CSC301: Introduction to Software Engineering

Lecture 2

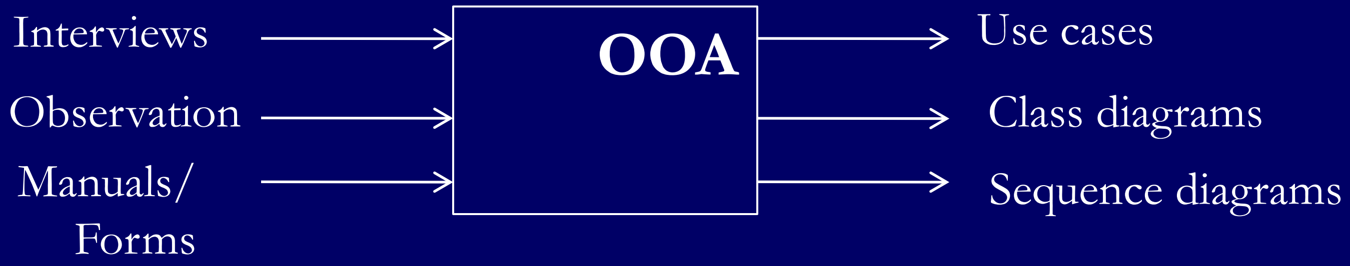
Wael Aboulsaadat

Object Oriented Design: design patterns

Object Oriented Analysis & Design

- OOA:
  - Input: written requirements statement, a formal vision document, interviews with stakeholders or other interested parties
  - Target: produce a conceptual model of the information that exists in the area being analyzed
  - Output: set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up

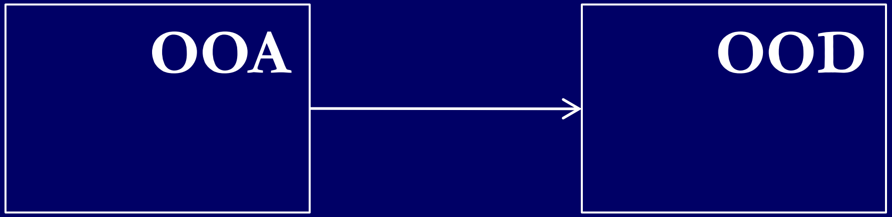
Object Oriented Analysis & Design



- Identify classes
- Identify relations between classes

Object Oriented Analysis & Design

- OOD:
  - Input: conceptual model produced in OOA
  - Target: a model of the solution domain, *how* the system is to be built, given the constraints identified in the OOA
  - Output: specification of implementation classes and interfaces



Techniques for Finding Objects

- OOA
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
- System Design
  - Subsystem decomposition
  - Try to identify layers and partitions
- Object Design
  - Find additional objects by applying implementation domain knowledge

Another Source for Finding Objects : Design Patterns

- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment
  - Then it describes the core of the solution to that problem, in such a way that you can use the this solution a million times over, without ever doing it the same twice

Patterns in Architecture

- Does this room makes you feel happy?
- Why?
  - Light (direction)
  - Proportions
  - Symmetry
  - Furniture
  - And more…

Design Patterns Types

- Creational Patterns
  - Focus: Creation of complex objects
  - Here we our goal is to provide a simple abstraction for a complex instantiation process.
  - We want to make the system independent from the way its objects are created, composed and represented.
  - Problems solved:
    - Hide how complex objects are created and put together

Creational
Factory Method Abstract Factory Builder Prototype Singleton

Design Patterns Types

- Structural Patterns
  - Focus: How objects are composed to form larger structures
  - They reduce the coupling between two or more classes
  - They introduce an abstract class to enable future extensions
  - They encapsulate complex structures
  - Problems solved:
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility

Structural
Adapter
Bridge
Composite
Decorator
Flyweight
Facade
Proxy

Design Patterns Types

- Behavioral Patterns
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Here we are concerned with algorithms and the assignment of responsibilies between objects: Who does what?
  - Behavioral patterns allow us to characterize complex control flows that are difficult to follow at runtime.
  - Problem solved:
    - Too tight coupling to a particular algorithm

Behavioural

Interpreter
Template Method
Chain of Responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor

