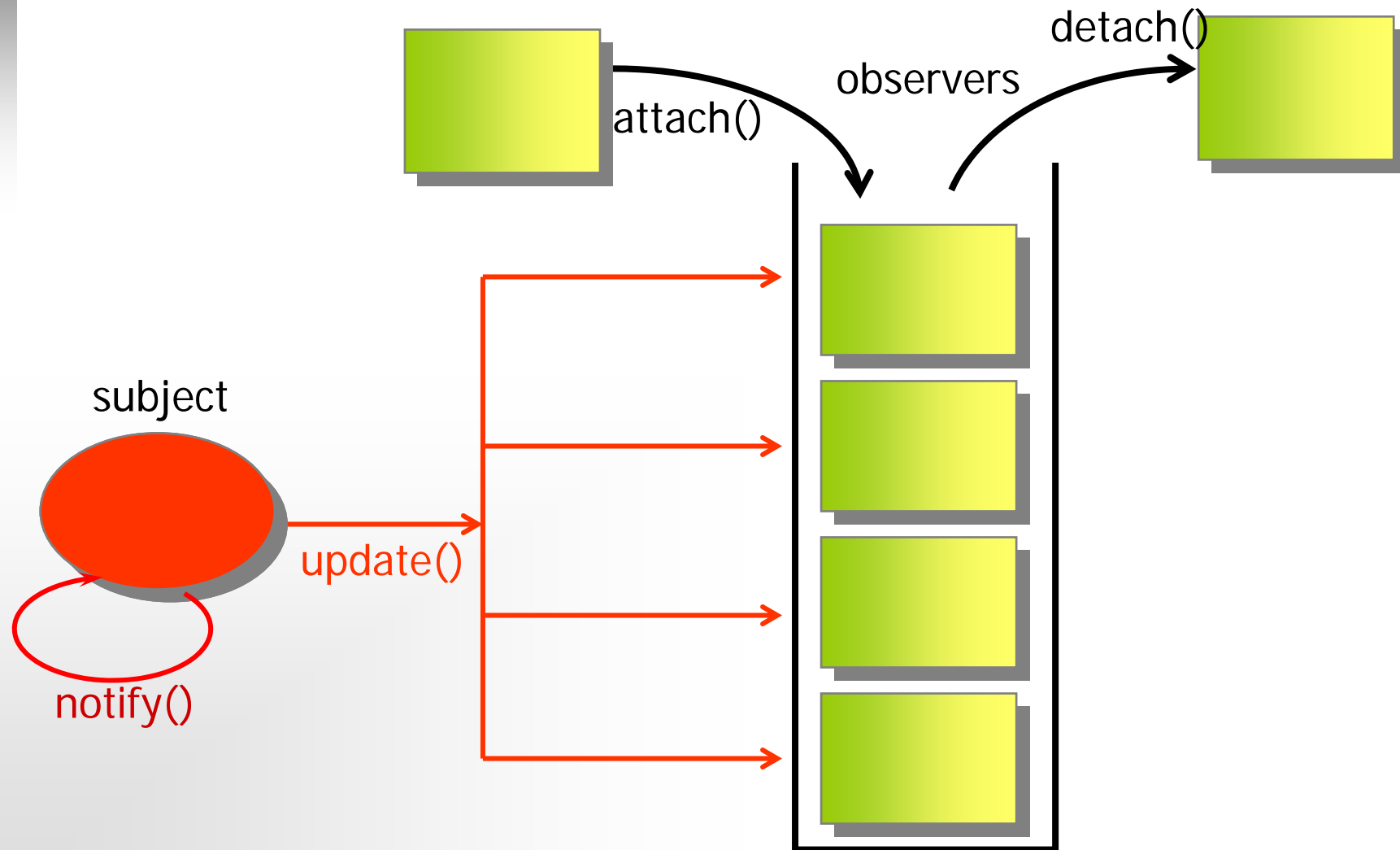
Model-View-Controller



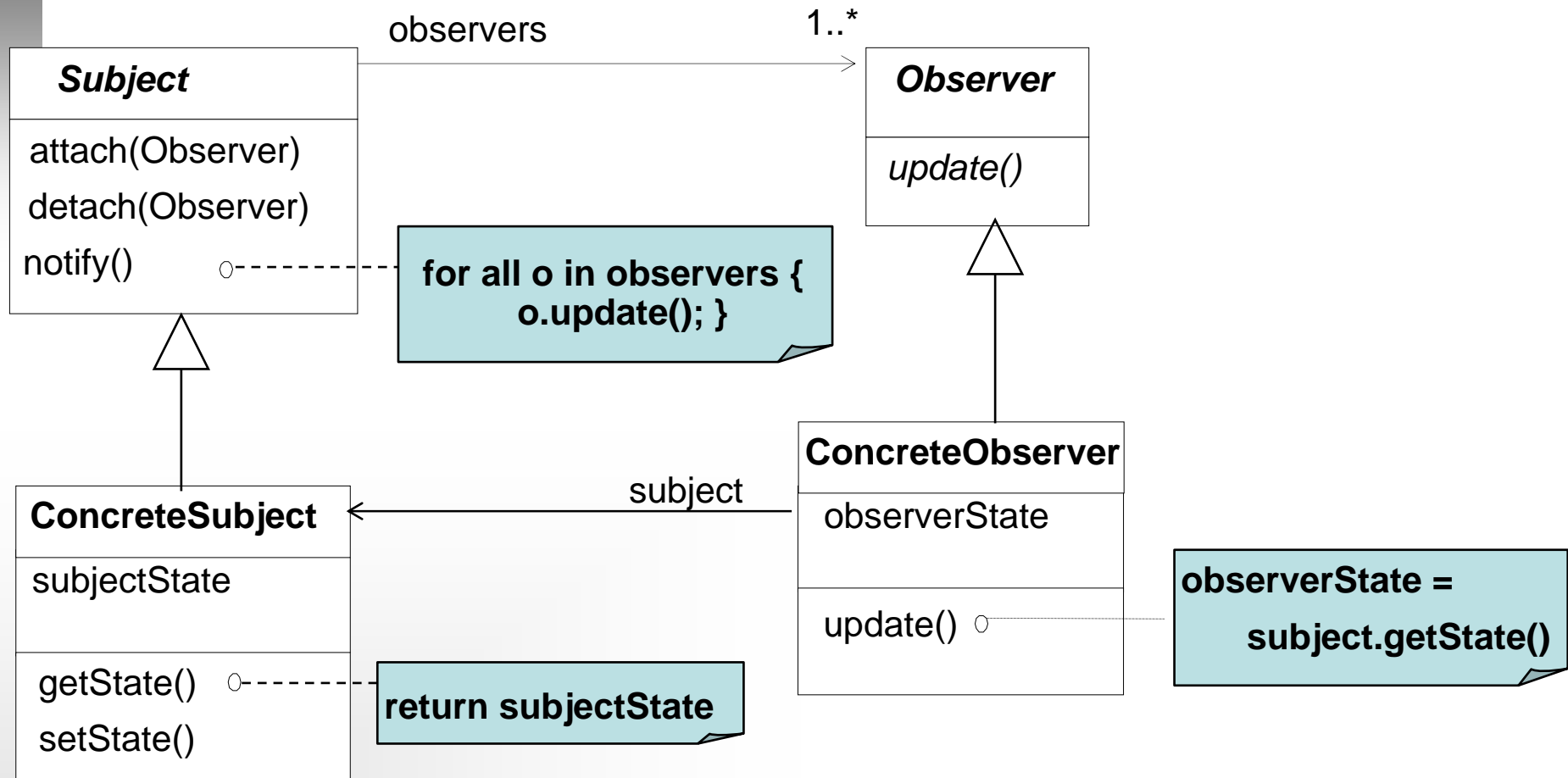
Model-View-Controller



Subject-Observer Protocol

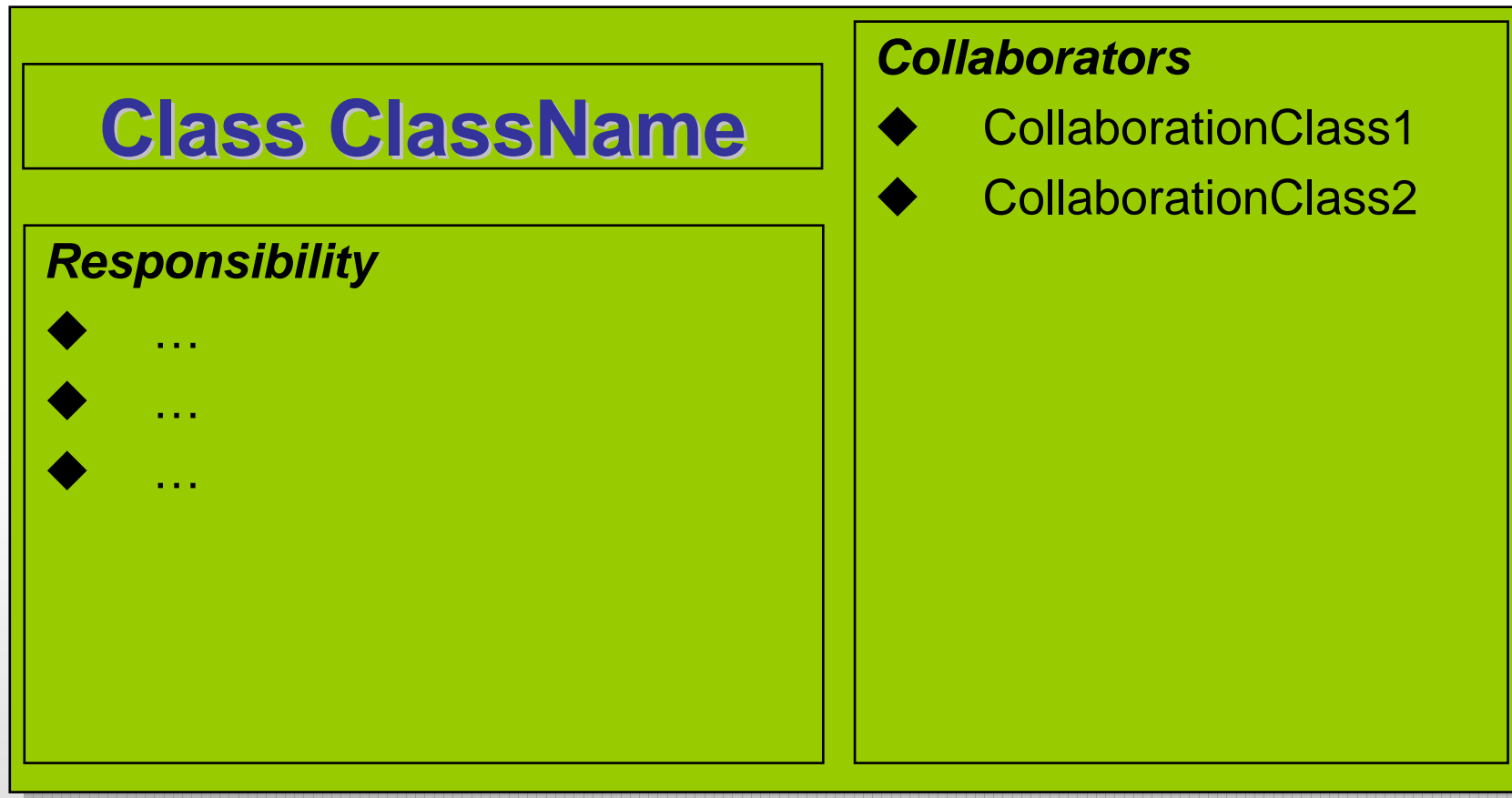


Subject-Observer Structure



Model-View-Controller

Class Responsibility Collaboration (CRC) Cards



Model-View-Controller

CRC Card for Model

Class Model

Responsibility

- ◆ Provides functional core of the application
- ◆ Registers dependent views and controllers and provides data access methods for them
- ◆ Notifies dependent components about data changes. Observer pattern may be used for the change propagation

