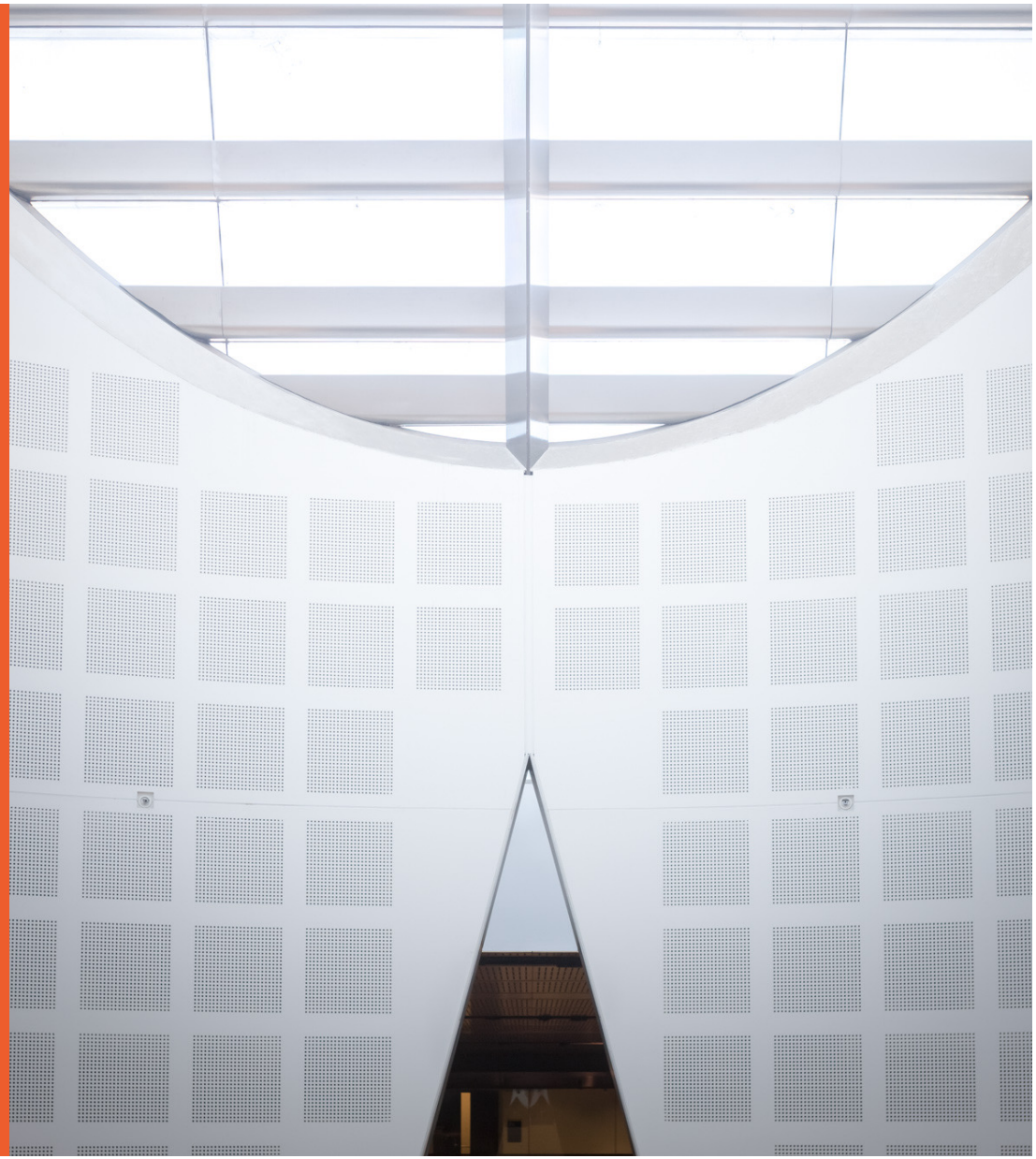
Software Design and Construction 1
SOFT2201 / COMP9201

Adapter and Observer

Dr. Xi Wu

School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

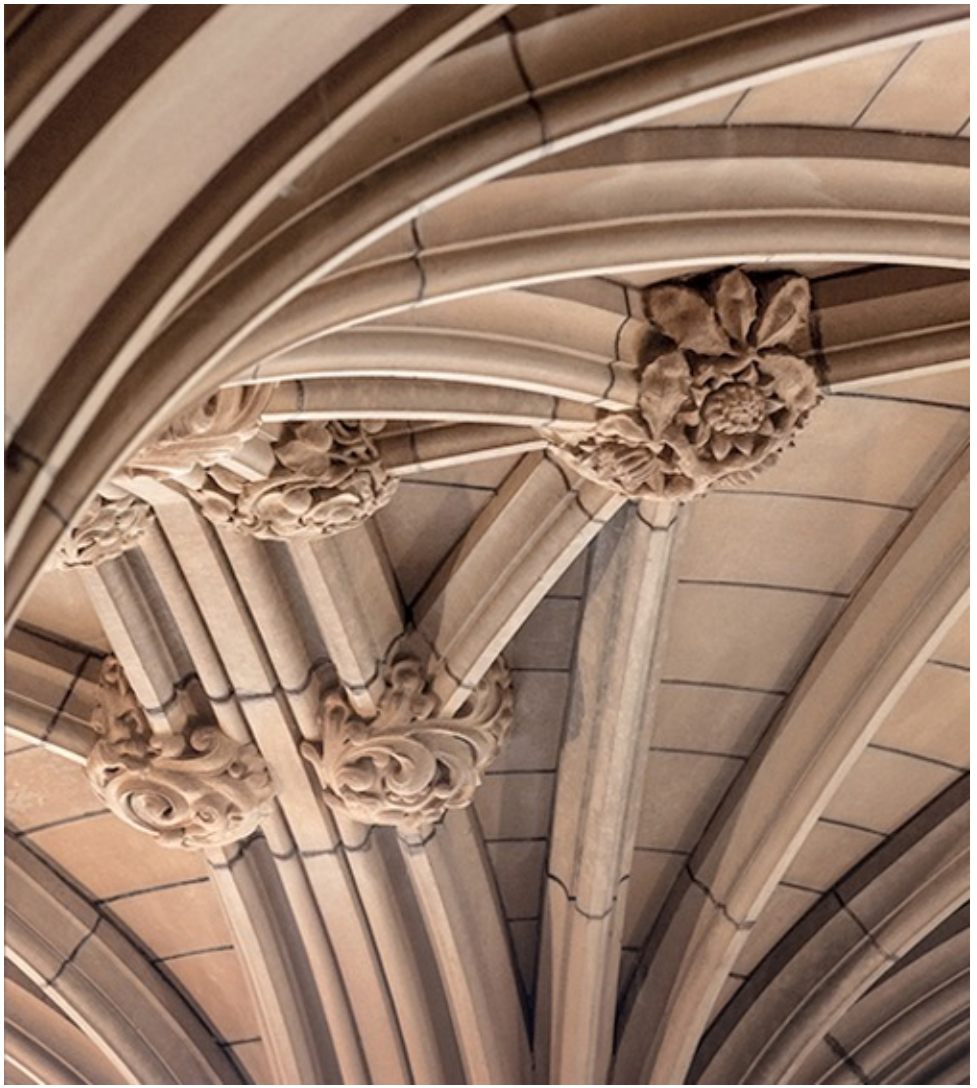
Do not remove this notice.