Elements of a Design Pattern

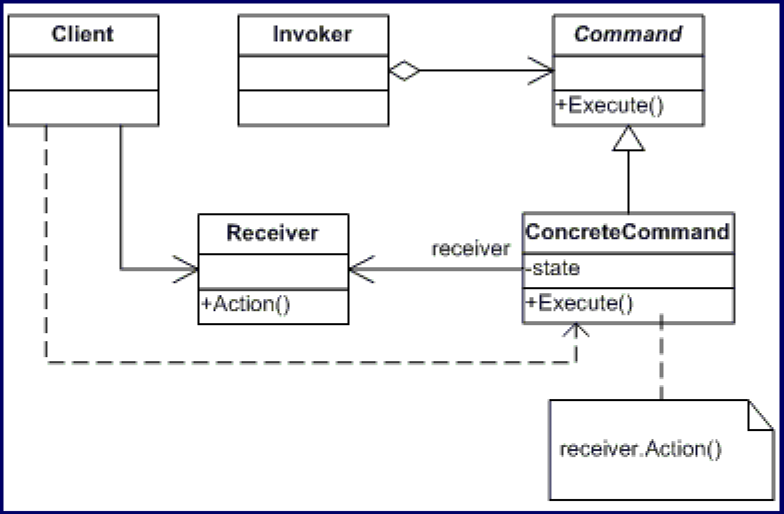
- Pattern Name
  - Increases design vocabulary, higher level of abstraction
- Problem
  - When to apply the pattern
  - Problem and context, conditions for applicability of pattern
- Solution
  - Relationships, responsibilities, and collaborations of design elements
  - Not any concrete design or implementation, rather a template
- Consequences
  - Results and trade-offs of applying the pattern
  - Space and time trade-offs, reusability, extensibility, portability

Pattern: Command

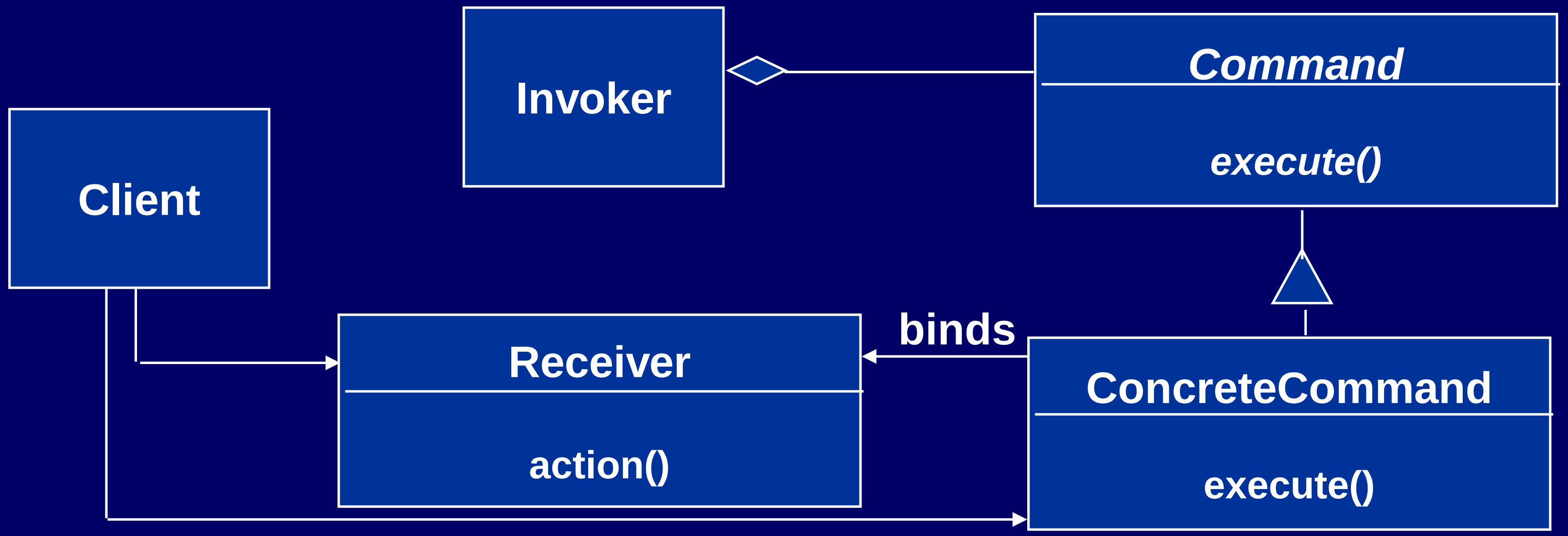
objects that represent actions…

Command pattern

- Command: an object that represents an action
  - sometimes called a "functor" to represent an object whose sole goal is to encapsulate one function



