2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Grading
- Graphical User Interface (GUI) design in Java:
  - Components
  - Events and Event Listeners
- Design Patterns (in general)
- Composite design pattern
- Façade design pattern

Grading in 2IP15

- See course webpage.
- Instructors grade large programming assignment and exam.
- Grading criteria for large programming assignment can vary a bit:

  Instructors may place their own emphasis (attend instructions).
- Study material, including books, slides, handouts, such as

  Programming in the Large with Design patterns by Eddie Burris

  Checklist for Larger OO Programs
- Large programming assignment is opportunity to demonstrate what you have learned.

Textual versus Graphical User Interface

Textual User Interface: The program controls the user

Graphical User Interface (GUI): The user controls the program:

- The user generates events through keyboard, mouse, etc.
- The main event loop dispatches events,
  by calling appropriate event handling methods (event handlers).
- Each event handler responds to events.

This way, the control flow in the program is partly invisible, hidden in the main event loop: the program consists of initialization code and a bunch of event handlers, called by the main event loop.

Graphical User Interfaces in Java

Java Foundations Classes:

- Abstract Window Toolkit (AWT): translates to native GUI.
  Offers the *Look & Feel* of the host operating system.

  Package: `java.awt`

- Swing: a 'light-weight' all-Java GUI library.
  Offers run-time selectable platform-independent *Look & Feel*.

  Package: `javax.swing`

- 2D Graphics and Imaging

- Drag-and-Drop, multi-threading (concurrency), ...

- Accessibility, Internationalization, ...

GUI Organization

- Components are the building blocks of a GUI: frames (windows), panels, labels, buttons, check boxes, text fields, text areas, ...
- Components hava a *hierarchical* (tree-like) organization: Container components can contain other components.
- A component can generate various events.

  To handle such events, a listener needs to implement the relevant listener interface(s) and must be registered with the component.
- Each component *paints* itself.

  The default paint behavior can be overridden to provide application-specific graphics.

Designing the Look of a GUI in NetBeans

- For each window, create a new `JFrame` descendant via

  File > New File... > Swing GUI Forms > JFrame Form

  and put it in the `gui` package.

  This adds a `.java` and related `.form` file to the project.

- In the IDE, select

  Source to work on the code view of the window frame

  NetBeans controls the `Generated Code`.

  Design to work on the graphical view of the window frame

  Drag components from the *Palette* to the frame, and set their properties in the *Properties* panel.

Designing the Feel of a GUI: Handling Button Events

- To handle button events, use `addActionListener()` to register a listener that implements interface

  `ActionListener`

  `actionPerformed(ActionEvent e)`

- A click on the button results in a call to `actionPerformed()` of each registered listener (callback: handler = strategy).

- In NetBeans, double clicking the button in Design mode will add the relevant Java code, where the body of `actionPerformed()` calls an *event handler* in the form that contains the button.

  You can supply a body for that event handler.

- Example: `InterestCompounder`

Designing the Feel of a GUI: Handling Mouse Events

To handle mouse events, create listeners implementing interfaces

MouseListener	**MouseMotionListener**
`mouseClicked(MouseEvent e)`	`mouseDragged(MouseEvent e)`
`mouseEntered(MouseEvent e)`	`mouseMoved(MouseEvent e)`
`mouseExited(MouseEvent e)`	
`mousePressed(MouseEvent e)`	
`mouseReleased(MouseEvent e)`	

(`MouseInputListener` extends both these interfaces)

and register them as listener:

```
component.addMouseListener(listener);
component.addMouseMotionListener(listener);
```