Collaborators

- ◆ View
- ◆ Controller

Model-View-Controller

CRC Card for View

Class View

Responsibility

- ◆ Creates and initializes a suitable associated controller (one-to-one)
- ◆ Retrieves data from the model
- ◆ Displays model information to the user
- ◆ Implements the update procedure
- ◆ Offers interfaces allowing the controller for some display manipulations that do not affect the model (e.g. scrolling)

Collaborators

- ◆ Controller
- ◆ Model

Model-View-Controller

CRC Card for Controller

Class Controller

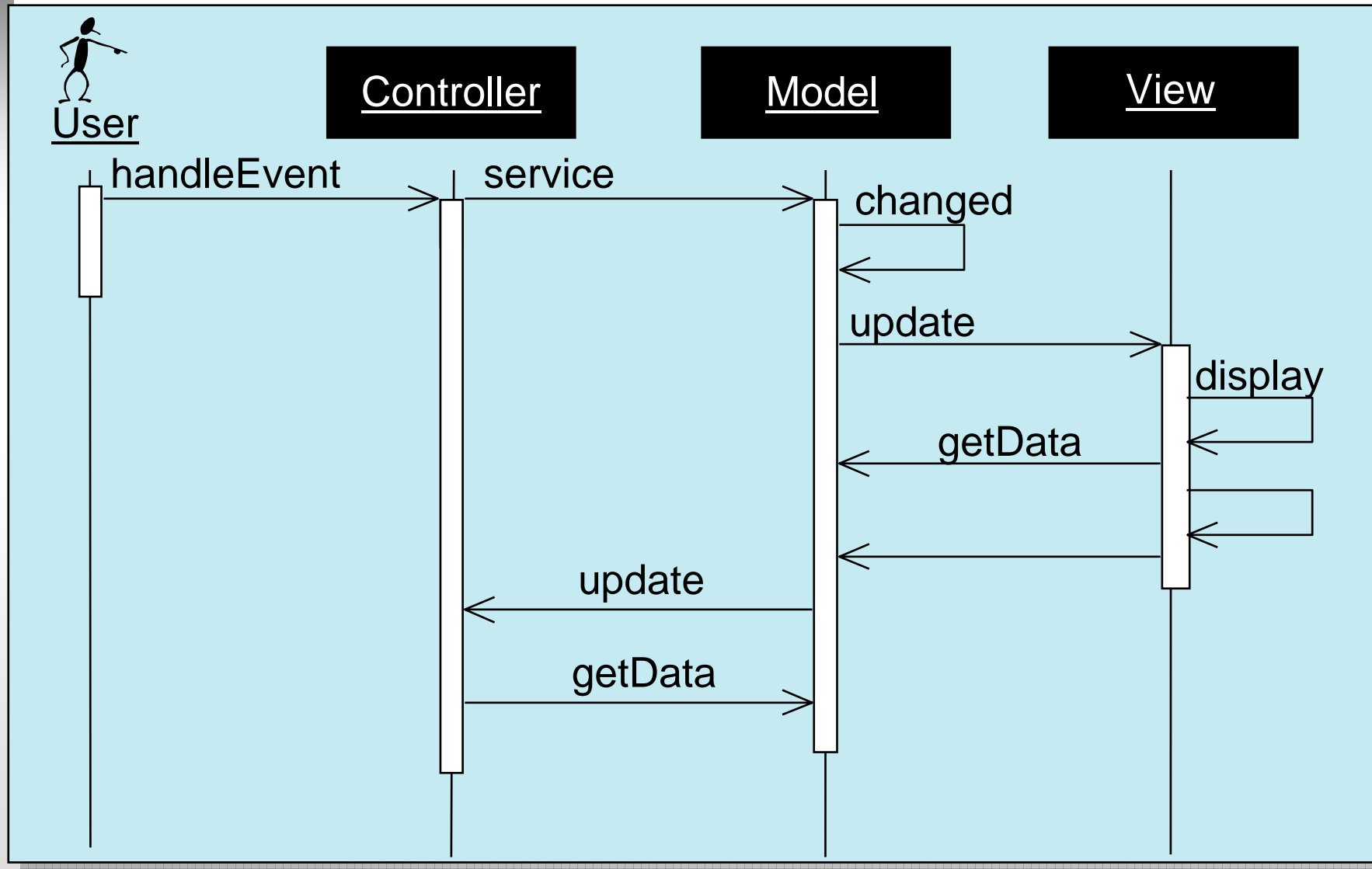
Responsibility

- ◆ Accepts user input as events
- ◆ Translates events to service requests for the model or display requests for the view
- ◆ Implements the update procedure, if required, e.g., in the case the behavior of the controller depends on the state of the model (e.g., a change to the model enables or disables a menu)

Collaborators

- ◆ View
- ◆ Model

MVC Sequence Diagram



Frameworks vs. Patterns

- **Patterns are smaller** than frameworks.
 - a **framework** generally **includes** concrete realizations of **several patterns**. The opposite is not true.
- **Patterns are more abstract** than frameworks.
 - **patterns** define **solutions for design issues** and may only be illustrated by means of sample code.
 - **patterns do not provide implementations** that can be reused as it is. Frameworks do.
- **Patterns are less specialized** than frameworks.
 - **framework**: template solution that **applies to a particular application domain**.
 - **pattern**: design technique that **applies to any OO software**

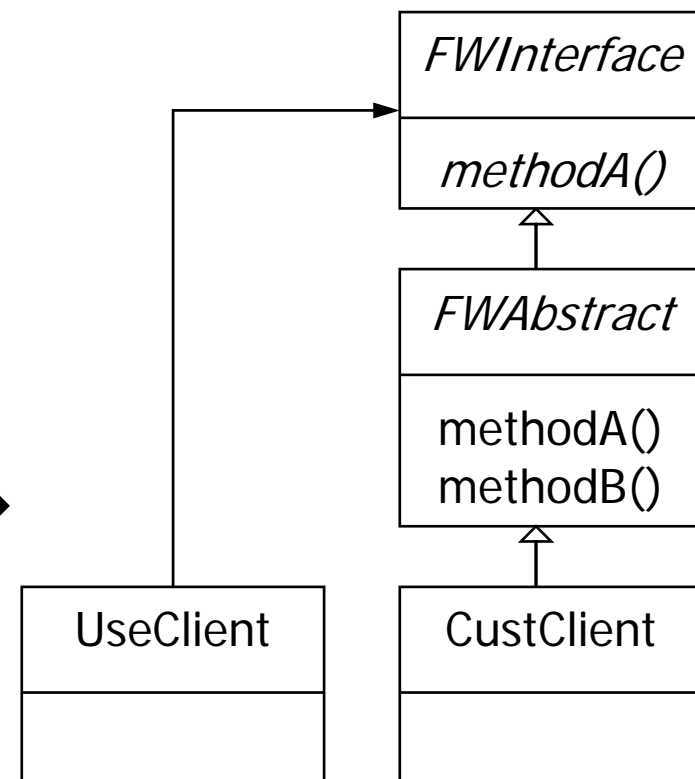
Black-Box vs. White-Box Frameworks

- **White-Box Frameworks:** Customized by subclassing existing framework classes and providing concrete implementations
- User provides concrete subclasses and methods for abstract classes and methods marked as “**hooks**” within the framework’s “**hot spots**” regions

Black-Box vs. White-Box Frameworks

- **White-Box Frameworks:** Two kinds of clients
 - “**customizing clients**” customize the framework for specific needs (instantiate the framework)
 - “**use clients**” use the framework from the outside

- **Different interfaces**
 - use clients depend on the abstract design
 - customizing clients use the inheritance interface → they need to know the framework in more detail



Black-Box vs. White-Box Frameworks

- **Black-Box Frameworks**
 - filling in parameters of or plugging together compatible components from a library of prefabricated components
 - **prefabricated components**: existing framework classes which must be complete implementations of some interface in the abstract design

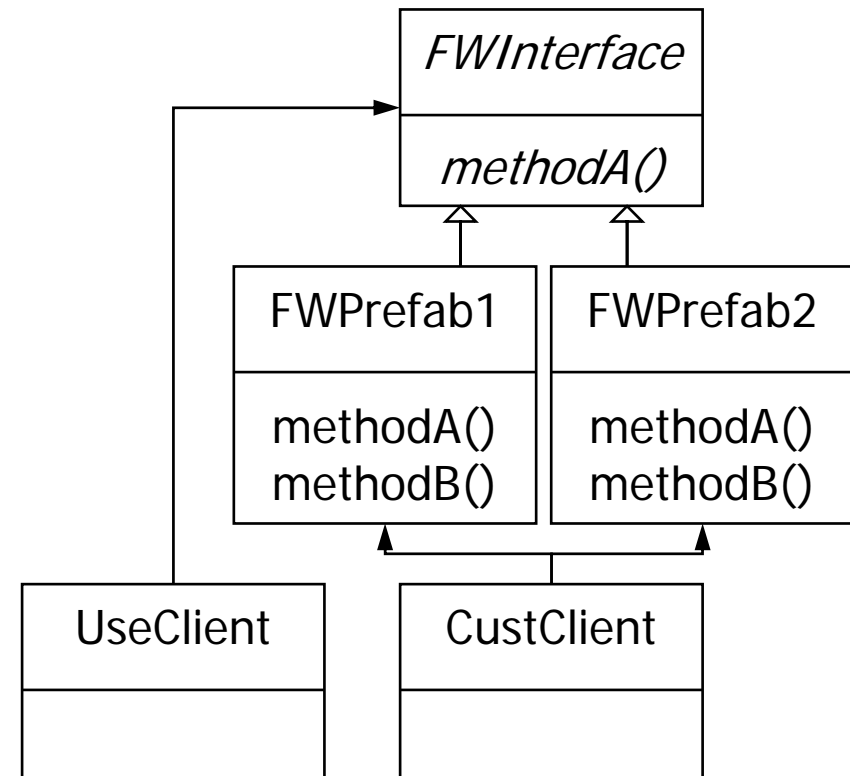
Black-Box vs. White-Box Frameworks

- **Application specific functionality** represented by **object parameterization** and **composition**
 - selection and parameterization supported by tools, e.g., GUI builders.
- **Simplest adaptation mechanism**
- **Flexibility limited** by the fixed number of choices foreseen by the framework developer