Command pattern



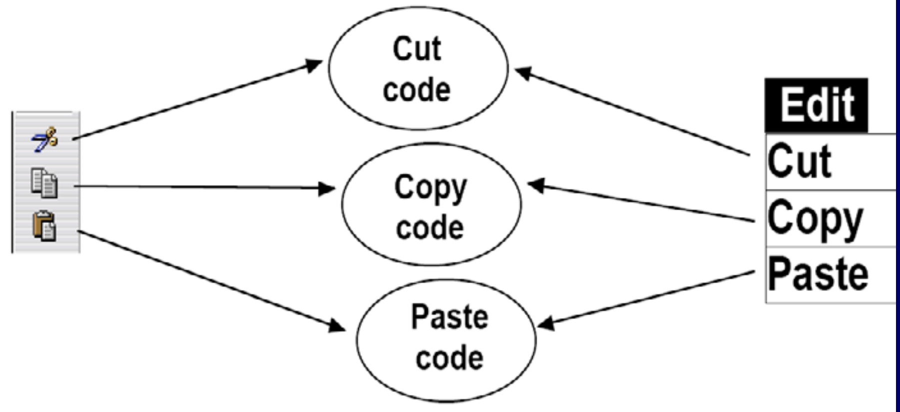
- Client creates a ConcreteCommand and binds it with a Receiver.
- Client hands the ConcreteCommand over to the Invoker which stores it.
- The Invoker has the responsibility to do the command (“execute” or “undo”).

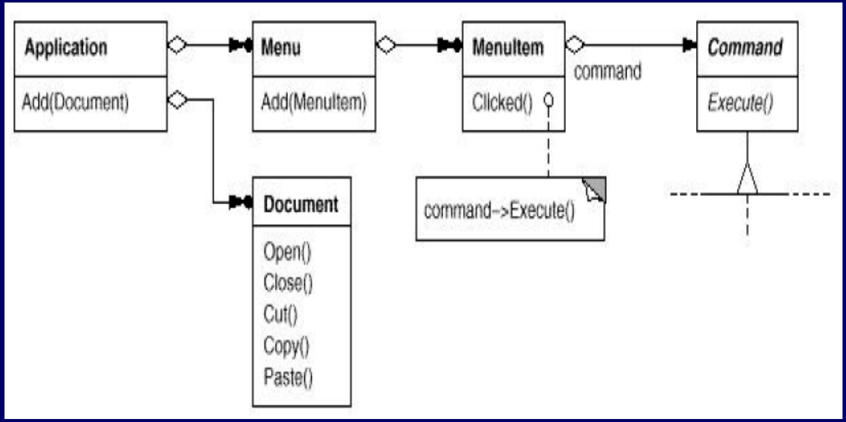
Command Pattern: motivation

- You want to build a user interface
- You want to provide menus
- You want to make the user interface reusable across many applications
  - You cannot hardcode the meanings of the menus for the various applications
  - The applications only know what has to be done when a menu is selected.
- Such a menu can easily be implemented with the Command Pattern

Common UI commands

- it is common in a GUI to have several ways to activate the same behavior
  - example: toolbar "Cut" button and "Edit / Cut" menu
  - this is *good* ; it makes the program flexible for the user
  - we'd like to make sure the code implementing these common commands is not duplicated