Agenda

- Structural Design Pattern
  - Adapter

- Behavioural Design Pattern
  - Observer

Structural Design Patterns



THE UNIVERSITY OF SYDNEY

Structural Design Patterns

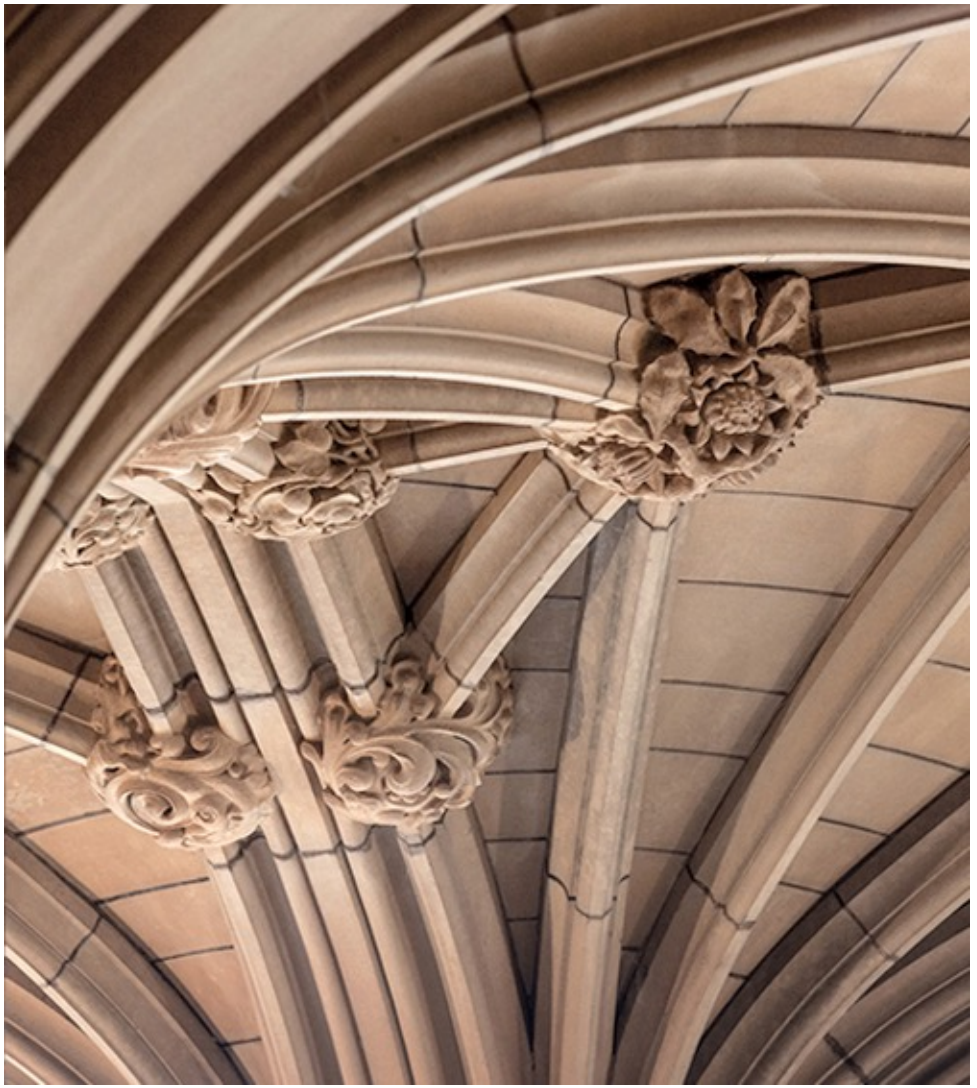
- How classes and objects are composed to form larger structures
- Structural *class* patterns use inheritance to compose interfaces or implementations
- Structural *object* patterns describe ways to compose objects to realise new functionality
  - The flexibility of object composition comes from the ability to change the composition at run-time

Structural Patterns (GoF)

Pattern Name	Description
Adapter	**Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface that clients expect.**
Façade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.
Decorator	Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)
Composite	Compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Proxy	Provide a placeholder for another object to control access to it

Adapter Pattern

Class, Object Structural



THE UNIVERSITY OF SYDNEY

Motivated Scenario

- Suppose you travel to Europe countries with your laptop you have bought in Australia.

Power Charger for Computer

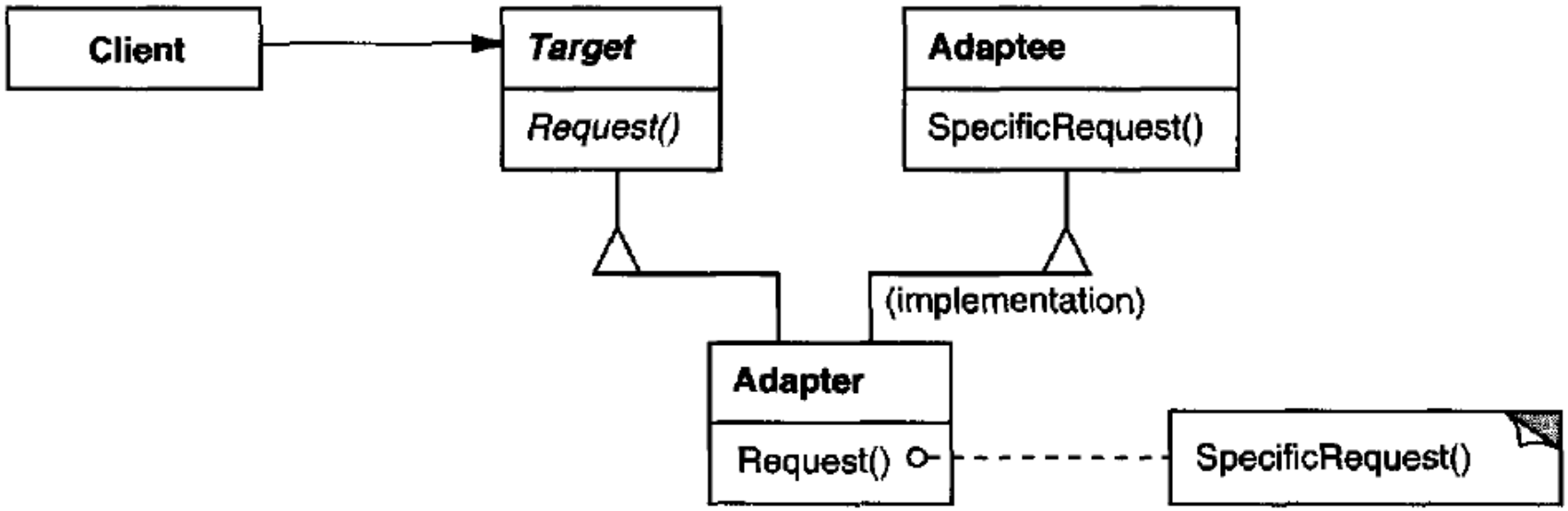
Adapter

European Power Strip

Adapter

- Intent
  - Convert the interface of a class into another interface that clients expect.
  - Lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Sometimes existing code that has the functionality we want doesn’t have the right interface we want to use
- Known as
  - Wrapper

