Advanced Software Design
Design Patterns

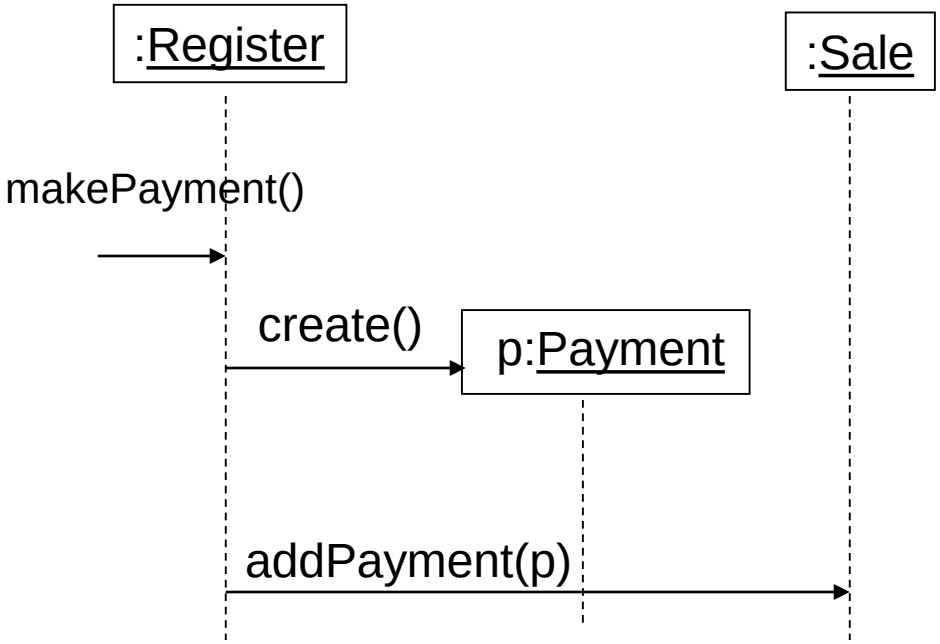
Gary Stewart
gstewart@cs.uct.ac.za
slides taken from prof. edwin blake

High Cohesion

- Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work.
- Problems because of a design with low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Delicate, affected by change.
- **Problem**: How to keep complexity manageable?
- **Solution**: Assign a responsibility so that cohesion remains high.