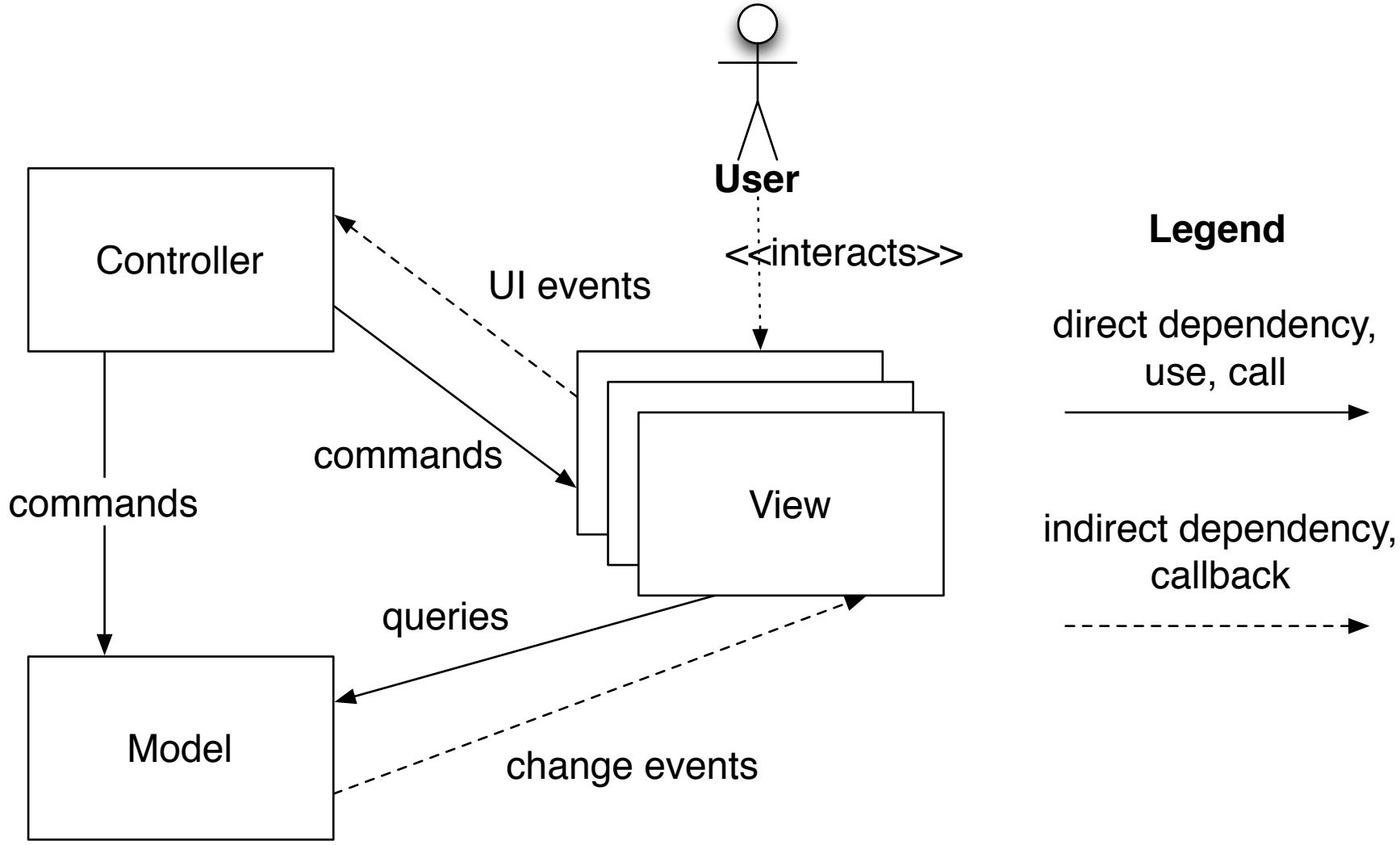
A `MouseEvent` provides `getPoint()` to obtain the mouse location.
See API for `java.awt.event.MouseEvent` for details.

Handling Some Mouse Events, but not all: Adapters

- When defining a listener object for a listener interface from scratch, *all* event methods must be implemented.
- An adapter class provides empty implementations for all events in a listener interface.
- A listener object can be created by instantiating a descendant of the adapter class and overriding some event methods:

```
1     jPanel1.addMouseListener(
2         new MouseAdapter() {
3             public void mousePressed(MouseEvent evt) {
4                 jPanel1MousePressed(evt);
5             }
6         }
7     );
```

Model–View–Controller Architecture



GUI Design in Java using Swing

- Manual design: see David Eck, Chapter 6 (and 13)
- Tool-supported design: NetBeans

Concepts involved:

- Layout Managers
- Hierarchical structure: components containing other components
- Properties of components: visual, behavioral
- Event handling: listeners, listener interfaces, registering

Graphics in Java using Swing

- A component paints itself via its `paintComponent(Graphics g)`.

  This method is called automatically whenever "needed".
  (It is called by `paint()`, which itself is a callback method.)
  Via `repaint()`, the application can request a `paint()` call.
  There is no guarantee when this is request will be honored.
  Multiple `repaint` requests may get merged into one update.

- Override `paintComponent()` to replace the default paint behavior.

  The new definition must first call **`super`**`.paintComponent(g)`.
  `Graphics g` is used for drawing: `g.drawXXX(...)`.

- `JFrame` uses `paint()` instead of `paintComponent()`.

  However, it is not recommended to override its `paint()` method.
  Instead, draw on a component (like a `JPanel`) inside the frame.