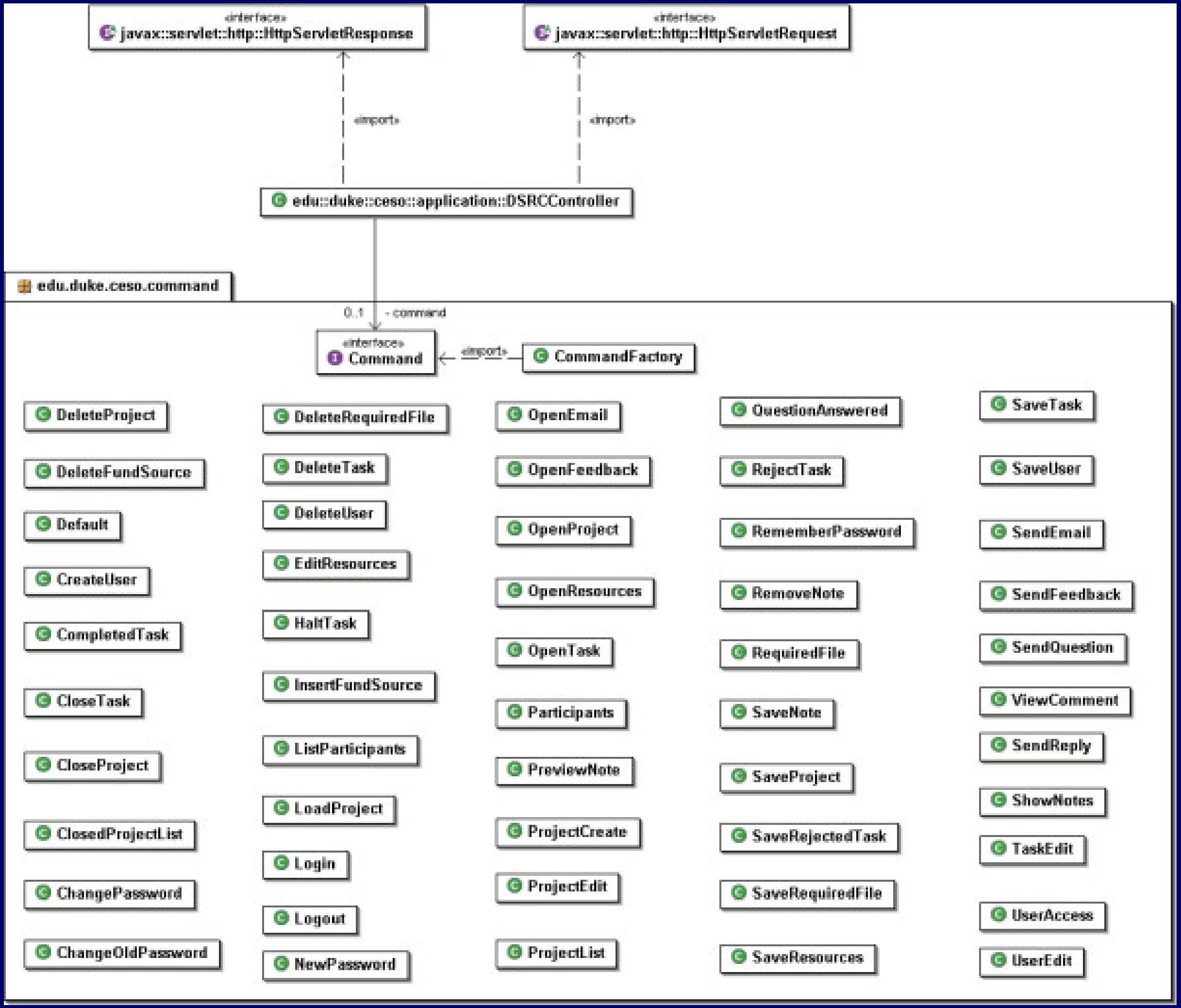




Command Pattern: second motivation

- A new way to think about designing the software

Command pattern - example



Command pattern Applicability

- “Encapsulate a request as an object, thereby letting you
  - parameterize clients with different requests,
  - queue or log requests, and
  - support undoable operations.”

- Uses:
  - Undo queues
  - Database transaction buffering

Pattern: Singleton

At max One Instance of a class!

Singleton Pattern

- Used to ensure that a class has only one instance. For example, one printer spooler object, one file system, one window manager, etc.

- Instead the class itself is made responsible for keeping track of its instance. It can thus ensure that no more than one instance is created. *This is the singleton pattern*.