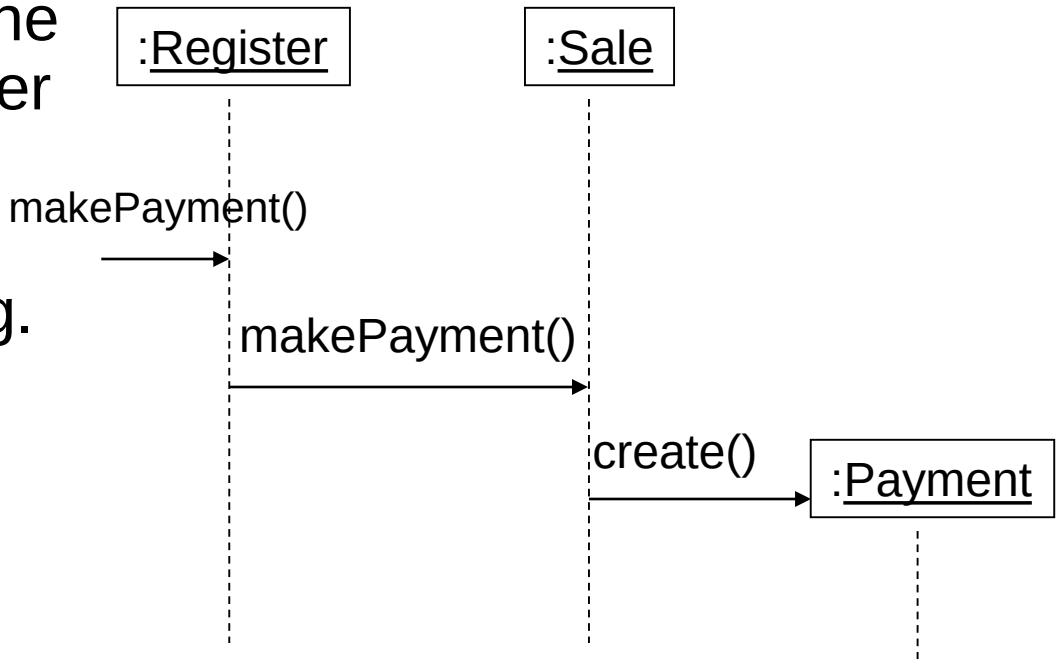
High Cohesion

- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.



High Cohesion

- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.



High Cohesion

- Scenarios that illustrate varying degrees of functional cohesion
- Very low cohesion: class responsible for many things in many different areas.
  - e.g. a class responsible for interfacing with a database and remote-procedure-calls.
- Low cohesion: class responsible for complex task in a functional area.
  - e.g. a class responsible for interacting with a relational database.

High Cohesion

- High cohesion: class has moderate responsibility in one functional area and it collaborates with other classes to fulfil a task.
  - e.g.: a class responsible for one section of interfacing with a database.
- Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates.

Coupling and Cohesion

- High Cohesion typically implies Low Coupling
- Low Cohesion typically implies High Coupling

Quick Summary of the remaining patterns

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations
  (Don’t talk to strangers)

Polymorphism

- How to design for varying, similar cases?
- Assign a polymorphic operation to the family of classes for which the cases vary.
- E.g., draw() which can be called for various shapes
  - Square, Circle, Triangle

Pure Fabrication

- Where to assign a responsibility, when the usual options based on Expert lead to problems with coupling and cohesion, or are otherwise undesirable.
- Make up an “artificial” class, whose name is not necessarily inspired by the domain vocabulary.
- E.g., in Register-Sales-Payment Example
  - ReceiptPrinter (a Pure Fabrication)

Indirection

- A common mechanism to reduce coupling?
- Assign a responsibility to an intermediate object to decouple collaboration from 2 other objects.

Protected Variations

- “Information Hiding”
- **Problem**: How to design objects so that changes in these objects do not have side effects on other objects?
- **Solution**: Put a wrapper or interface object round them. The wrapper gives a stable interface and is all that has to be altered when changes happen.

“Don’t Talk to Strangers”

- Special case of Protected Variations
- In a method, only send messages to “familiars”:

1. this or self
2. parameters
3. own attributes (including elements of collections)
4. object created in the method (local)

In other words don’t go down a chain of links to indirectly known “strangers”.

This avoids coupling to distant objects.