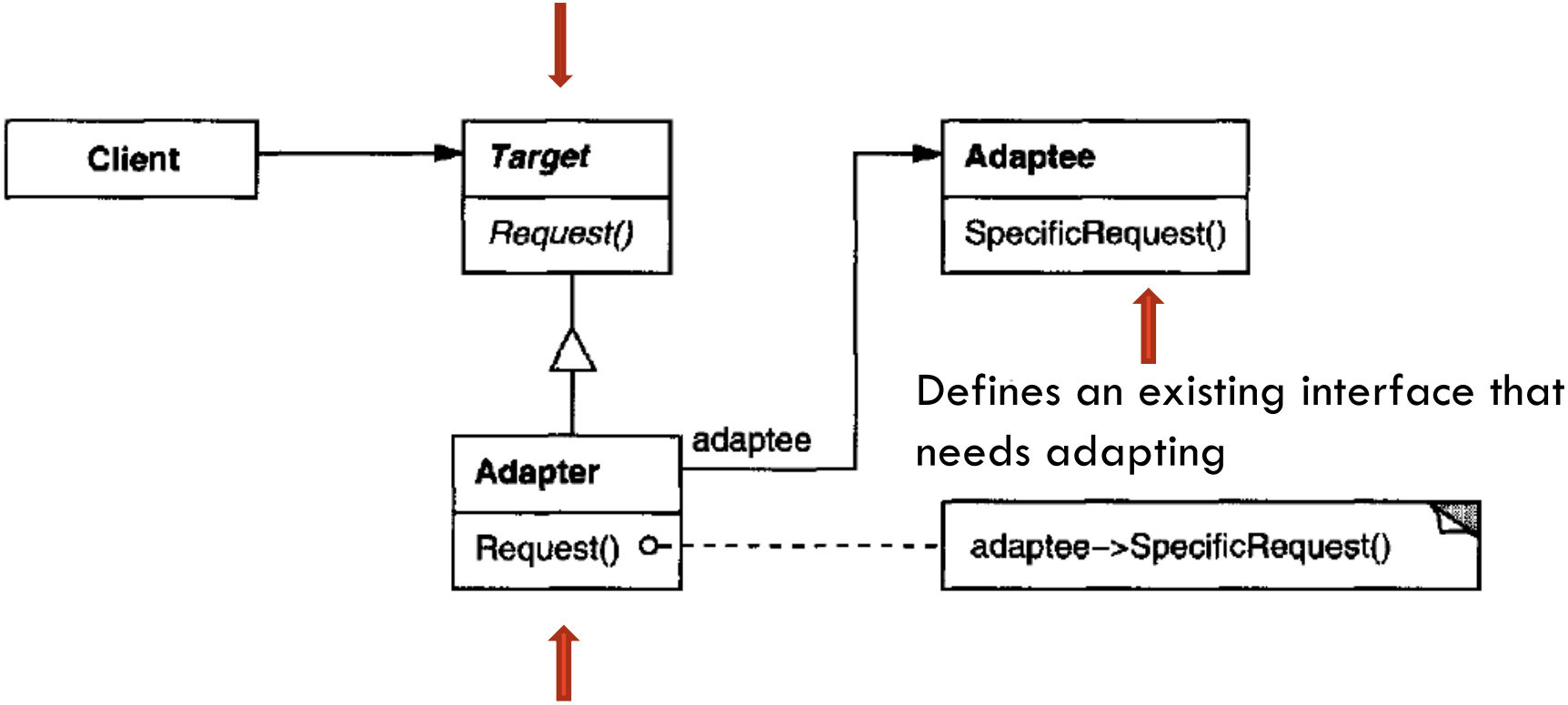
Class Adapter – Structure



- Request multiple inheritance to adapt the Adaptee to Target, **supported by C++**

Object Adapter – Structure

Defines the domain-specific interface that Client uses



Defines an existing interface that needs adapting

Adapts the interface of Adaptee to the Target interface

Adapter – Participants

- **Target**
  - Defines the domain-specific interface that Client uses
- **Client**
  - Collaborates with objects conforming to the Target interface.
- **Adaptee**
  - Defines an existing interface that needs adapting.
- **Adapter**
  - Adapts the interface of Adaptee to the Target interface

Adapter

- Applicability
  - To use an existing class with an interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, i.e., classes that don't necessarily have compatible interfaces
  - **(Object adapter only)** Adapt an existing interface, which has several existing implementations.
- Benefits
  - Code reuse
- Collaborations
  - Clients call operations on an Adapter instance. In turn, the Adapter calls Adaptee operations that carry out the request

Class Adapter – Consequences

- When we want to adapt a class *and* all its subclasses, a class adapter won't work
  - It adapts Adaptee to Target by committing to a concrete Adaptee class

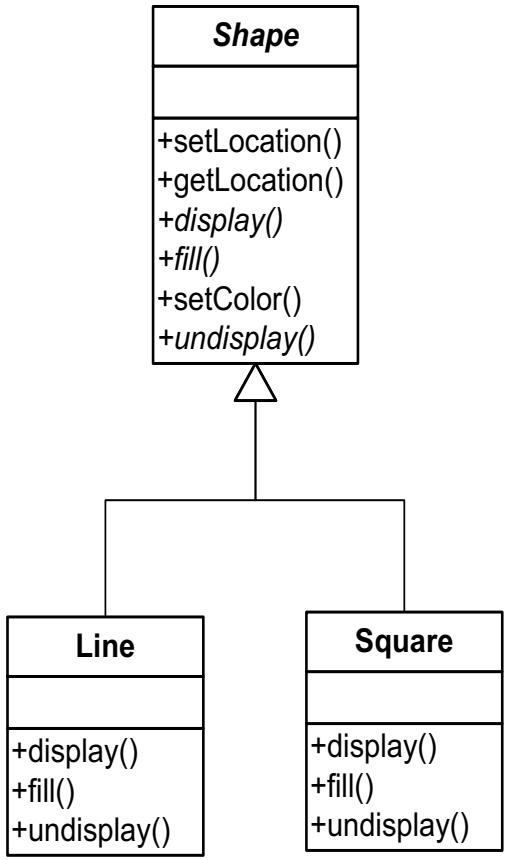
- Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee

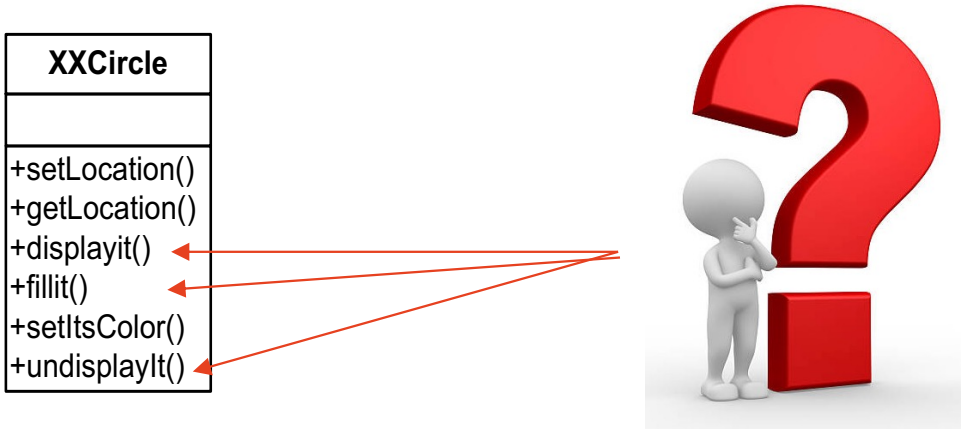
Object Adapter – Consequences

- Lets a single Adapter work with many Adaptees – i.e., the Adaptee itself and all of its subclasses (if any).