Singleton example code

```
public class MySingletonClass {

 private static MySingletonClass instance
   = new MySingletonClass();

 public static MySingletonClass getInstance()
 {
      return instance;
 }

   /** There can be only one. */
 private MySingletonClass() {}

}
```

Pattern: Observer

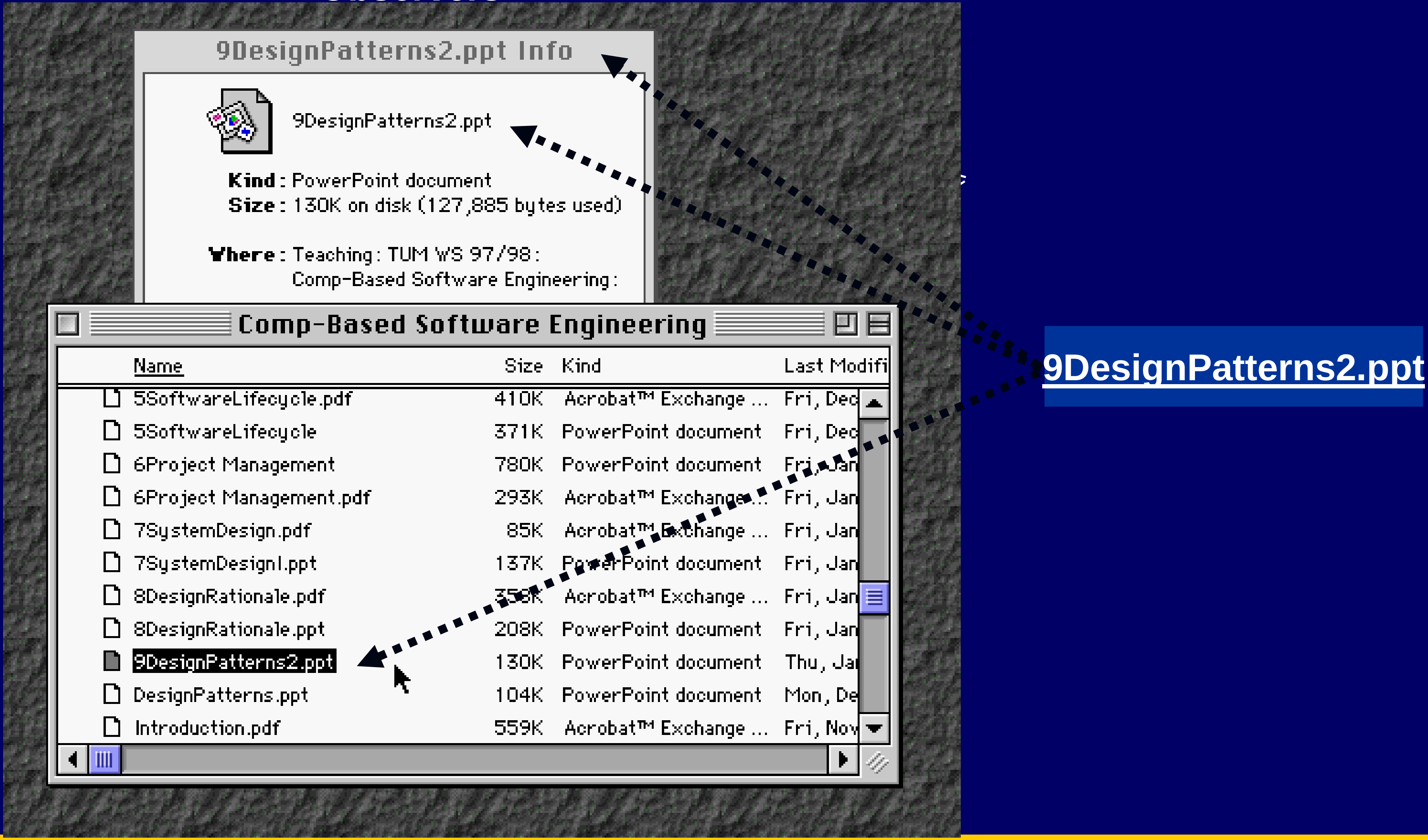
Observer pattern

- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”
- Also called “Publish and Subscribe”
- Uses:
  - Maintaining consistency across redundant state
  - Optimizing batch changes to maintain consistency