USE CASE REALIZATIONS

Responsibility-Driven Design

GRASP

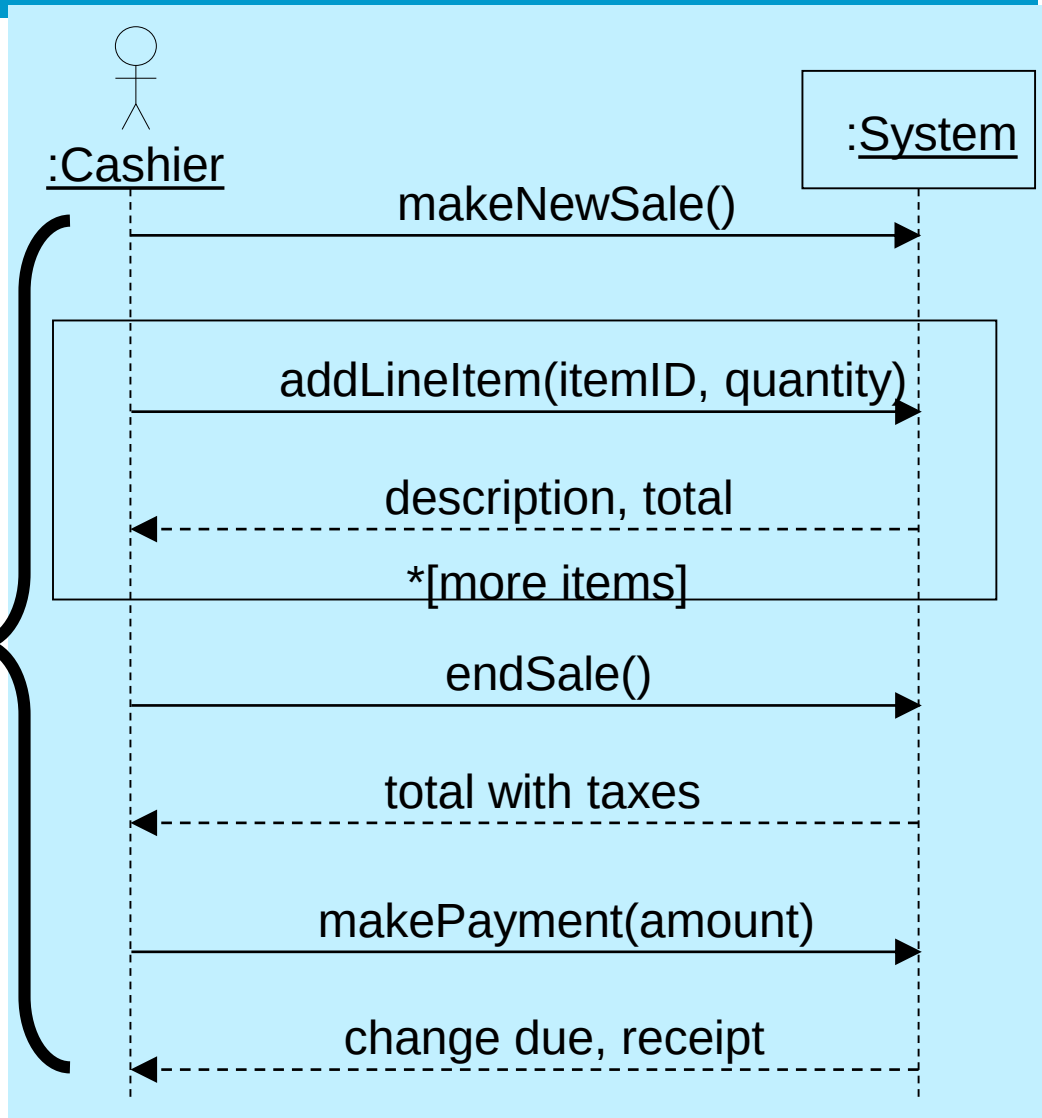
Use Case Realizations

GoF Patterns

Conclusion

Definition: Use Case Realizations

- In the vocabulary of the UP and use case modelling, when we design the objects to handle the systems operation for a scenario, we are designing a use case realization.