Black-Box vs. White-Box Frameworks

Black-Box Frameworks

- Two kinds of clients:
 - those that use the framework from the outside
 - those that configure it
- Different interfaces
 - the first one uses the abstract design
 - the second one uses the composition interface
- The two client roles frequently are played by the same client



Black-Box vs. White-Box Frameworks

- **Grey box frameworks:** frameworks using both parameterization and refinement
- Frameworks typically evolve from white-box to black-box frameworks over a number of iterations

Characteristics of Frameworks

- Frameworks designed for extension
- Two types of functionalities
 - domain-specific behavior
 - interaction behavior
- Communication framework / application
 - one-way (the normal case)
 - two way
- Framework acquires single thread of control ("Hollywood principle")

Problems with Framework Integration

- Characteristics result in problems when integrating more than one framework
- **Inversion of control**
 - frameworks assume to own thread of control
- Work-around
 - one thread per framework

Framework Integration cont'd

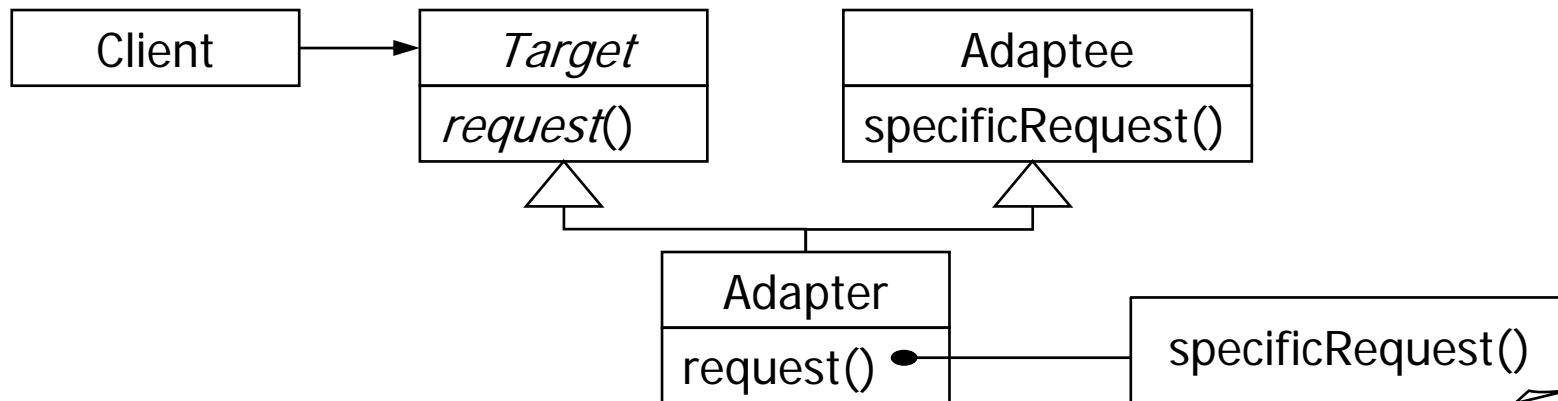
- **Integration with legacy code**
 - framework relies on subclassing
- **Work-around**
 - adapter design pattern

Design Pattern: Adapter

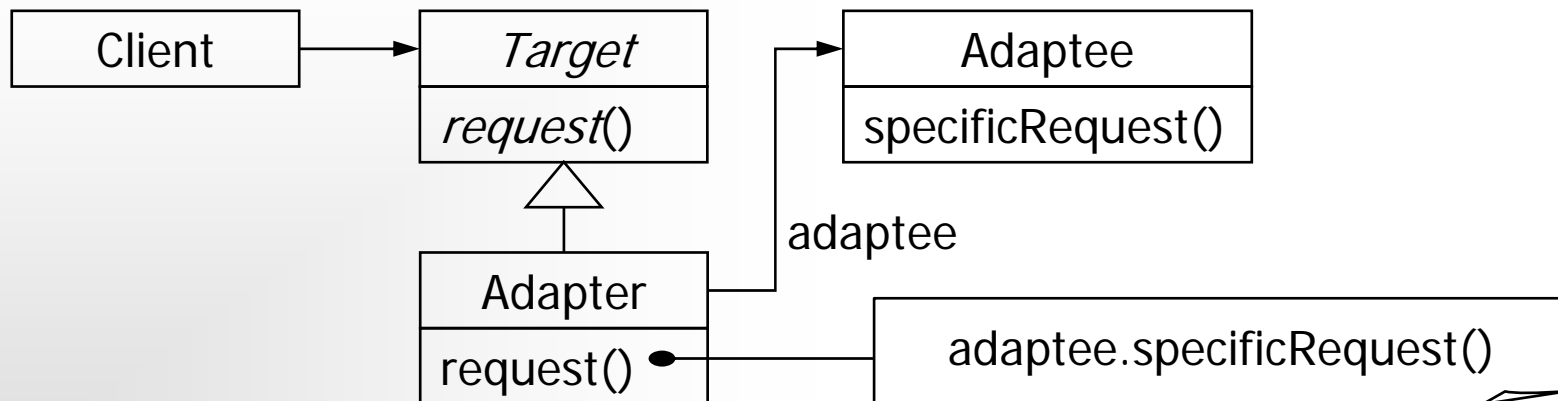
- Intent
 - Convert the interface of a class into another interface clients expect.
- Two variations
 - Class adapter uses inheritance
 - Object adapter uses object composition
- Participants
 - Client
 - Target: the interface the client wants to deal with
 - Adaptee: the existing interface that needs adapting
 - Adapter: adapts the interface of Adaptee to Target

Design Pattern: Adapter

- Class Adapter



- Object Adapter



Framework Integration cont'd

- **Overlapping of framework components**
 - frameworks have same real-world components
 - different representations
 - share properties

Framework Integration cont'd

- **Architectural mismatches**
 - different models of integrated framework components
 - different interactions
 - different pragmatics
 - object oriented principle
 - pipes and filters architectural style