GUI Facilities and Object-Orientation

- The organization of GUI components and events is a prime example of Object-Oriented (OO) design.
- Polymorphism, specialization through inheritance
- Events, listeners, adapters
- Recurring *patterns* in solutions to make the facilities generic
- These patterns are also useful in other contexts.

Binary Puzzle Assistant: Development Steps

1. Model: stores state of the puzzle (grid with symbols), providing
   - constructor(s) to create it,
   - queries to inspect it, and
   - commands to change it
2. View: presents graphical/textual rendering of the state, generating view-based events for user initiated operations
3. Controller: responds to user actions to load, change, save model
4. Solvers: support automatic solving

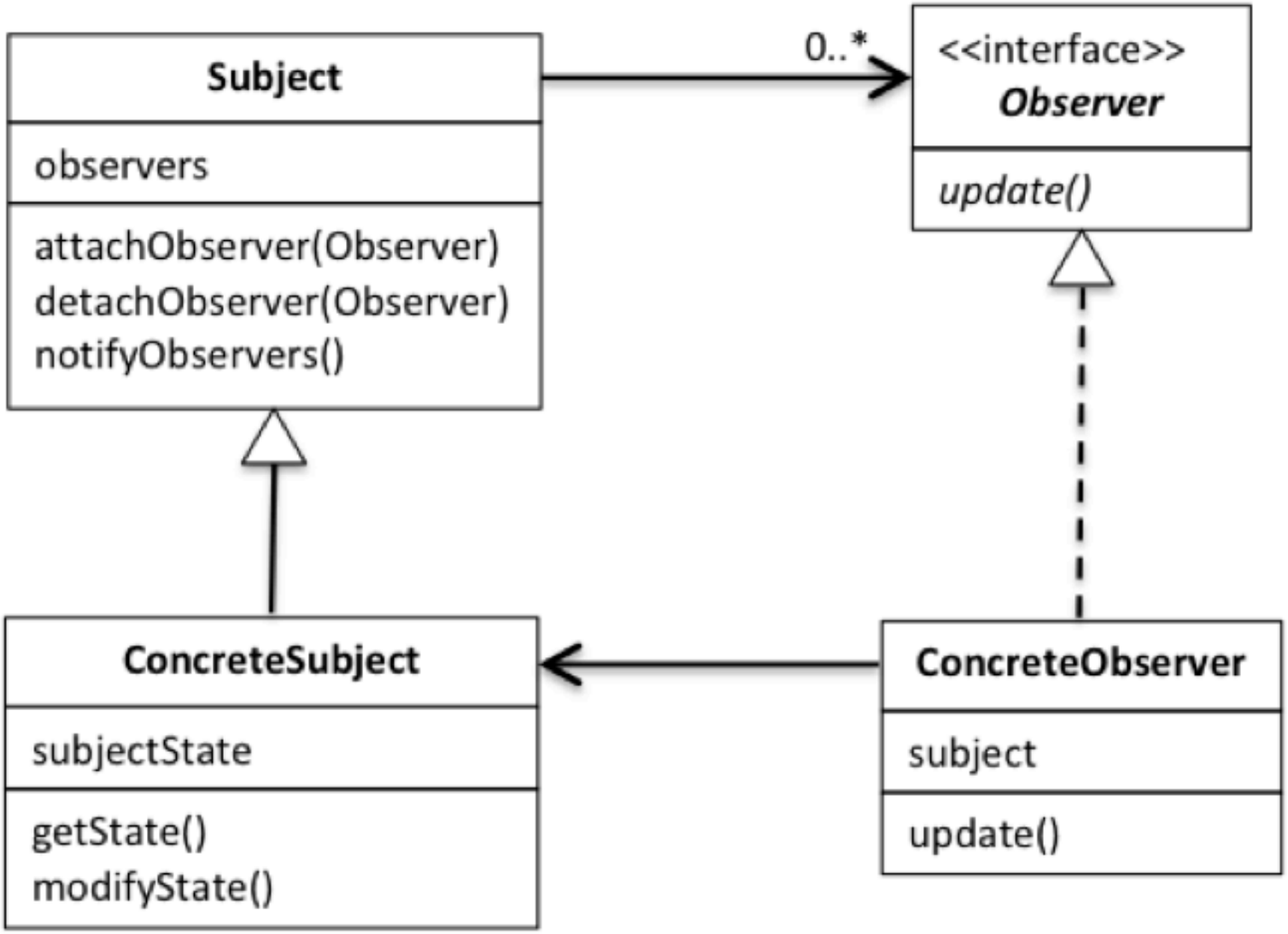
Design Patterns

- A Design Pattern is an *outline* of a *general* solution to a design problem, *resuable* in *multiple, diverse, specific* contexts.
- It is *not* a solution in the form of source code (such as a *library*) that can be reused "as is".
- Description of design pattern:
  - Name, to aid communication
  - Context, constraints
  - Problem, intent, forces,
  - Solution, roles, relations, to guide use in specific context
  - Examples
  - Antipatterns, related patterns