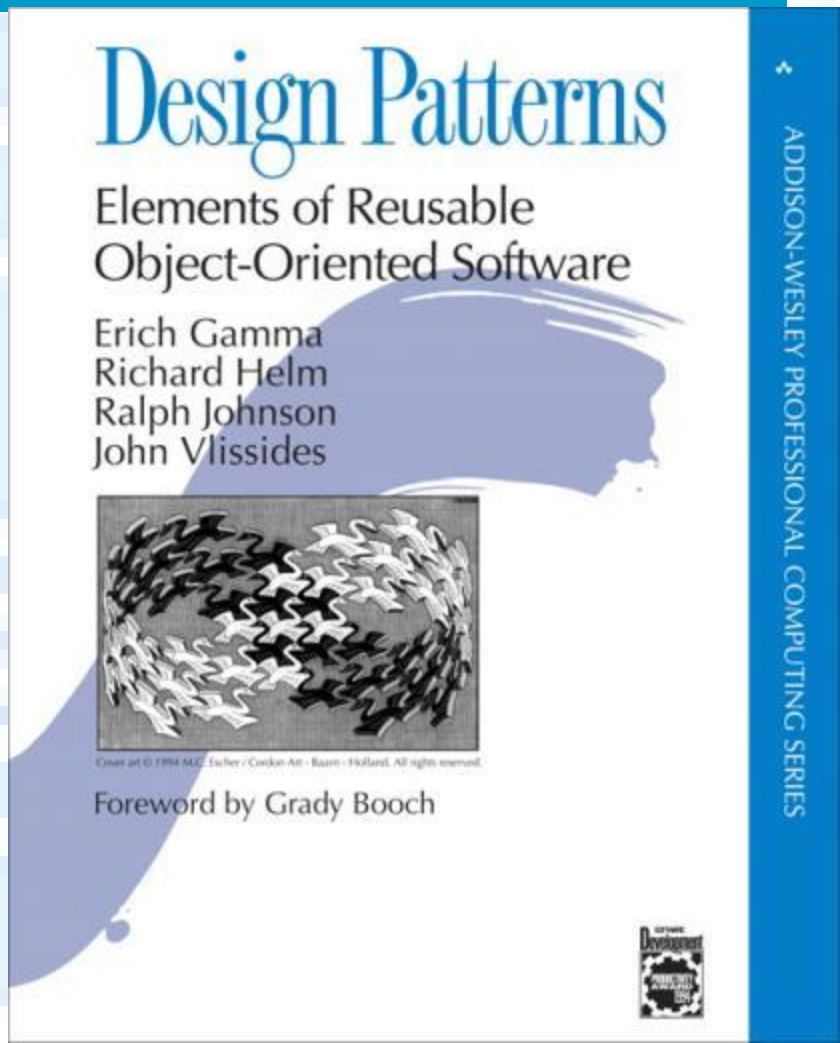


Designing for Visibility

- Fact: To send a message to B, A must have *visibility* to B. It doesn't happen by "magic."
- Not a GRASP pattern but important!
- Kinds of visibility:
  - Attribute (instance variable)
  - Parameter
  - Local variable
  - Global variable

GOF PATTERNS

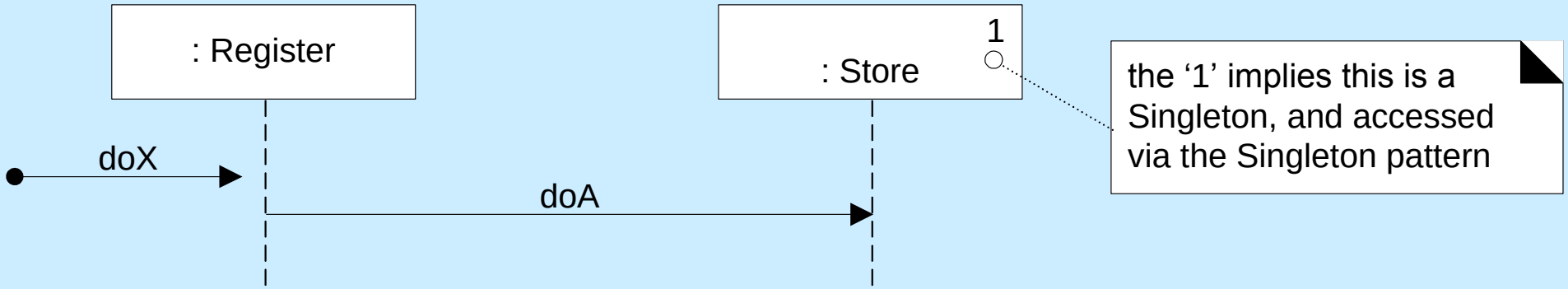
- Responsibility-Driven Design
- GRASP
- Use Case Realizations
- **GoF Patterns**
- Conclusion



Singleton Pattern

- Ensures a class only has one instance, and provide a global point of access to it
  - Sometimes we want just a single instance of a class to exist in the system.
    - want just one window manager, or one factory for a family of products.
    - need to have that instance easily accessible and we want to ensure that additional instances of the class cannot be created.
  - Benefits
    - Controlled access to sole instance

GoF: “Ensure a class has one instance, and provide a global point of access to it.”