Case-studies

- Eclipse



- JHotDraw



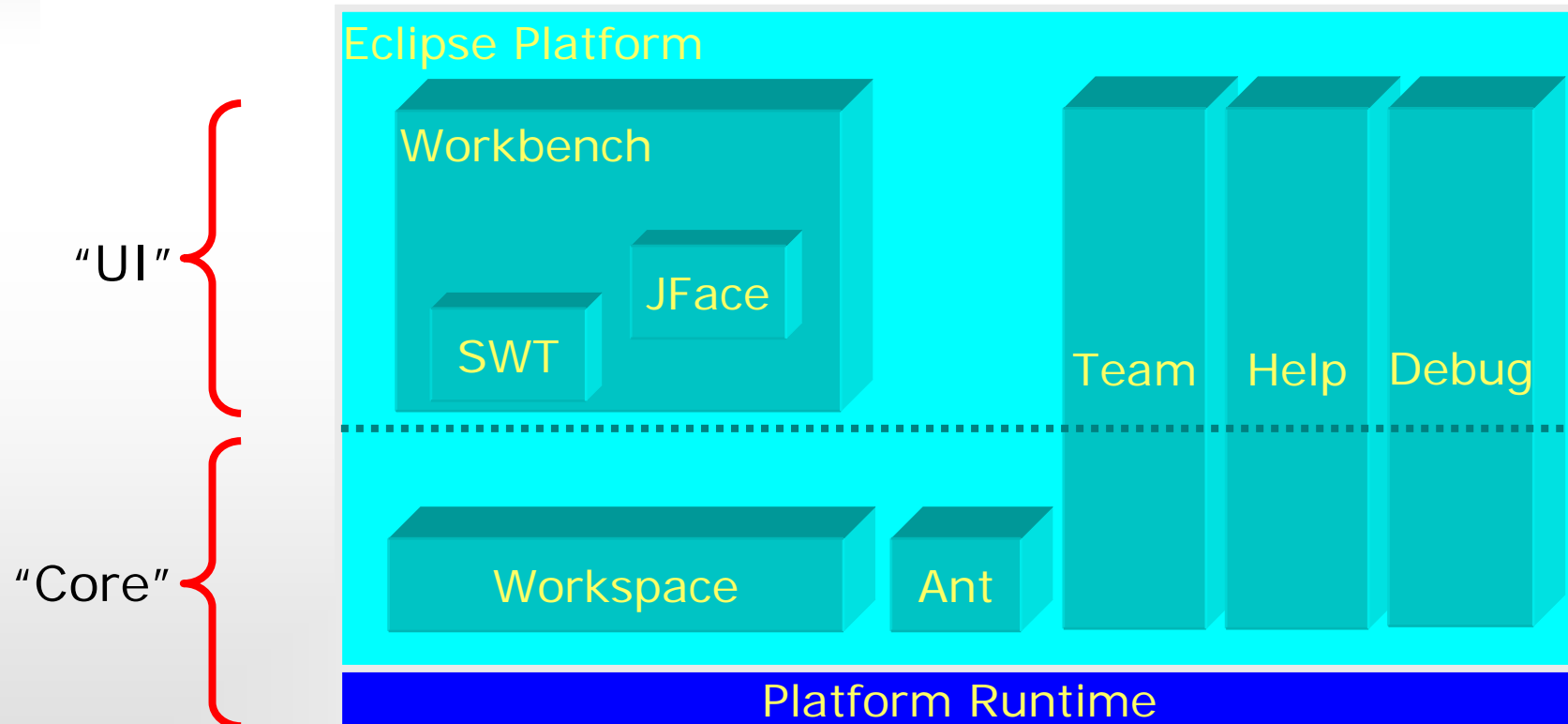
- GEF

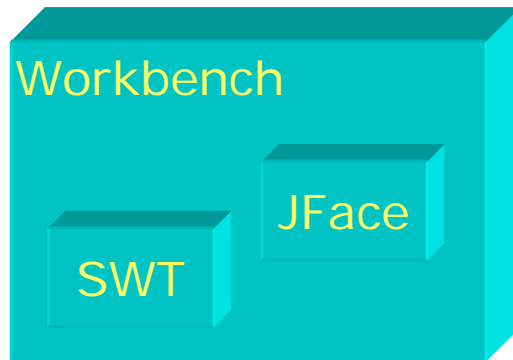


Case-Study: Eclipse Platform



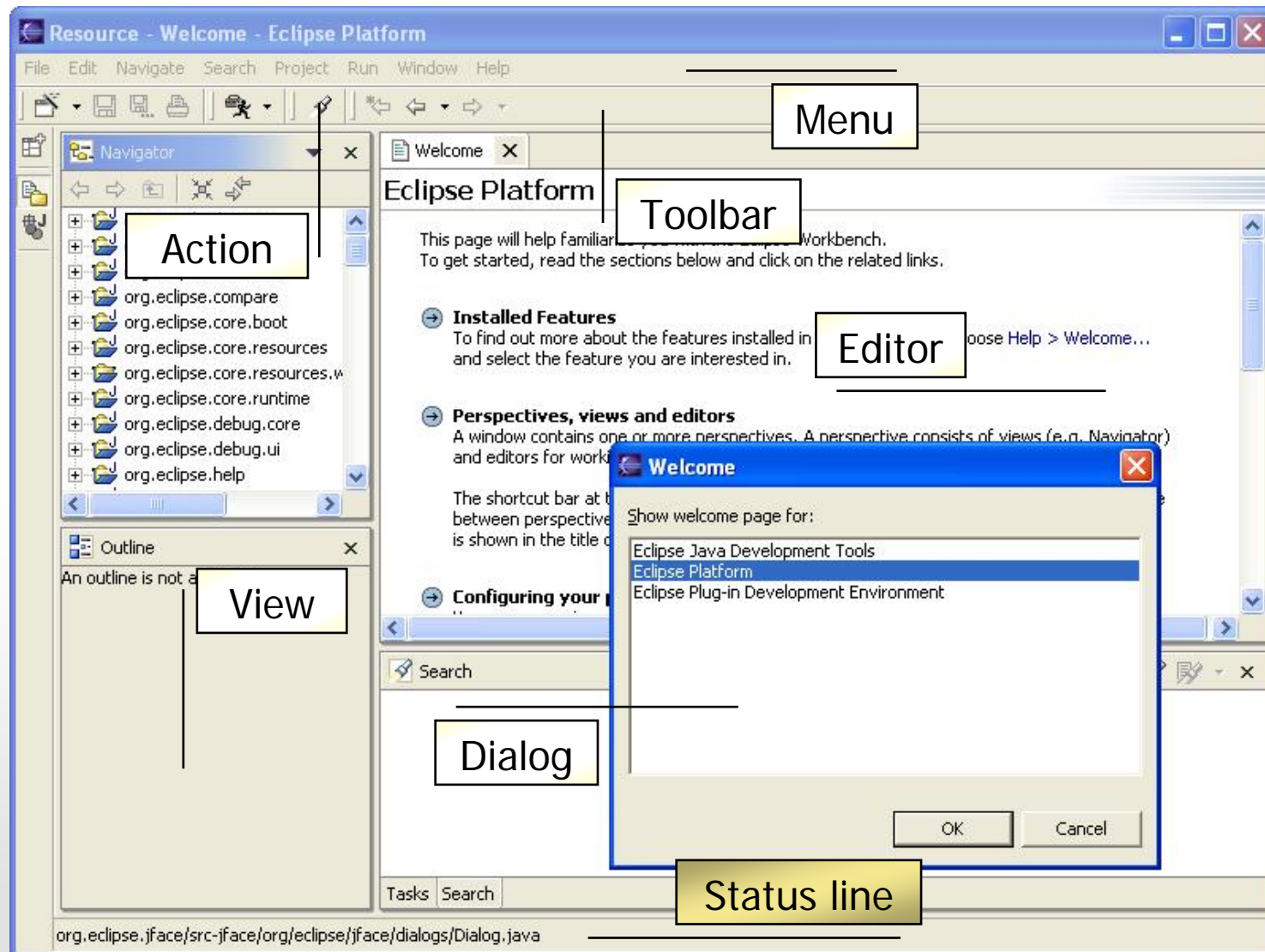
- Eclipse Platform is the common base
- Consists of several key components





- SWT – generic low-level graphics and widget set
- JFace – UI frameworks for common UI tasks
- Workbench – UI personality of Eclipse Platform

Eclipse Workbench cont'd





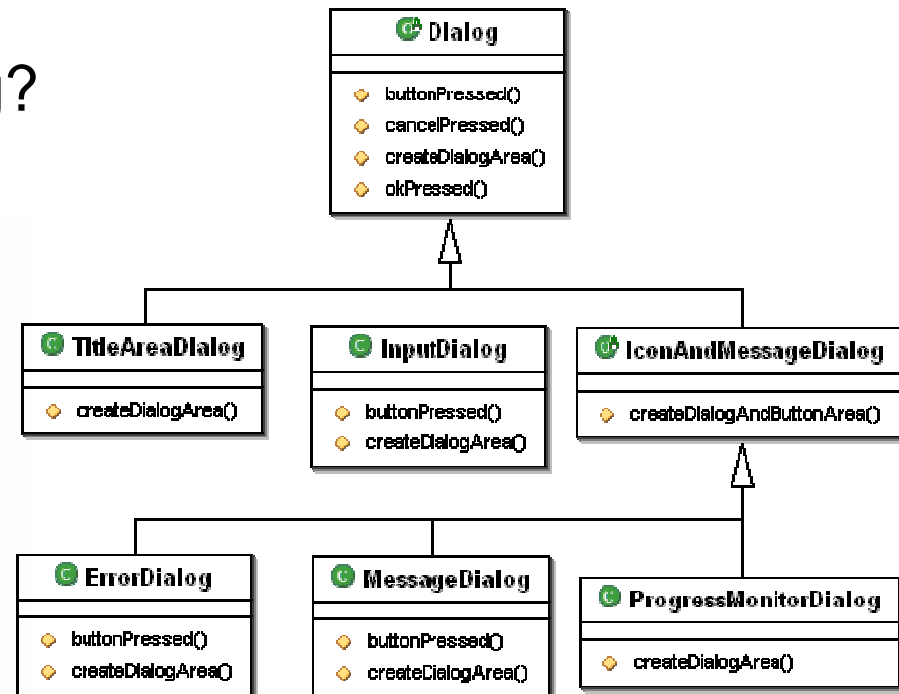
- JFace is set of UI frameworks for common UI tasks
- Frameworks / APIs
 - Image and font registries
 - Dialog, preference, and wizard frameworks
 - Structured viewers
 - Model-aware adapters for SWT tree, table, list widgets
 - Text infrastructure
 - Document model for SWT styled text widget
 - Coloring, formatting, partitioning, completion
 - Actions
 - Location-independent user commands
 - Contribute action to menu, tool bar, or button

- Use clients
 - “plug” classes together
 - may be customized or ready-to-use components
 - use plug-in manifest file
 - written in different language (XML)
- Customizing client
 - subclass or compose framework classes
 - customizing with Java code

Example: White-Box customization



- Overwrite protected methods from `Dialog`
- Dialog framework calls back
- How to document methods / classes in frameworks?
 - intended for subclassing?
 - how to use super implementation?
 - when to make super call?



```
/**
 * Creates and returns the contents of the upper part
 * of this dialog (above the button bar).
 * The Dialog implementation of this framework method
 * creates and returns a new Composite with
 * standard margins and spacing.
 * The returned control's layout data must be an instance of
 * GridData.
 * Subclasses must override this method but may call super
 * as in the following example:
 * Composite composite =
 *     (Composite) super.createDialogArea(parent);
 * //add controls to composite as necessary
 * return composite;
 */
protected Control createDialogArea(Composite parent) { ... }
```