Observer pattern (continued)

Observers

Subject



9DesignPatterns2.ppt Info

9DesignPatterns2.ppt

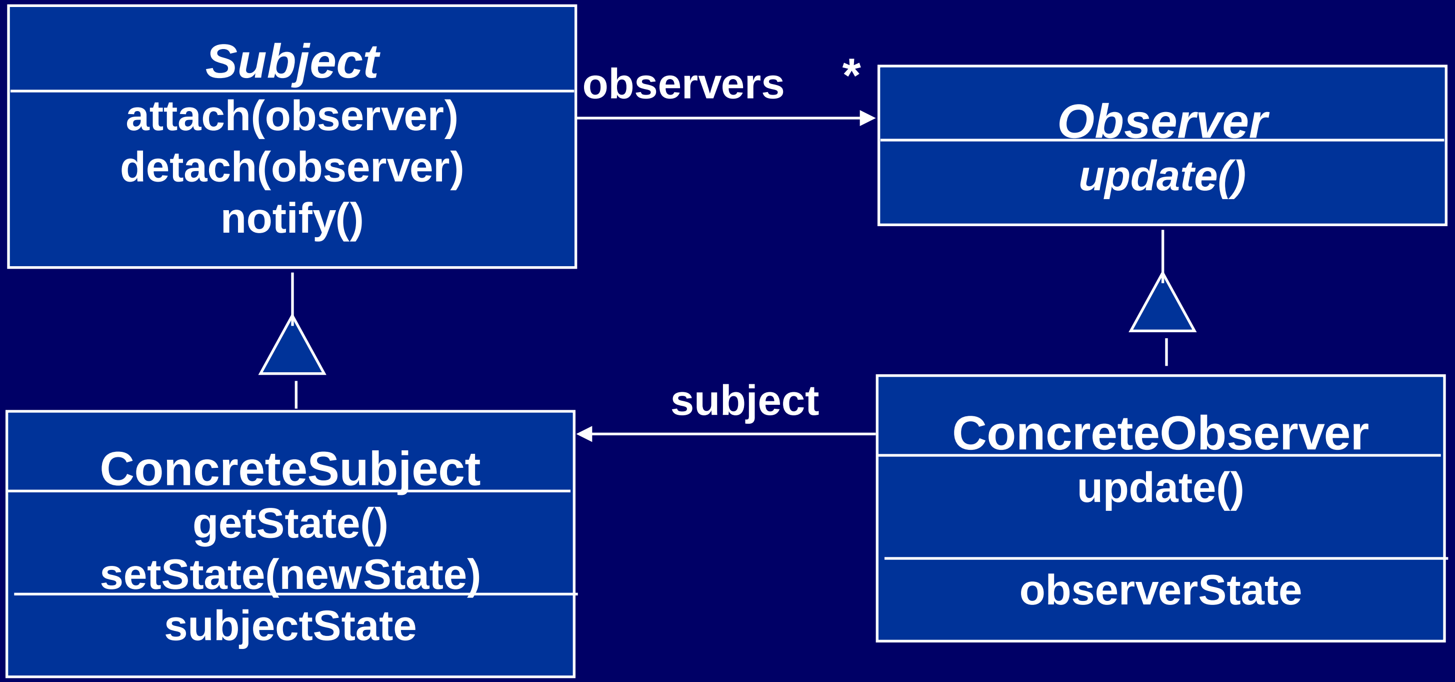
Kind : PowerPoint document
Size : 130K on disk (127,885 bytes used)

Where : Teaching : TUM WS 97/98 :
Comp-Based Software Engineering :