Singleton Pattern

Only one instance of certain classes *never* two.

Structure for a unique object or sub-system.

Composite Pattern

- **Problem**: Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects uniformly.
- **Solution**: Define an abstract base class (Component) that specifies the behaviour that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

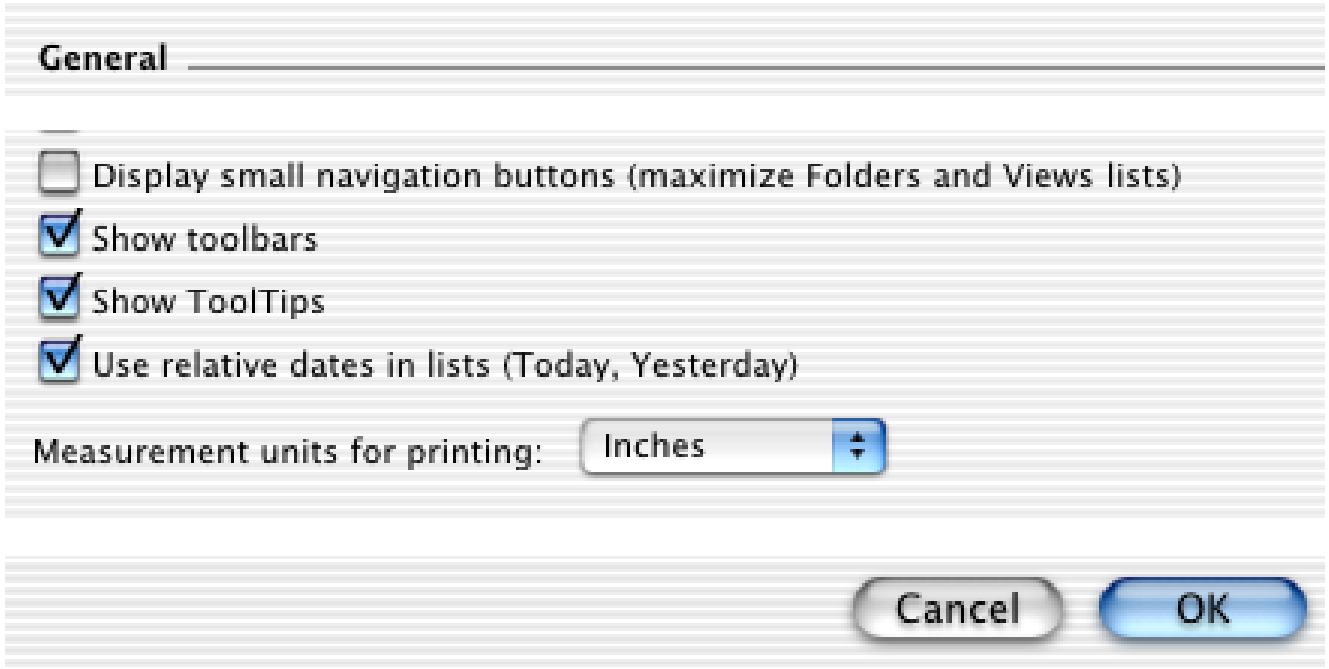
Anatomy of a preference dialog

- Aggregates (Composites), called Panels, are used for grouping user interface objects that need to be resized and moved together.