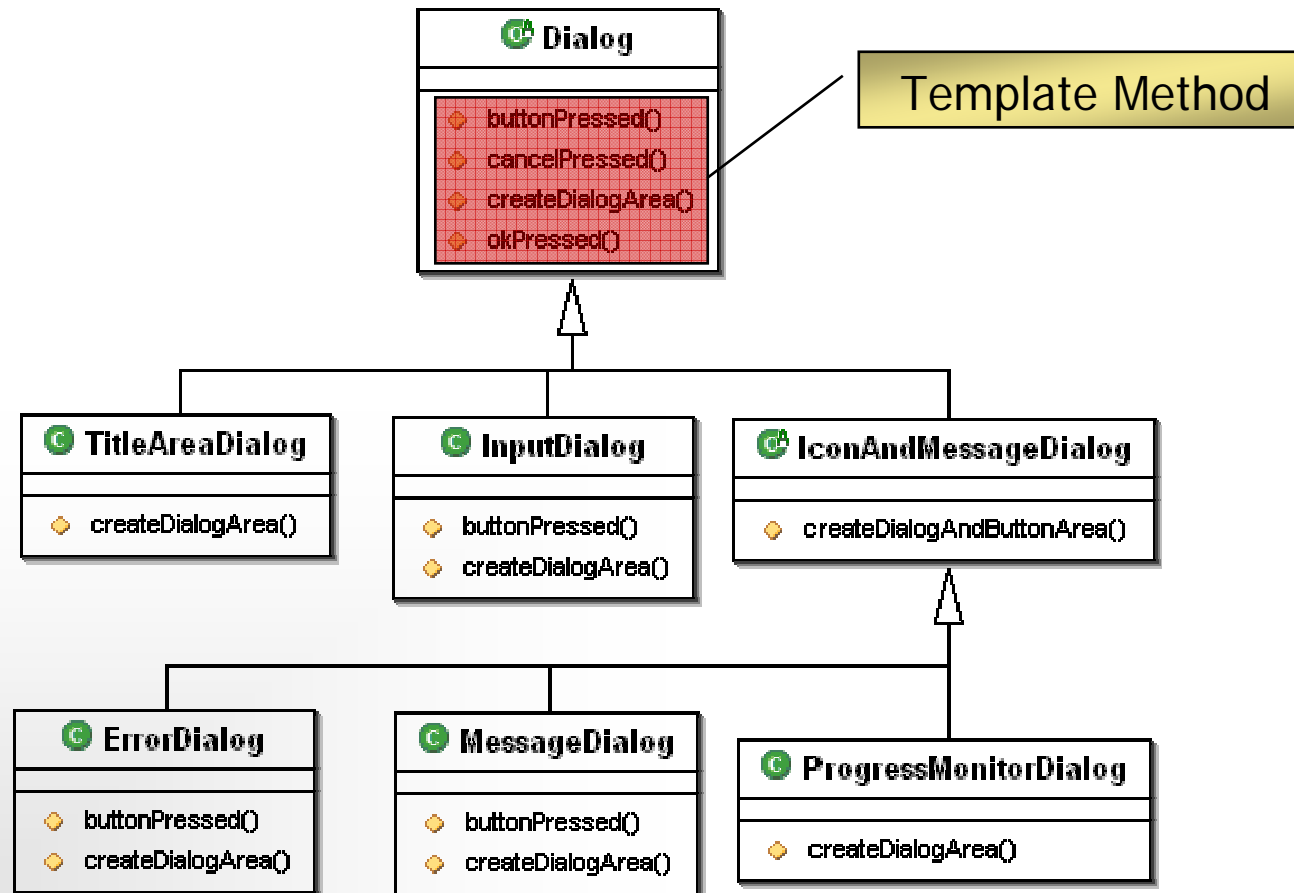
Default
behaviour

Intention

Constraints

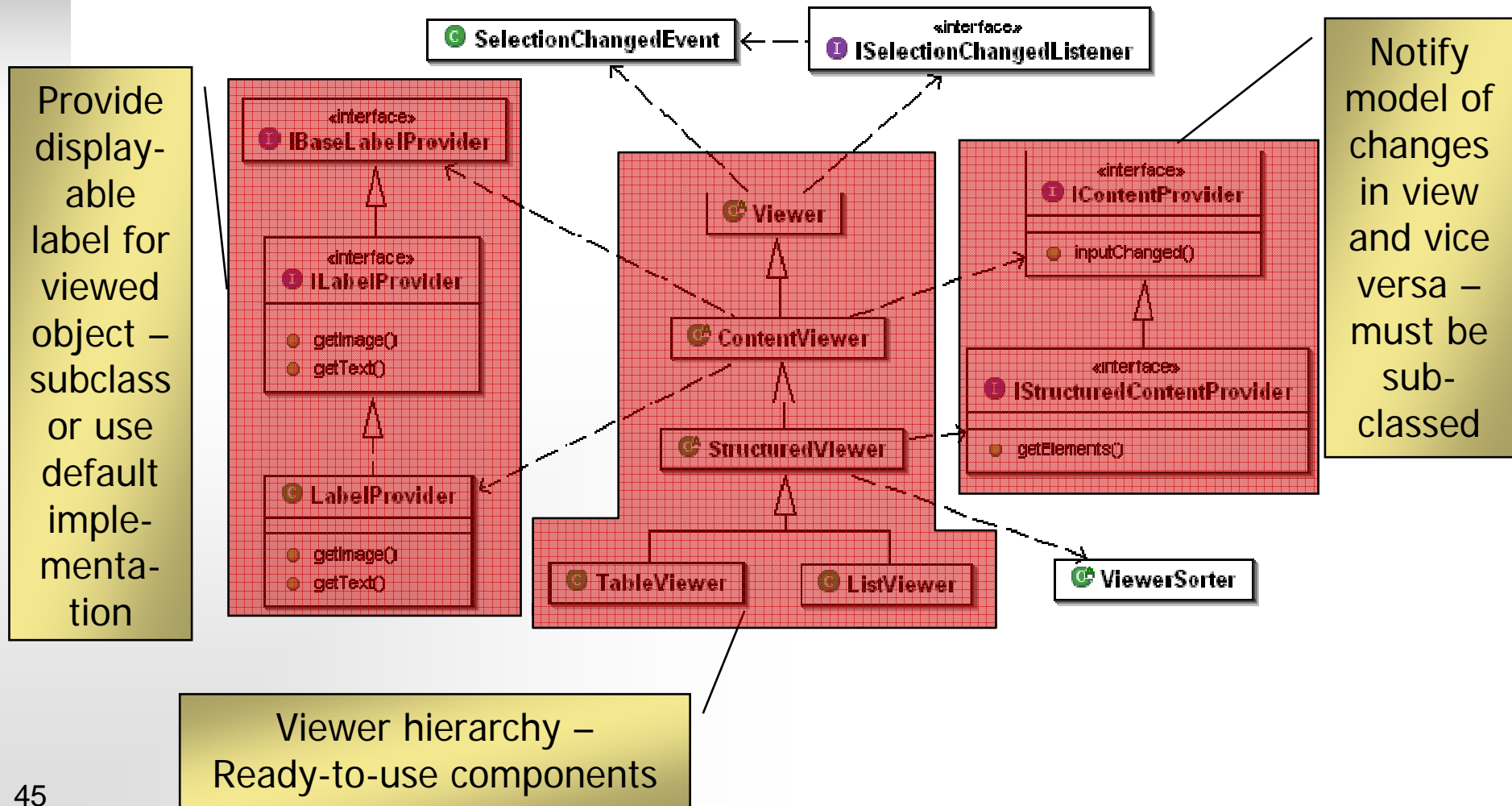
Override
instructions

Eclipse Dialogs: Design Patterns

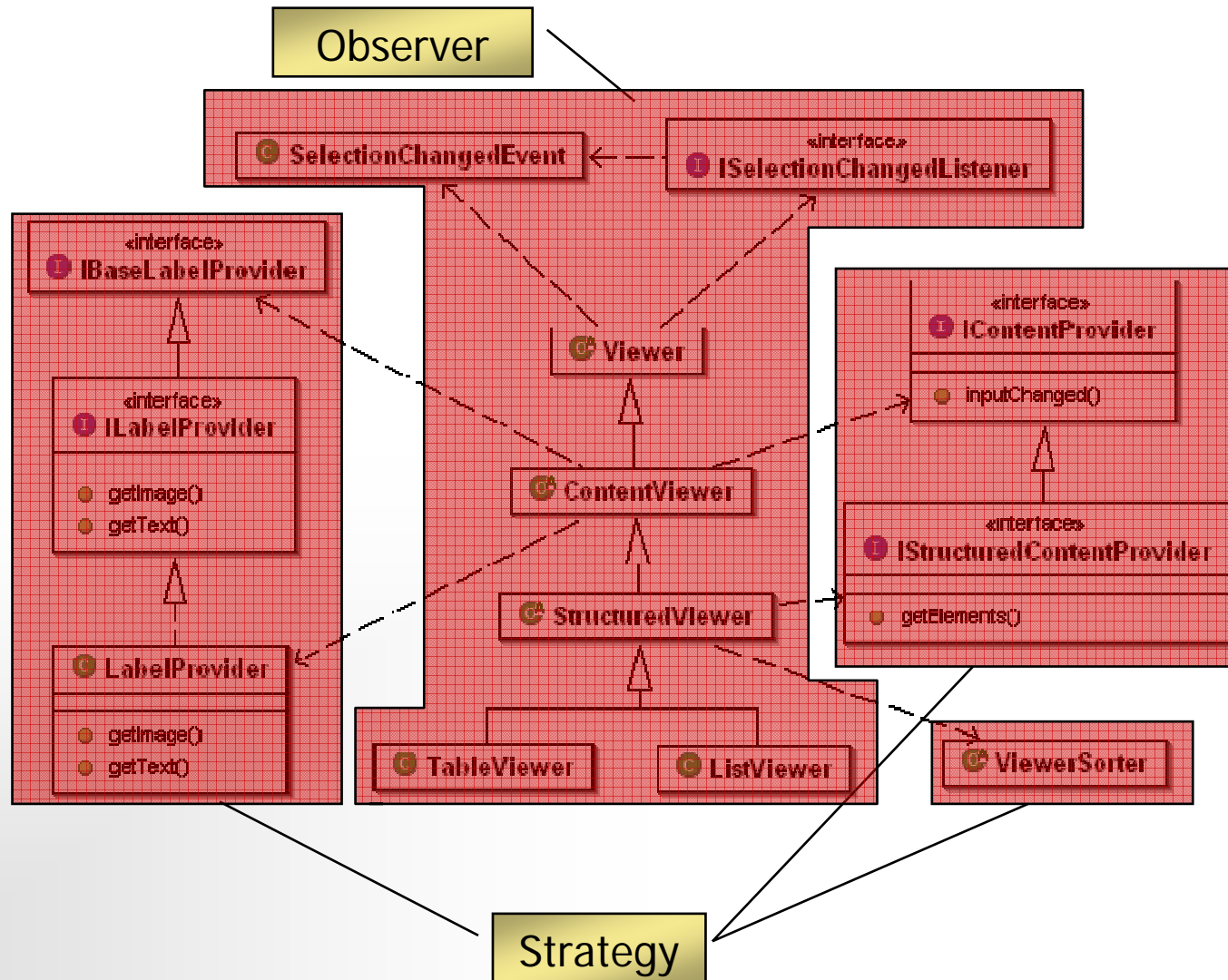




JFace viewer package



JFace Viewer: Design Patterns



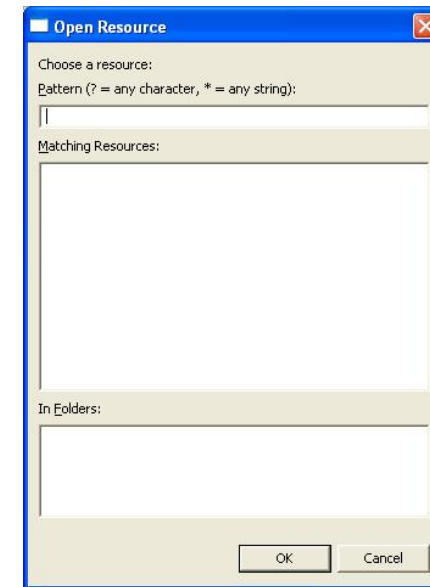
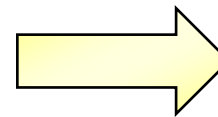
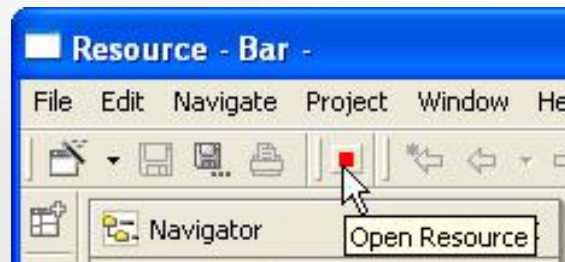
Example: Use of Ready-to-use components