- Makes it harder to override Adaptee behavior. It will require sub-classing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself

Adapter Example -- Problem

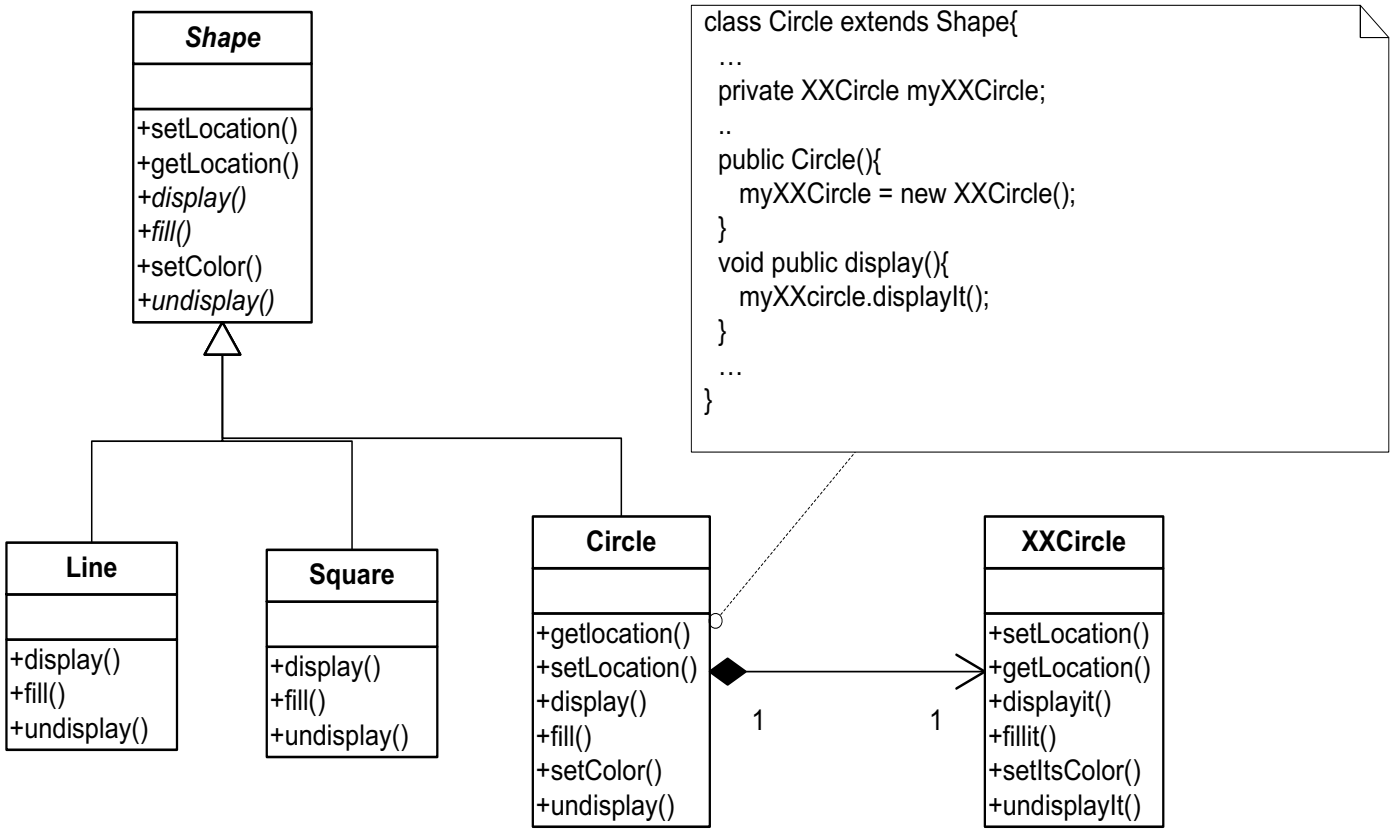




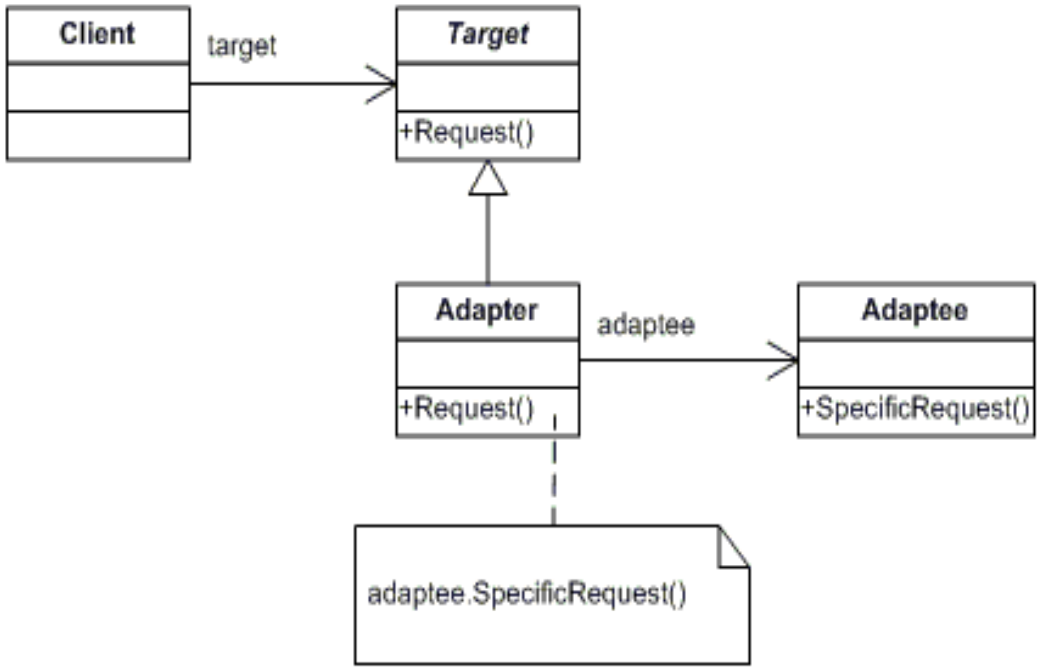
Adapter Example -- Solution

What should a client do?