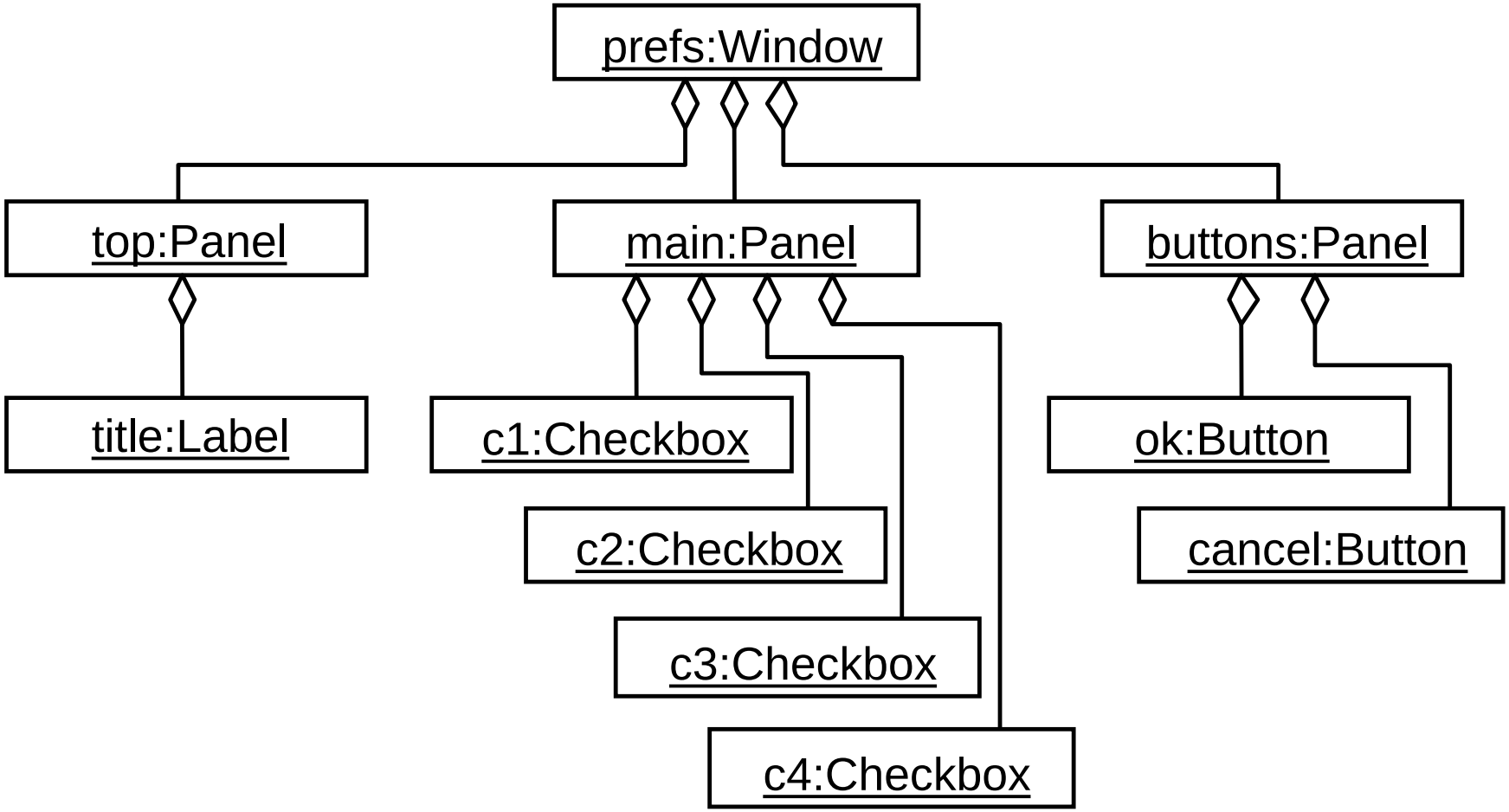
Top panel

Main panel

Button panel

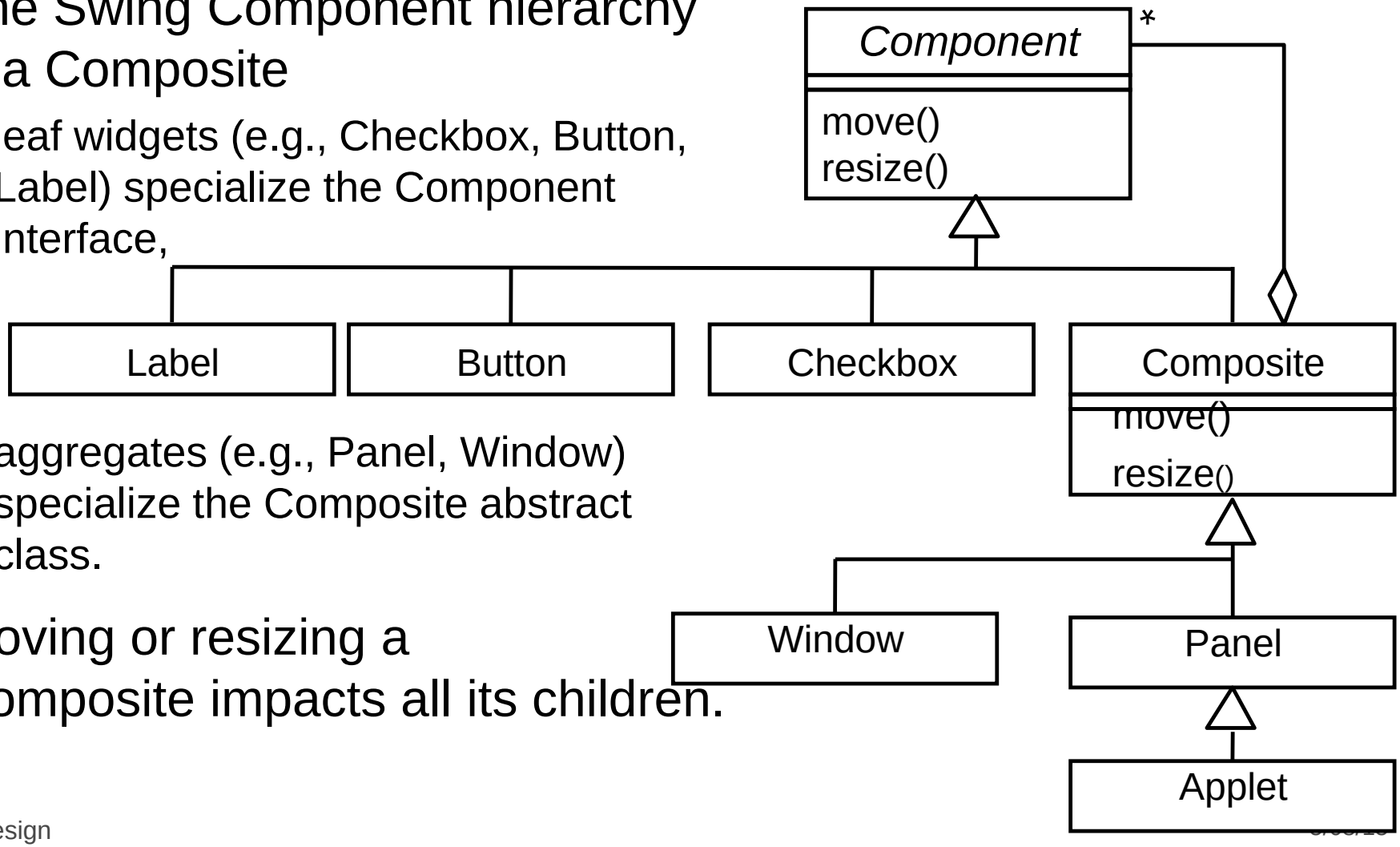


UML object diagram for UI Preference



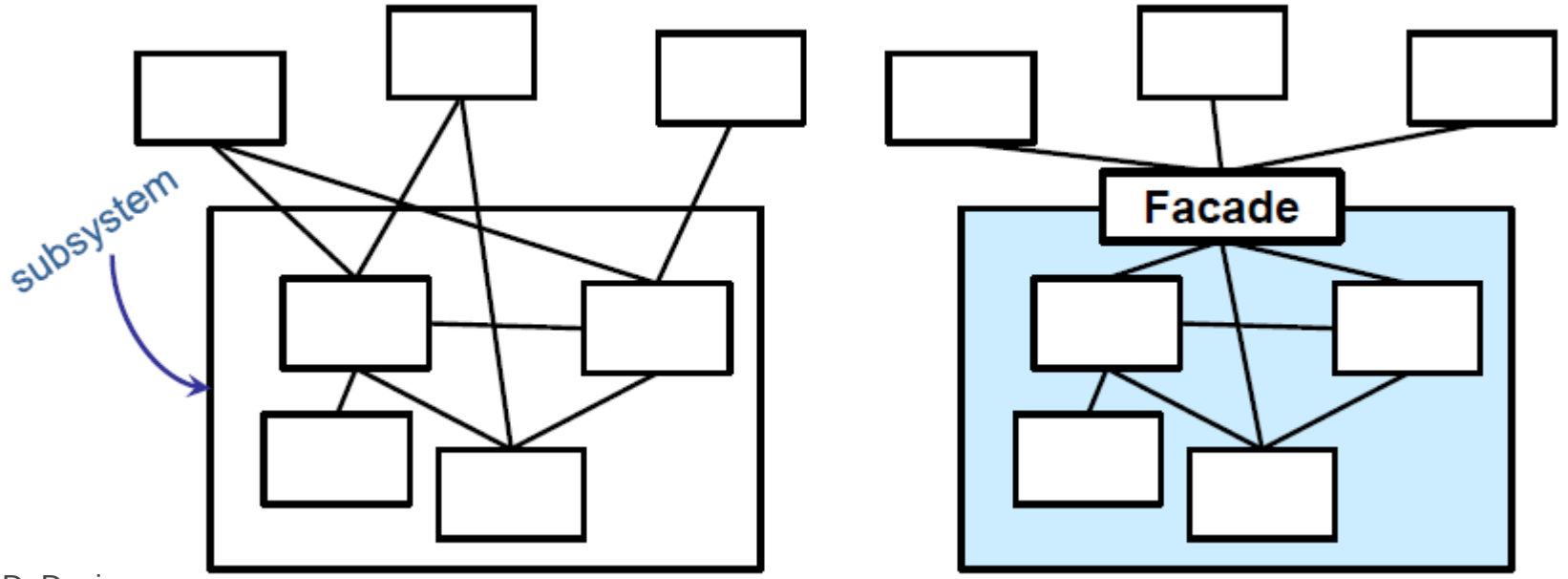
Applying *Composite* pattern to UI widgets

- The Swing Component hierarchy is a Composite
  - leaf widgets (e.g., Checkbox, Button, Label) specialize the Component interface,
  - aggregates (e.g., Panel, Window) specialize the Composite abstract class.
- Moving or resizing a Composite impacts all its children.



Façade Pattern

- Define a single point of contact to the subsystem –a façade object that wraps the subsystem. This façade object presents a single unified interface and is responsible for collaborating with the subsystem components.



Observer Pattern