- Contribute existing action to toolbar

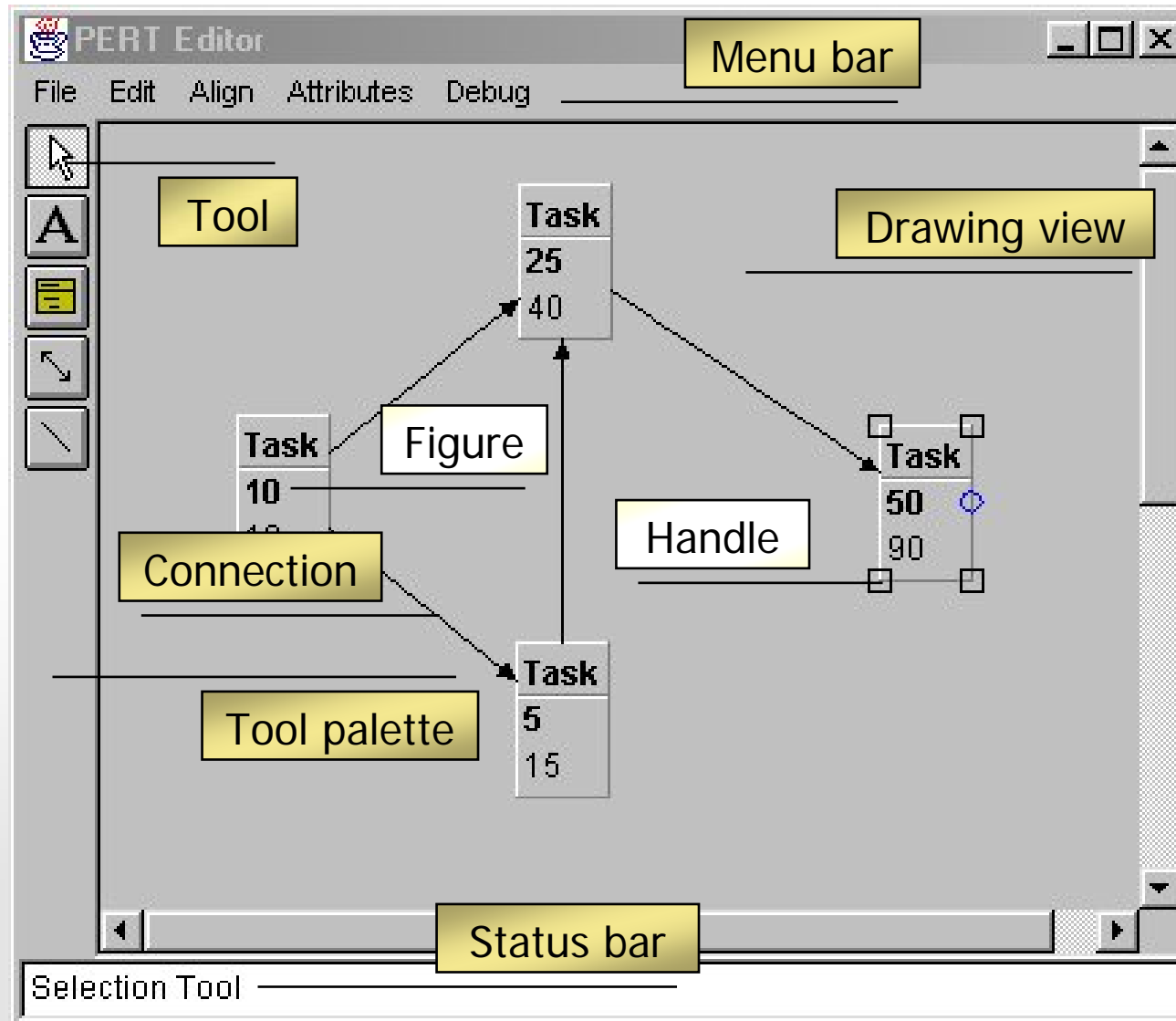
```
<extension point="org.eclipse.ui.actionSets">  
  <actionSet id="aoodActionSet" label="AOOD action set">  
    <action label="openResource" toolbarPath="aood"  
      class="org.eclipse.ui.internal.actions.OpenWorkspaceFileAction"  
      id="aoodAction" tooltip="Open Resource">  
    </action>  
  </actionSet>  
</extension>
```

Where to plug
What to plug



- **JHotDraw** is an application framework to develop editors for specialized two-dimensional structured drawing
 - schematic diagrams, blueprints, or program designs
 - elements of these drawings can have connections between them, can react to commands by the user, can be animated, ...
- Developed by Erich Gamma and Thomas Eggenschwiler in Java; Its predecessor – HotDraw – was developed in Smalltalk by Kent Beck, Ward Cunningham and Ralph Johnson

Case-Study: JHotDraw - overview



- **Figure** is the most important abstraction of the framework
- Classification
 - geometric (**EllipseFigure**, **PolygonFigure**, **RectangleFigure**, **LineFigure**, ...)
 - textual (**TextFigure**, **NumberTextFigure**)
 - structural (**CompositeFigure**, **GroupFigure**, ...)
 - decorating (**DecoratorFigure**, **BorderDecorator**, ...)

- *Figure*

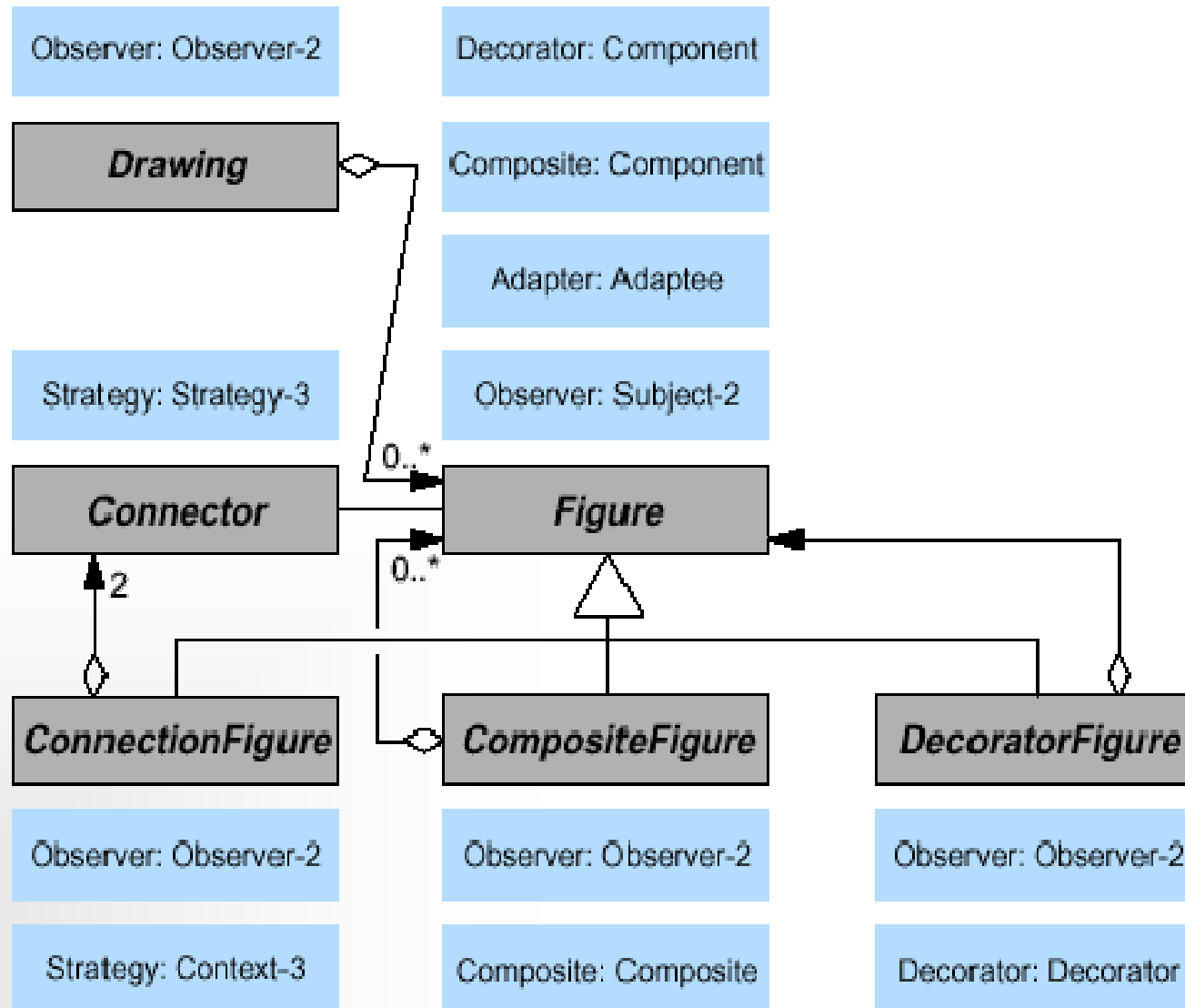
- knows its display box and can draw itself
- can be composed of several figures
- has a set of handles to manipulate its shape or attributes
- has one or more connectors that define how to locate a connection point
- can have an open ended set of attributes (key value pairs)

- *CompositeFigure* a figure that is composed of several figures
- It does not define any layout behavior
 - subclasses' responsibility
- A composite figure lets you treat a composition of figures like a single figure → **Composite pattern**

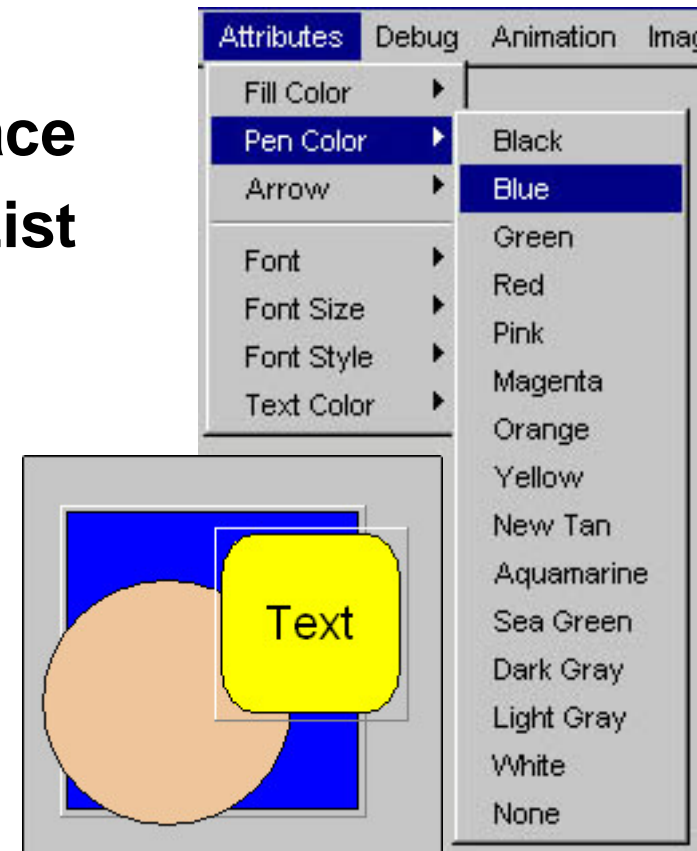
- *DecoratorFigure* is used to decorate other figures with decorations like borders.
- It forwards all the method invocations to its contained figure based on the **Decorator** pattern
- Subclasses can selectively override and add methods to extend and filter behavior

- *ConnectionFigure* connects connector objects provided by figures;
 - knows its start and end connector and uses them to locate its connection points
 - **strategy** pattern is used to encapsulate the algorithm to locate a connection point
 - *ConnectionFigure* is the **Context**
 - **Connector** is the **Strategy** participant
- **Observer** is used to track changes of connected figures
 - *ConnectionFigure* registers itself as an **observer** of the **source** and **target** figure

JHotDraw: Design Patterns in the Figures Subsystem



- Large number of attributes of different types dependent on figure type
 - would lead to a huge interface
 - alternative: use a **PropertyList**



- *Tool*

- defines a mode of a drawing view
- all input events targeted to the drawing view are forwarded to its current tool.
- tool plays the role of the **State**. It encapsulates all state specific behavior. A drawing view plays the **Context** role of the **State** pattern.