Shape shape = new Circle();
shape.display();

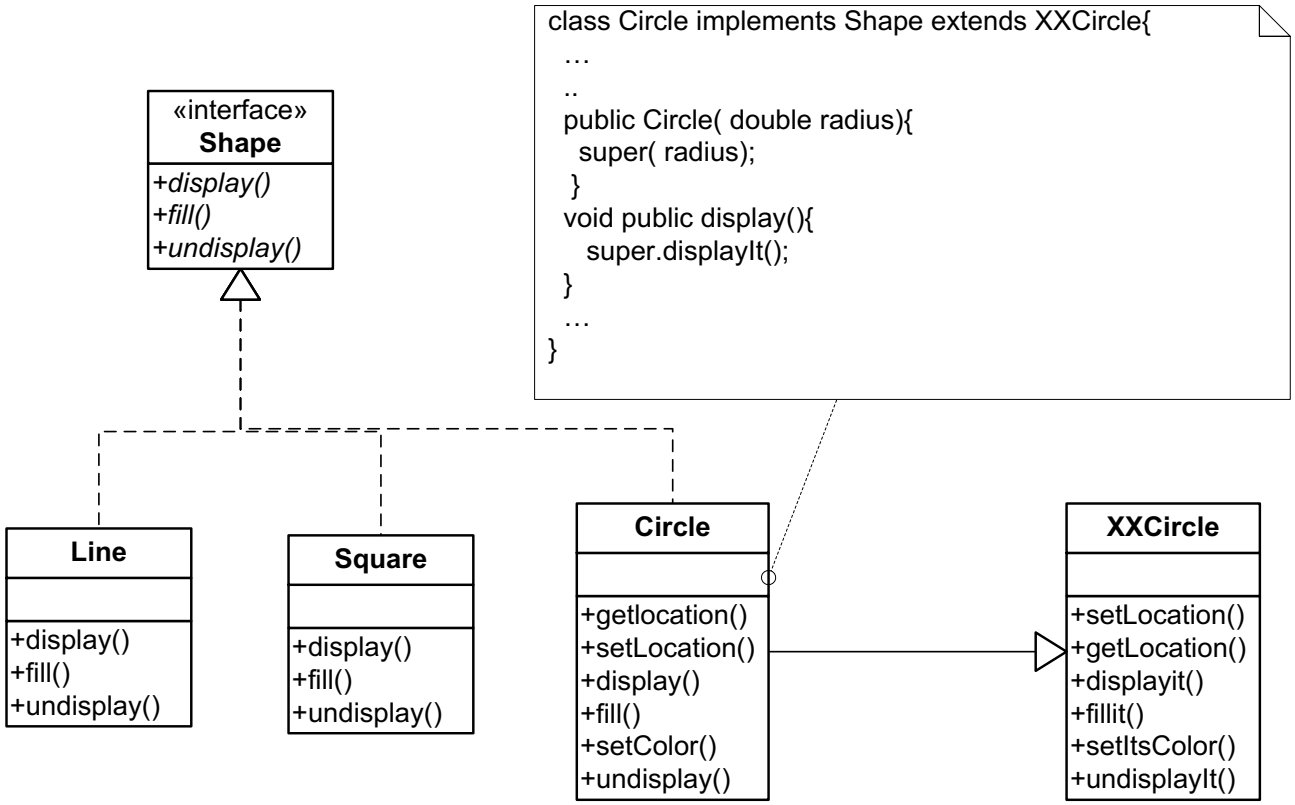


Adapter Example: Identify Participants



- **Target (`Shape`)**
  - defines the domain-specific interface that Client uses.
- **Adapter (`Circle`)**
  - adapts the interface **`Adaptee`** to the **`Target`** interface.
- **Adaptee (`XXCircle`)**
  - defines an existing interface that needs adapting.
- **Client (`ShapeApp`)**
  - collaborates with objects conforming to the Target interface.

Different Implementations of Adapter



Object Adapter and Class Adapter

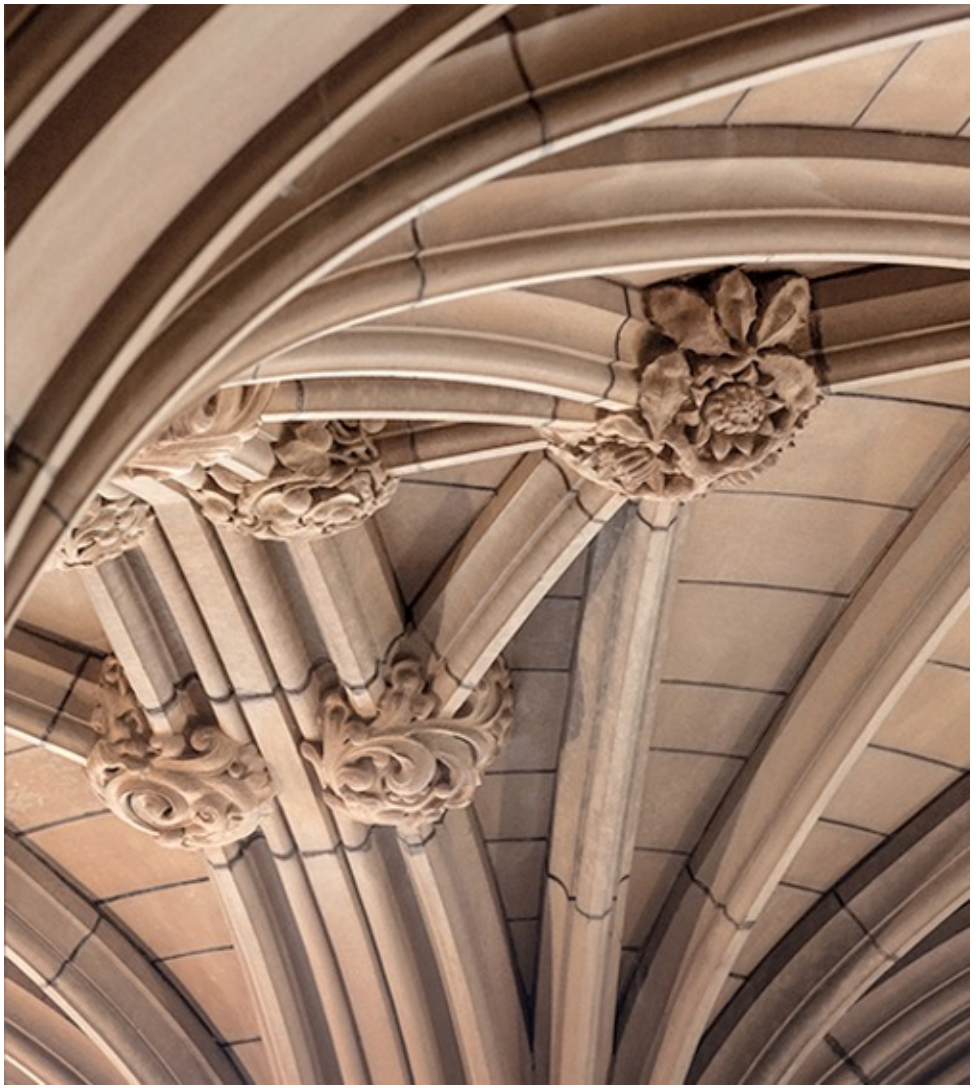
- Object Adapter
  - Relies on object composition to achieve adapter
- Class Adapter
  - Relies on class inheritance to achieve adapter

Two Reuse Mechanisms

- Inheritance and Delegation
  - Inheritance: reuse by subclassing;
    - "is-a" relationship (**white-box reuse)**
  - Delegation: reuse by composition;
    - "has-a" relationship (**black-box reuse)**
    - A class is said to delegate another class if it implements an operation by resending a message to another class
- Rule of thumb – design principles #1
  - **Favour object composition over class inheritance**

Observer Pattern

Object Behavioural



THE UNIVERSITY OF SYDNEY

Motivated Scenario

- Anytime the SOFT2201/COMP9201 unit coordinator Xi sent an announcement on Ed, all students could be notified by receiving an email.