- Generalization of the MVC Pattern
  - Want to display data in more than one form at the same time and have all of the displays reflect any changes in that data.
- Assumes the object containing the data is separate from the objects which display the data,
  - these display objects observe changes in that data.
- Liabilities
  - Possible cascading of notifications
  - Observers are not aware of each other and must be careful about triggering updates
  - Simple update interface requires observers to deduce changed item

Designing for Low Representational Gap

- Normally we design for a low representational gap between real world and software …
- Typically designed objects typically correspond to real world objects
- But (contra-indication!):
  - Don’t design for low representational gap regarding actors.
  - E.g., we don’t make a software class Clerk do all the work in the software system.
- Exercise: Why?

Patterns: Summary

- Principles (expressed in *patterns*) guide choices in where to assign responsibilities.
- A pattern is a named *description of a problem* and a *solution* that can be applied to new contexts;
  - it provides advice on how to apply it in varying circumstances.

Patterns Resources

- GRASP (General Responsibility Assignment Software Patterns)
  - Larman:
    - 2nd edition
      - Chapter 16
      - Chapter 17 & 22
    - 3rd edition
      - Chapter 17 ← main interest
      - Chapter 18 & 25
- GoF (Gang-of-Four)
  - Larman
    - 2nd edition
      - Chapter 23, 33 & 34
    - 3rd edition
      - Chapter 26, 36 & 37