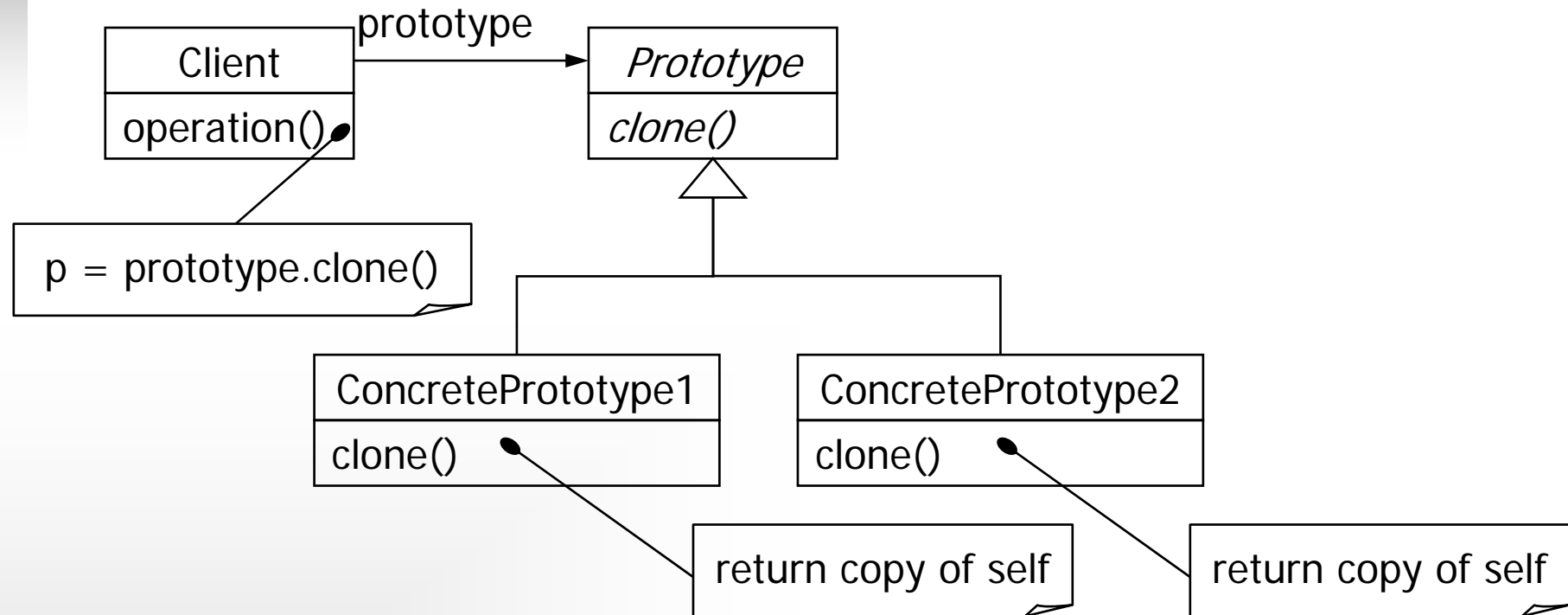
- ***SelectionTool***: used to select and manipulate figures
- It is in one of three states:
 - background selection,
 - figure selection, or
 - handle manipulation
- Different child tools handle the different states - ***SelectionTool*** is the **Context** and a child tool is the **State** participant of the **State pattern**. A selection tool delegates state specific behavior to its current child tool

- Creation would appear to require a great number of tools
 - one for every type of figure
- *CreationTool* uses **Prototype Pattern**
 - is instantiated with prototype of figure to create
 - can have multiple creation tools by configuration

Design Pattern: Prototype

- Intent
 - Create new objects by copying a prototypical instance
- Participants
 - Prototype: declares an interface for cloning itself
 - ConcretePrototype: implements the cloning
 - Client: creates objects by asking them to clone themselves

Design Pattern: Prototype



JHotDraw: Handles

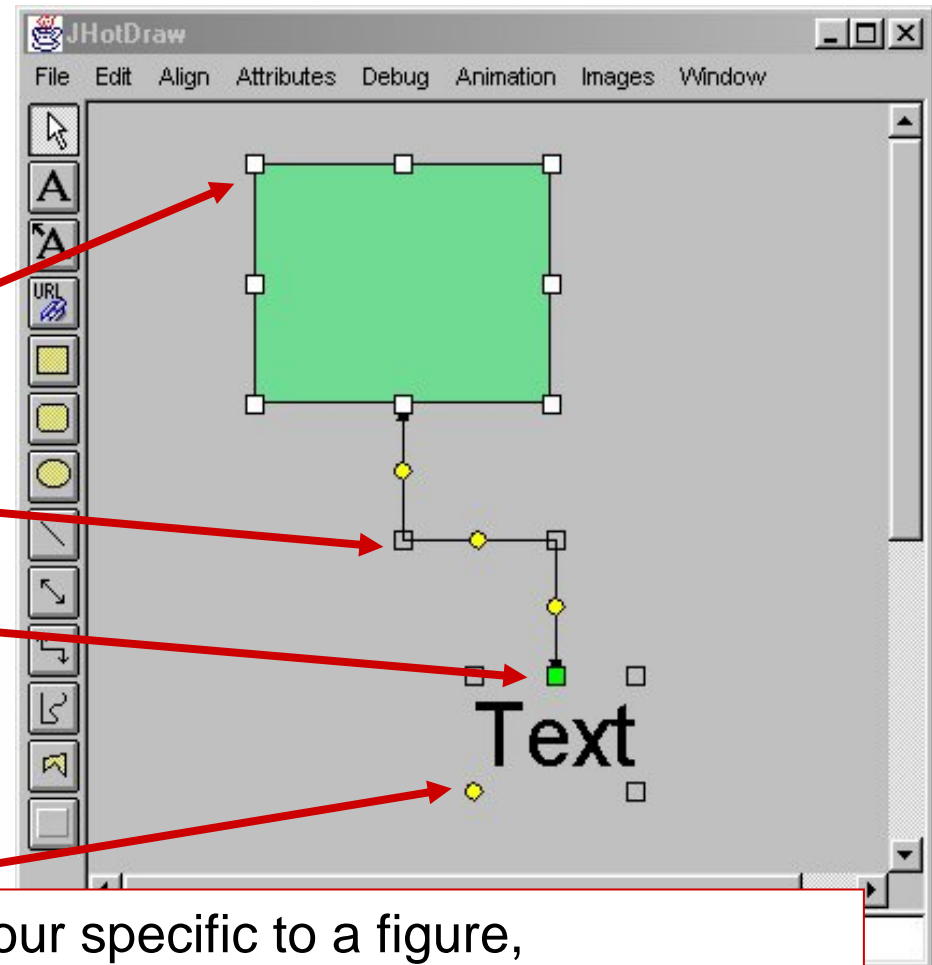
4 different appearances of handles representing different behaviors

white are the default handles in JHotDraw and provide resize functionality

empty - null handles

green in the top middle of the TextFigure allows the connector to be disconnected from the figure

yellow dots provide custom behaviour specific to a figure, i.e., on the ElbowConnector they allow each segment of the connector to be repositioned independently while on the TextFigure the single yellow dot changes the font size of the text.



- A **Handle** is an **Adapter** for **Figures**
- **Handle** interface defines the important behaviours
 - get location where to paint handle
 - get owner figure
 - hit testing
 - call back methods for user interaction
- Several types of handle predefined
 - *ChangeConnectionHandle*, *ElbowHandle*, *LocatorHandle*, and *PolygonHandle*

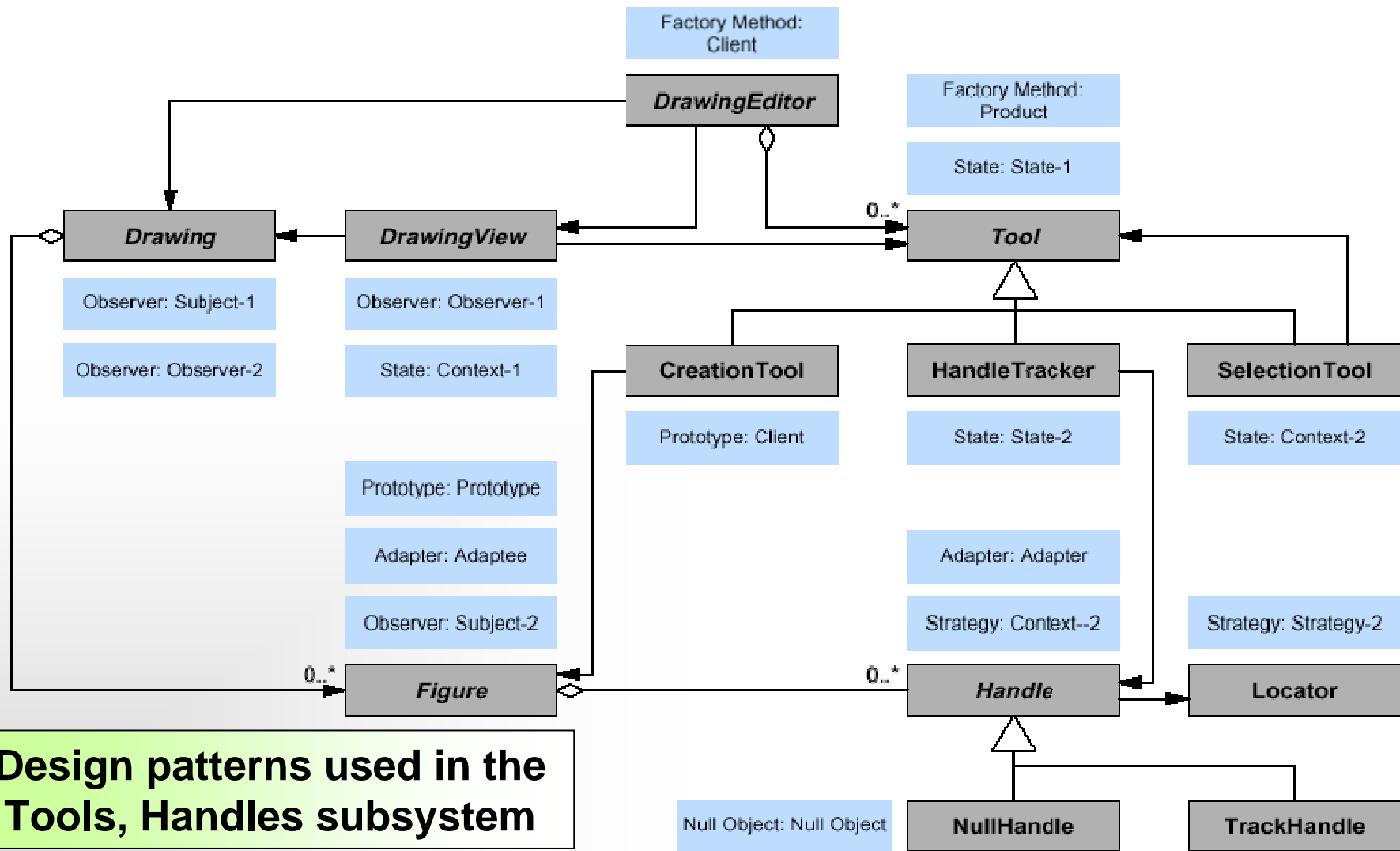
- ***NullHandle***: a handle that does not change the owned figure.
 - its only purpose is to show that a figure is selected.
 - lets you treat handles that do not do anything in the same way as other handles.