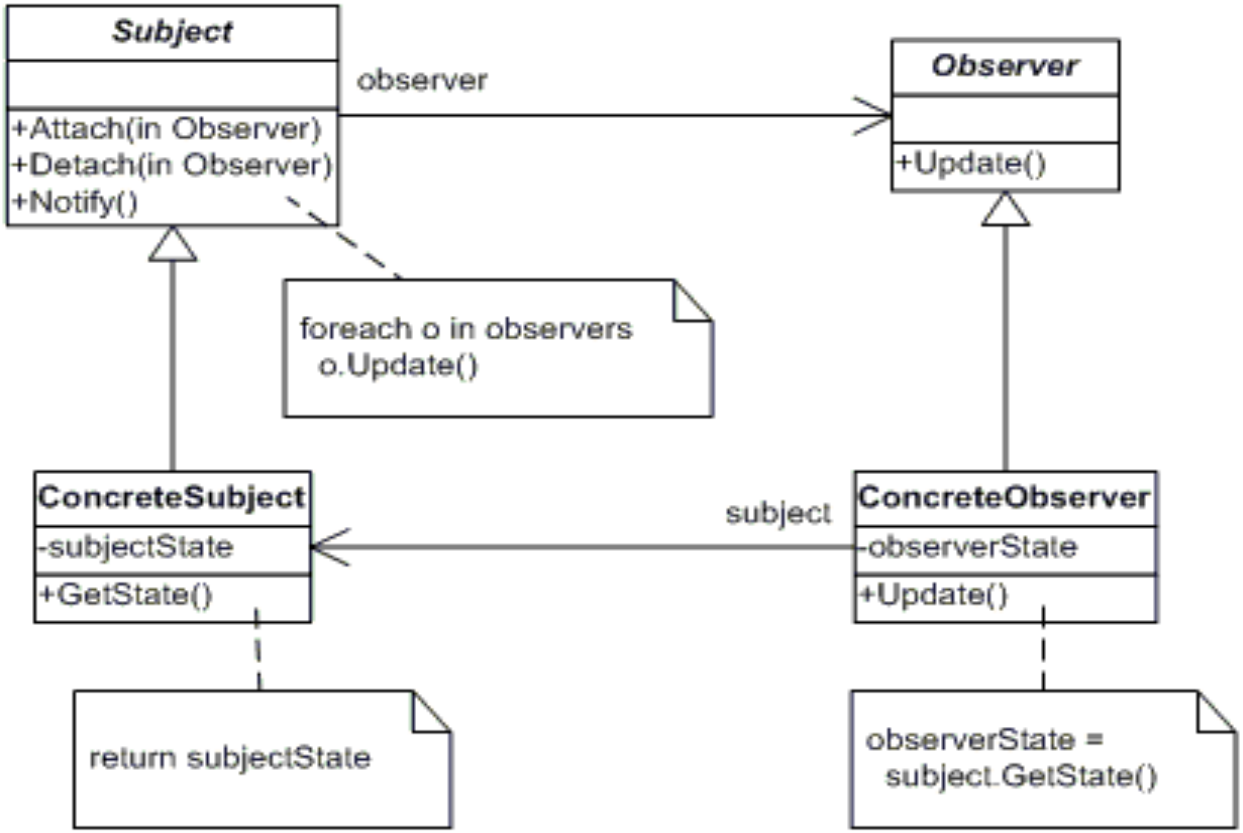
A1 marks has been released

Observer

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - A collection of cooperating classes (consistency between related objects)

- **Known as**
  - Dependents, Publish-Subscribe

Observer – Structure



Observer – Participants

Participant	Goals
Subject	Knows its observers. Any number of observer objects may observe a subject. Provides an interface for attaching and detaching observer objects
Observer	Defines an updating interface for objects that should be notified of changes in a subject
ConcreteSubject	Stores state of interest to ConcreteObserver objects Sends notifications to its observers when its state changes
ConcreteObserver	Maintains a reference to a ConcreteSubject object Stores state that should stay consistent with the subject's. Implements the observer's updating interface to keep its state consistent

Revisit the Motivated Example

Subject Perspective

```
public abstract class Subject {
    3 usages
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    3 usages
    public void Attach(Observer observer){
        observers.add(observer);
    }

    public void Detach(Observer observer){
        observers.remove(observer);
    }

    1 usage
    public void Notify() {
        for (Observer o: observers) {
            o.Update();
        }
    }
}
```

```
public class concreteSubject extends Subject{
    2 usages
    private String subjectState;

    1 usage
    public String getSubjectState() {
        return subjectState;
    }

    1 usage
    public void setSubjectState (String newState) {
        subjectState = newState;
    }
}
```

Revisit the Motivated Example

Observer Perspective

```
public interface Observer {
    1 usage   1 implementation
    public void Update();
}
```

Client Perspective

```
concreteSubject s = new concreteSubject();
s.Attach(new concreteObserver(s, name: "Tim"));
s.Attach(new concreteObserver(s, name: "Daniel"));
s.Attach(new concreteObserver(s, name: "Abbey"));

s.setSubjectState("A1 mark has been released");
s.Notify();
```

```
public class concreteObserver implements Observer{
    2 usages
    private String name;
    2 usages
    private String observerState;
    2 usages
    private concreteSubject subject;

    3 usages
    public concreteObserver(concreteSubject subject, String name){
        this.name = name;
        this.subject = subject;
    }

    1 usage
    @Override
    public void Update() {
        observerState = subject.getSubjectState();
        System.out.println("The latest announcement for " + name + " is: " + observerState);
    }
}
```

```
The latest announcement for Tim is: A1 mark has been released
The latest announcement for Daniel is: A1 mark has been released
The latest announcement for Abbey is: A1 mark has been released
```