Observer Pattern: Description (from Burris)

- Name: Observer
- Context: One or more objects (observers) need to be made aware of state changes in another object (the subject).
- Problem: The objects doing the observing (observers) should be decoupled from the object under observation (the subject).
- Solution: Define a class or interface `Subject` with methods for attaching and detaching observers, as well as a method for notifying attached observers, when the state of the subject changes. Define an interface `Observer` that defines a callback method that subjects can use to notify observers of a change.

Observer Pattern: Class Diagram (from Burris)



How to Study Design Patterns

- Read chapter in the book by Eddie Burris
- Analyze the structure: roles, relationships
- Understand why it solves the problem; why antipatterns don't
- Be critical, understand the why
- Consult other sources

  E.g., `en.wikipedia.org/wiki/Software_design_pattern`
- Answer review questions (on book's website)

Taxonomy of Design Patterns

- Creational patterns

  Factory, Singleton

- Structural patterns

  Composite, Façade, Adapter, Decorator

- Behavioral patterns

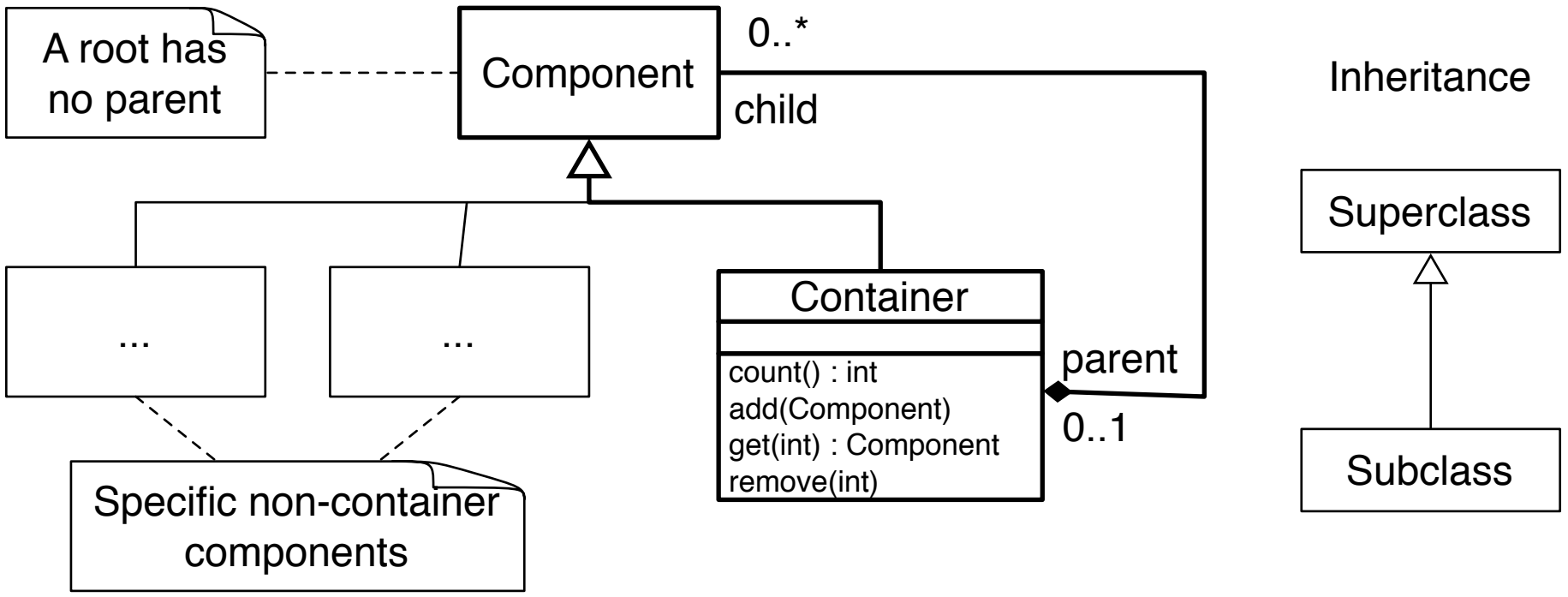
  Strategy, Iterator, Observer, Command, State, Template Method

- Concurrency patterns

  "`SwingWorker`"