- Strategy of calculating the handle's position relative to figure may be shared among handles
- *LocatorHandle* delegates location requests to a locator object
- *Locator*: used to locate a position on a figure
 - locators encapsulate a strategy to locate a handle on a figure
 - may be shared among handles and other classes

- To define new kinds of handles, developers will subclass *AbstractHandle*
 - override `draw()` method to change appearance
 - overriding the `locate()` to change positioning

- The main behavior of a handle is defined in three methods, each called by a user initiated event :
 - **invokeStart()** is called whenever the user clicks the mouse down on a handle,
 - **invokeStep()** when ever a user drags a handle
 - **invokeEnd()** when ever the user releases the mouse button.
 - **Template Method** pattern

JHotDraw: Tools, Handles



Design patterns used in the Tools, Handles subsystem

SomethingApp.java

```
public class SomethingApp extends DrawApplication {
    public SomethingApp() {
        super("Something");
    }
    protected void createTools(Palette palette) {
        Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton( IMAGES+"TEXT",
                                    "Text", tool));
        palette.add(createToolButton( IMAGES+"MYICON",
                                    "MyCreation Tool", tool));
        tool = new ConnectionTool(view(), new LineConnection());
        palette.add(createToolButton( IMAGES+"CONN",
                                    "Connection Tool", tool));
    }
    public static void main(String[] args) {
        DrawApplication window = new SomethingApp();
        window.open();
    }
}
```

Implement primitive method used by template method in DrawApplication

Use predefined components

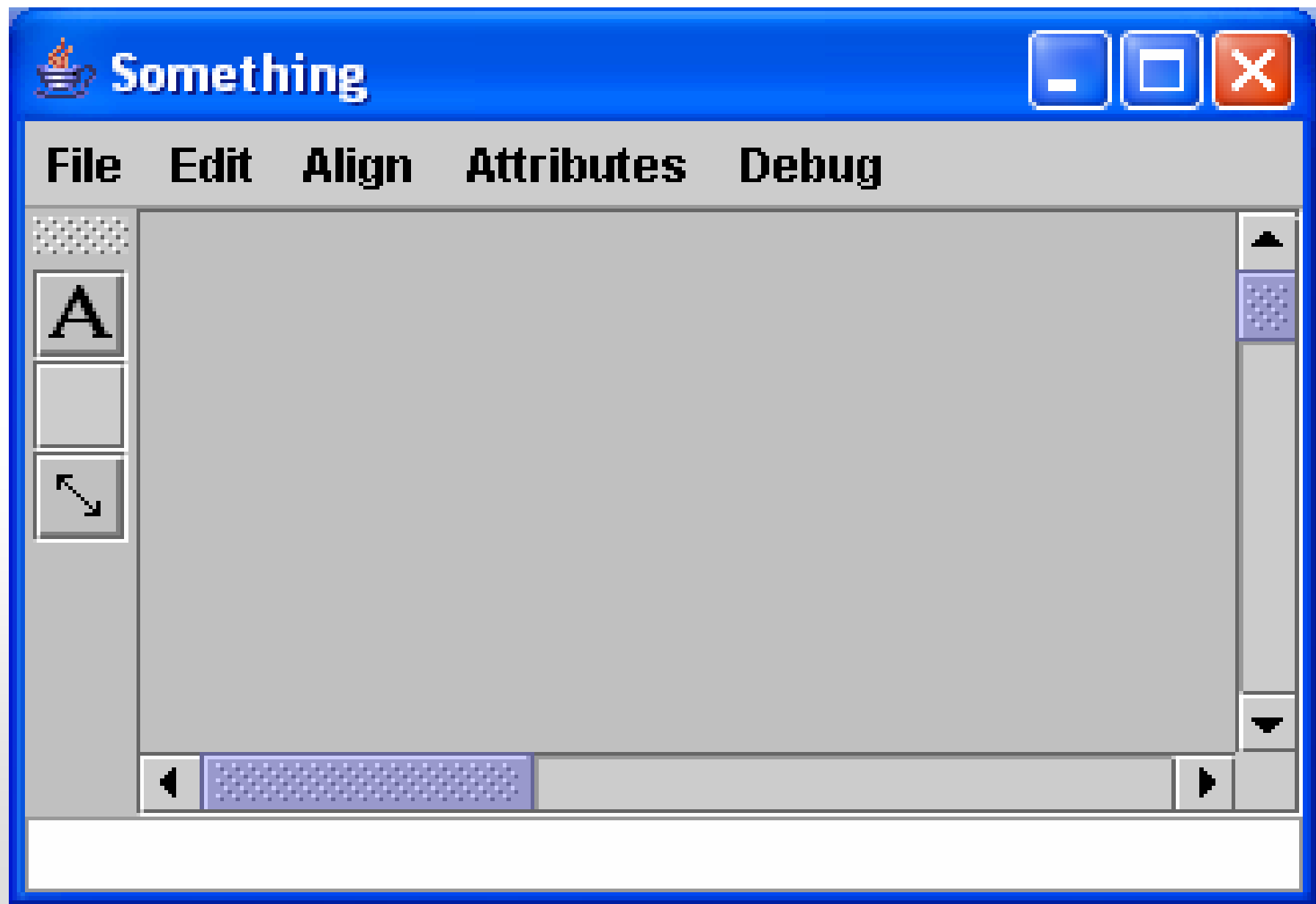
Custom defined component

Use prototype pattern

Customizing client

Use client

SomethingApp.java



Case Study: GEF



- Graphical Editing Framework
- An Eclipse Project
- Model-View-Controller architecture

GEF: The model



- Model is domain specific
- GEF knows nothing about the model
- GEF communicates with the model via Commands
 - Conceptually, Commands are part of the model
- The model notifies GEF employing the Observer pattern

GEF: A split model



- GEF needs to know, where to paint model elements
- But, position is not necessarily an attribute of the element in the real world
- Avoid polluting the model with diagram specific information by splitting it into
 - Business Model: base model storing only domain specific information
 - View Model: a “view” on something in the business model

GEF: A split model



- Solution
 - View-Model elements are adapters of the Business-Model elements
 - The controller knows about the View-Model elements
 - View-Model elements and Business-Model elements know each other

GEF: The view



- Ready to use views:
 - Tree viewer: uses SWT TreeItems
 - Graphical viewer: uses Draw2D figures
- Figures can be used as they are, subclassed, or composed
- Configured by installing a factory for controllers
 - factory gets a model object
 - returns controller for the model object

GEF: The controller

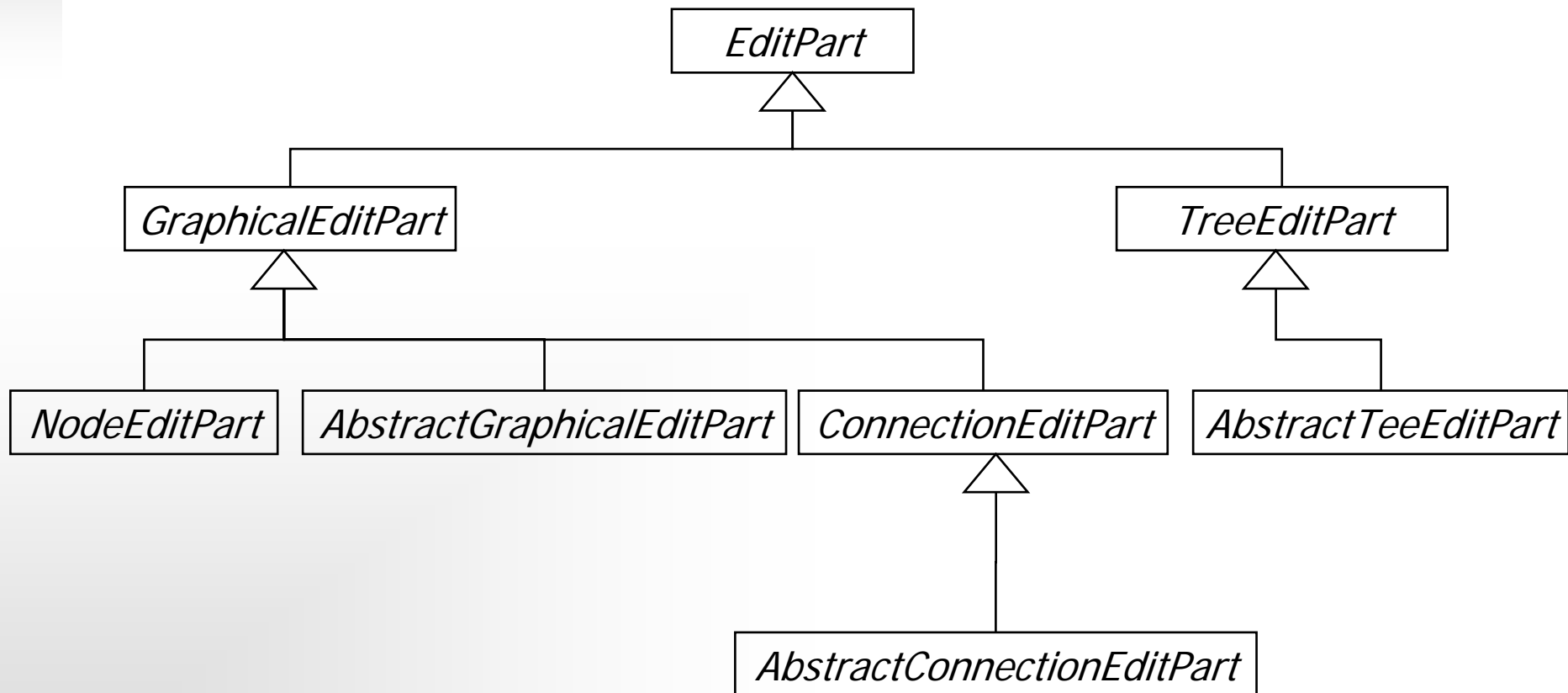


- The controller bridges the model and the view
- For each instance of a model element there is one instance of an EditPart (the controller)
- Responsible for
 - populating the viewer
 - maintain visuals as model changes
 - performing graphical editing
(manipulate the model, display feedback)

GEF: The controller



- Controller is customized by sub-classing



GEF: The EditDomain



- The EditDomain ties everything together
- Collective state of GEF application
(Command-Stack, EditPartViewers, active Tool)
- Mediates between Tools, Viewer and Command
- Decoupled from controller to gain flexibility
(several editors with own controllers can share EditDomain)