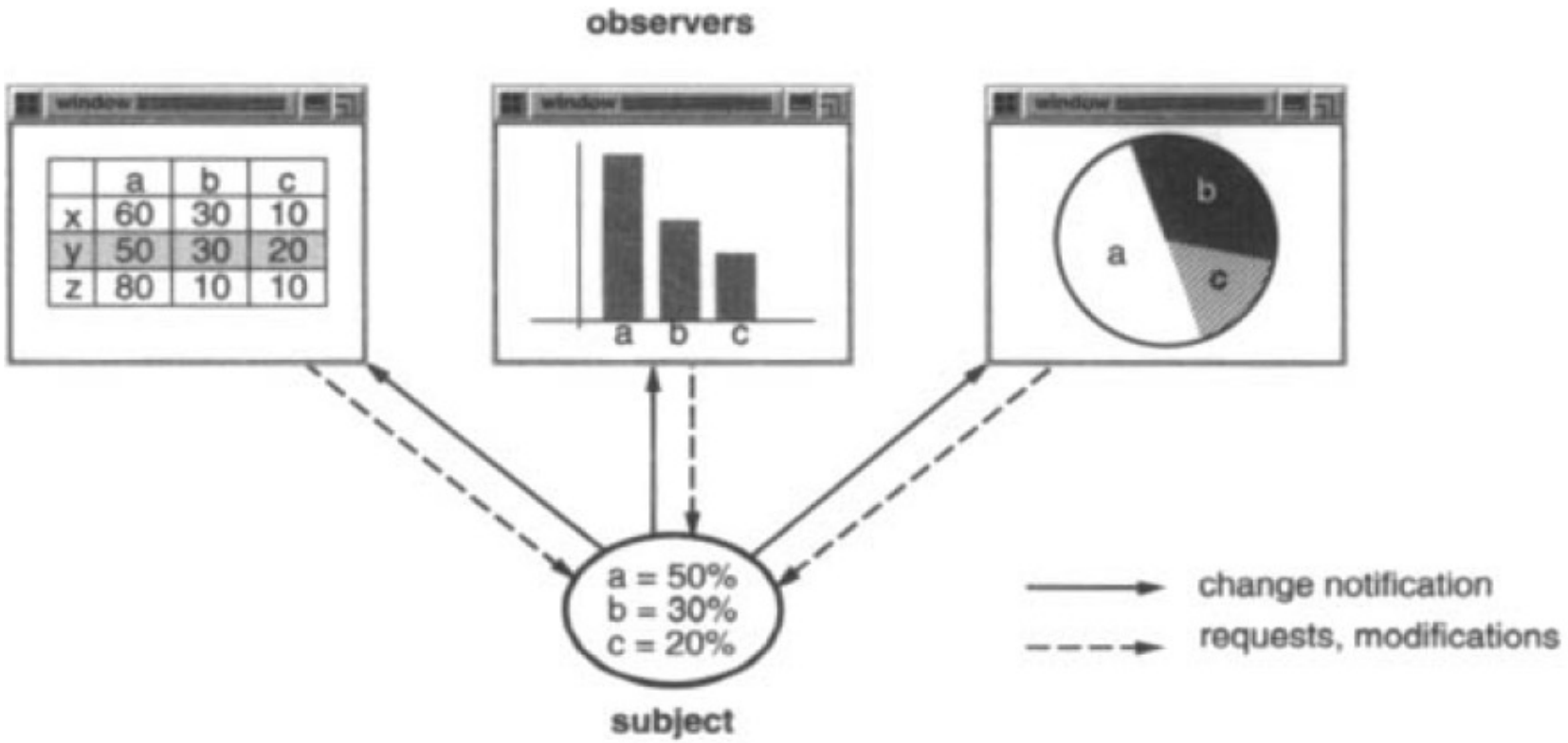
Observer – Applicability

- An abstraction has two aspects, one dependent on the other
- A change to one object requires changing others, and it's not clear how many objects need to be changed
- An object should be able to notify other objects without making assumptions about who these objects are

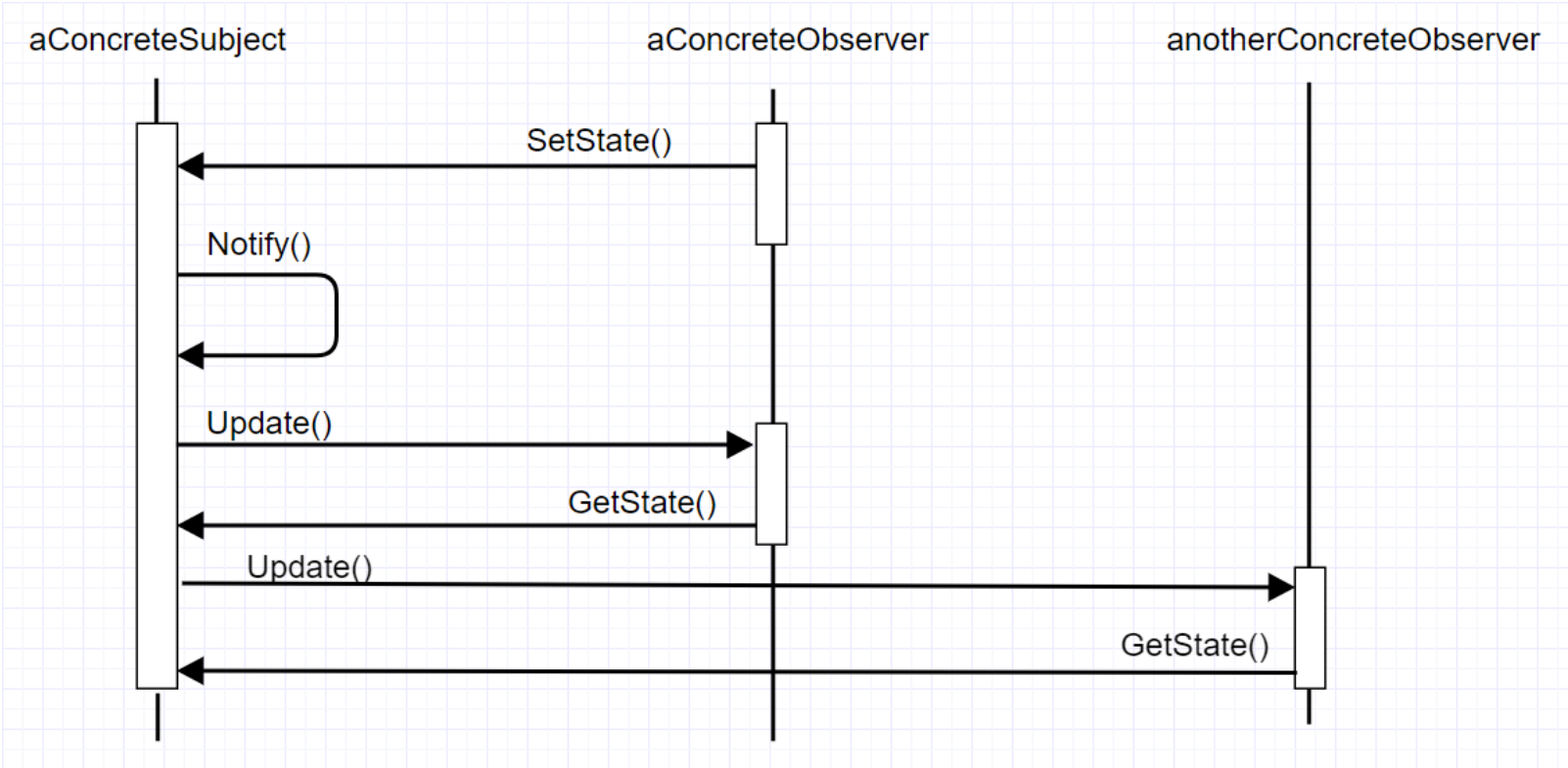
Observer – Publish-Subscribe

- Problem
  - You need to notify a varying list of objects that an event has occurred
- Solution
  - Subscriber/listener interface
  - Publisher: dynamically register interested subscribers and notify them when an event occurs