The Composite Design Pattern for Hierarchical Structure

- See `http://en.wikipedia.org/wiki/Composite_pattern`
- Example: GUI components and containers in Java AWT/Swing

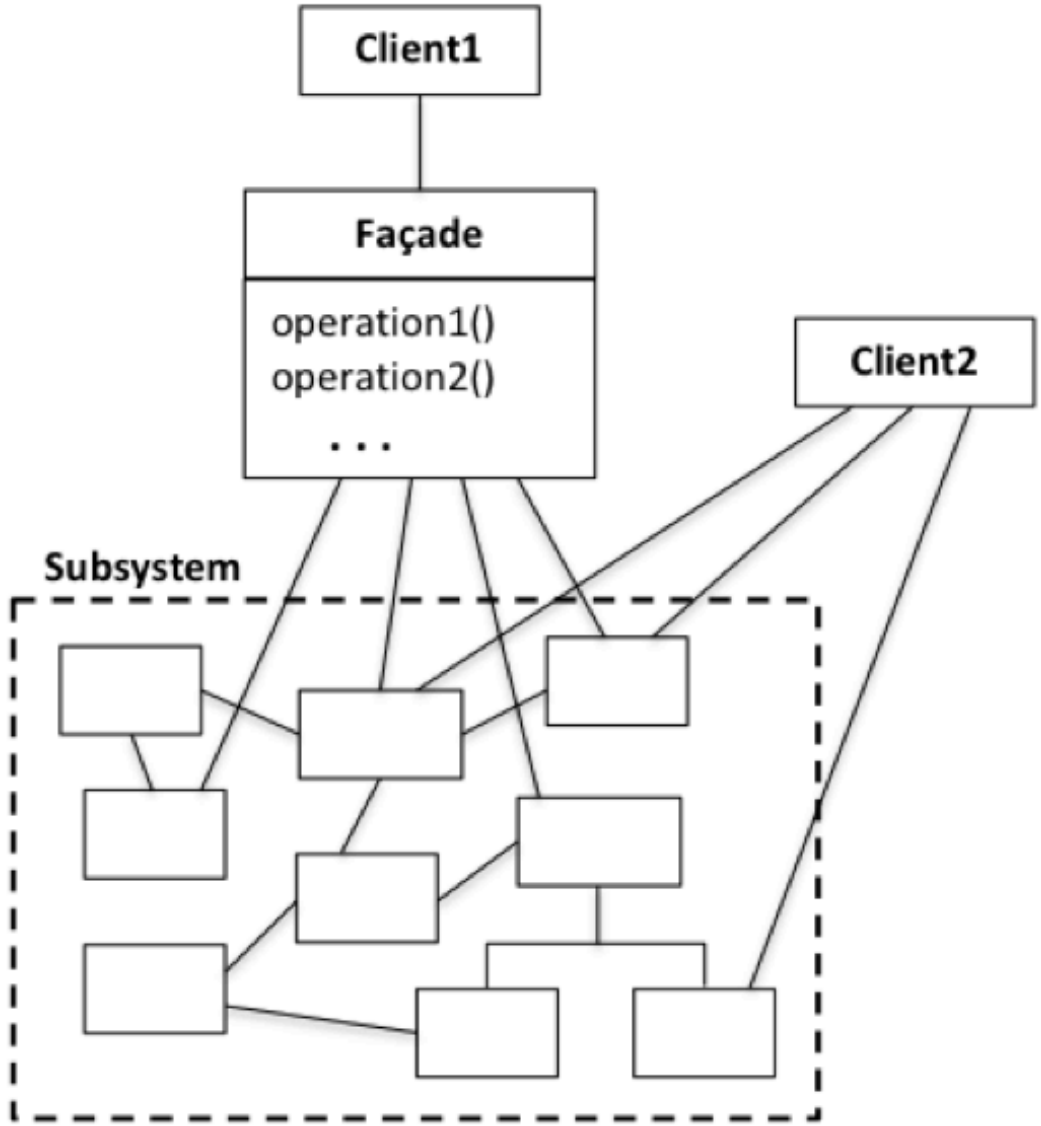


Façade Design Pattern

- Problem: Avoid high coupling to multiple classes in subsystem.
- Bad solution: Make client depend on multiple internal classes.
- Preferred solution:
  - Provide unified interface to set of interfaces in subsystem.
  - A façade defines an abstract interface, making the subsystem easier to use.
  - The façade merely delegates to internal classes.
- Example: Façade for Model in MVC architecture

  Binary Puzzle

Façade Pattern: Class Diagram (from Burris)



Summary

- GUI components, events, and listeners
- Design Pattern in general
- Design Pattern: *Façade*
- Design Pattern: *Composite*