Observer – Example (Data Representation)



Observer – Collaborations



Observer – Consequences

- Abstract coupling between Subject and Observer
  - *Subject* only knows its *Observers* through the abstract *Observer* class (it doesn't know the concrete class of any observer)
- Support for broadcast communication
  - Notifications are broadcast automatically to all interested objects that subscribe to the *Subject*
  - Add/remove Observers anytime
- Unexpected updates
  - Observers have no knowledge of each other's presence, so they can be blind to the cost of changing the subject
  - An innocent operation on the subject may cause a cascade of updates to Observers and their dependents

Observer In GUI Class Library

- Observer pattern is the basic structure of event handling system in Java's GUI library
- Each GUI widget is a publisher of GUI related events
  - ActionEvent, MouseClickedEvent, ListSelectionEvent,…
- Observer can be any other objects (GUI or non-GUI objects)

Observer in Java GUI

- Example
  - A simple GUI with a **`JTextField`** and a **`JList`** component
  - Each time the List is selected, the selected text is displayed in the JTextField.
  - **`JList`** supports **`ListSelectionListener`**
  - A subclass of **`JTextField`** should implements **`ListSelectionListener`**

Observer in Java GUI

```
1  import javax.swing.*;
2  import javax.swing.event.*;
3  import java.awt.*;
4
5  class MyTextField extends JTextField implements ListSelectionListener{
6      public void valueChanged(ListSelectionEvent lse){
7          JList myList = (JList)lse.getSource();
8          setText((String)(myList.getSelectedValue()));
9      }
10 }
11
```

Method Summary

void	**valueChanged**(ListSelectionEvent e) Called whenever the value of the selection changes.

QuickEntryApplication

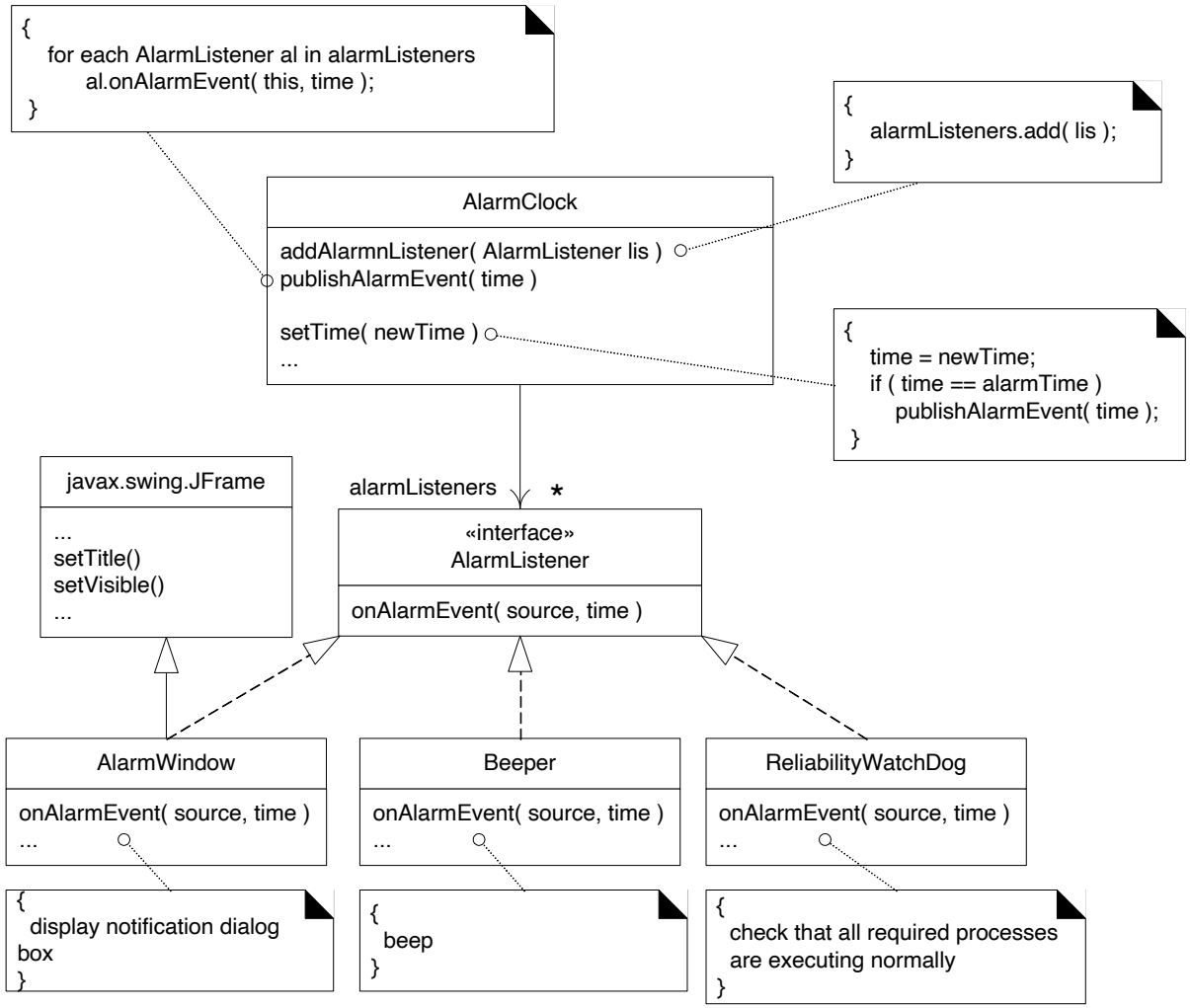
```
1  public class QuickEntryApplication{
2      public static void main(String argv[]){
3          MyTextField myTextField = new MyTextField();
4          String[]listItems = {"NSW","VIC","SW"};
5          JList myList = new JList(listItems);
6          myList.addListSelectionListener(myTextField); //add listener
7          JFrame myFrame = new JFrame();
8          Container contentpane = myFrame.getContentPane();
9          contentpane.setLayout(new GridLayout(0,1));
10         contentpane.add(new Label("State:"));
11         contentpane.add(myTextField);
12         contentpane.add(myList);
13         myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14         myFrame.pack();
15         myFrame.show();
16     }
17 }
18
```

Publisher-Many-Subscribers

- One publisher instance could have zero to many registered subscribers
  - E.g., one instance of an *AlarmClock* could have three registered alarm windows, four *Beepers*, and on *ReliabilityWatchDog*
  - When an alarm even happens, all eight of these *AlarmListeners* are notified via an *onAlarmEvent*

- Observer Implementation (Java)
  - Events are communicated via a regular message, such as on *onPropertyEvent*
  - Event is more formally defined as a class, and filled with appropriate event data
  - The event is then passed as a parameter in the event message

Publisher-Many-Subscribers



Task for Week 7

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize your time for assignment 2
- Prepare questions and ask during tutorials
- Self learning on Next Gen POS system

What are we going to learn in week 8?

- Code Review

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.