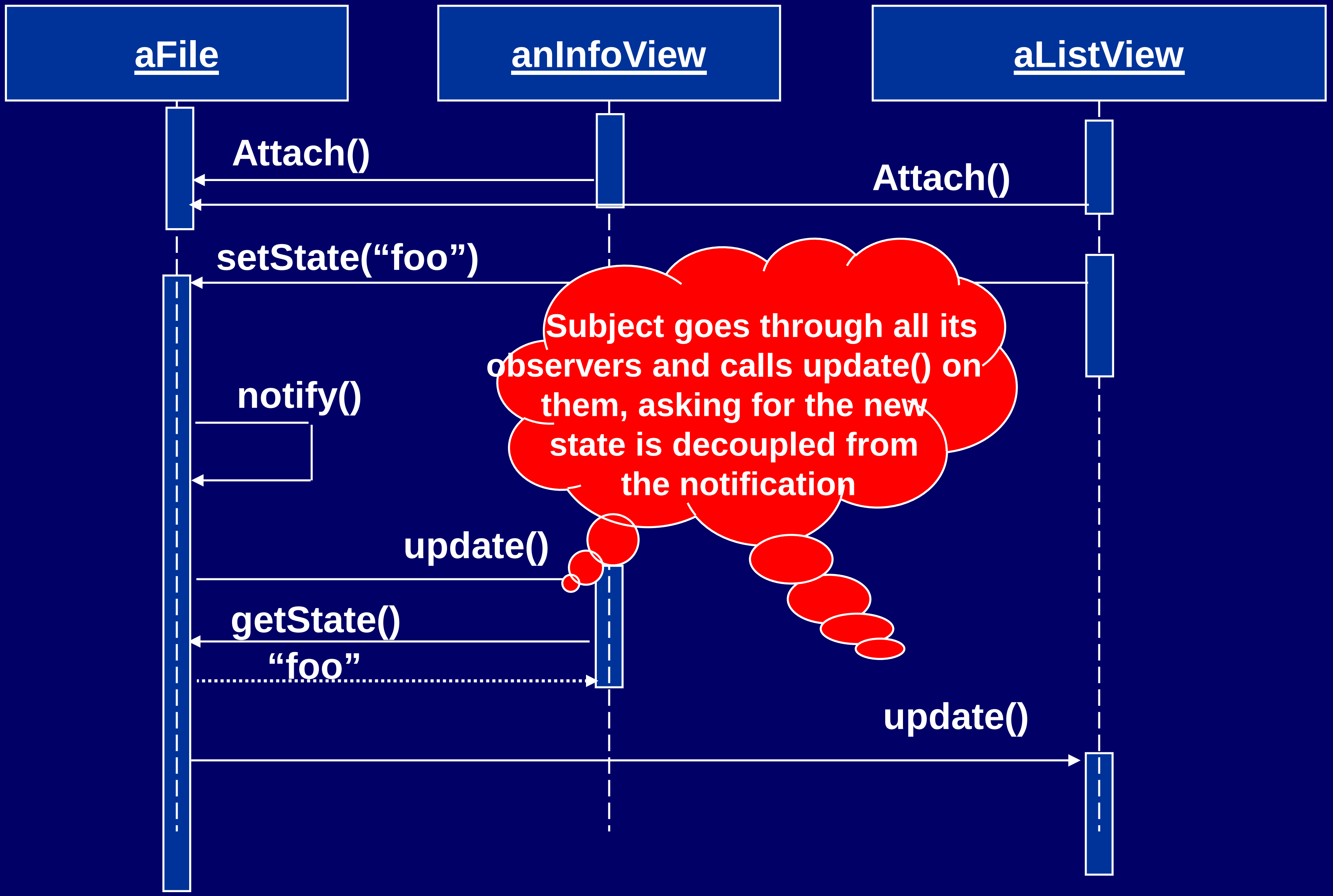
Comp-Based Software Engineering

Name	Size	Kind	Last Modifi
5SoftwareLifecycle.pdf	410K	Acrobat™ Exchange ...	Fri, Dec
5SoftwareLifecycle	371K	PowerPoint document	Fri, Dec
6Project Management	780K	PowerPoint document	Fri, Jan
6Project Management.pdf	293K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesign.pdf	85K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesignI.ppt	137K	PowerPoint document	Fri, Jan
8DesignRationale.pdf	358K	Acrobat™ Exchange ...	Fri, Jan
8DesignRationale.ppt	208K	PowerPoint document	Fri, Jan
9DesignPatterns2.ppt	130K	PowerPoint document	Thu, Ja
DesignPatterns.ppt	104K	PowerPoint document	Mon, De
Introduction.pdf	559K	Acrobat™ Exchange ...	Fri, Nov

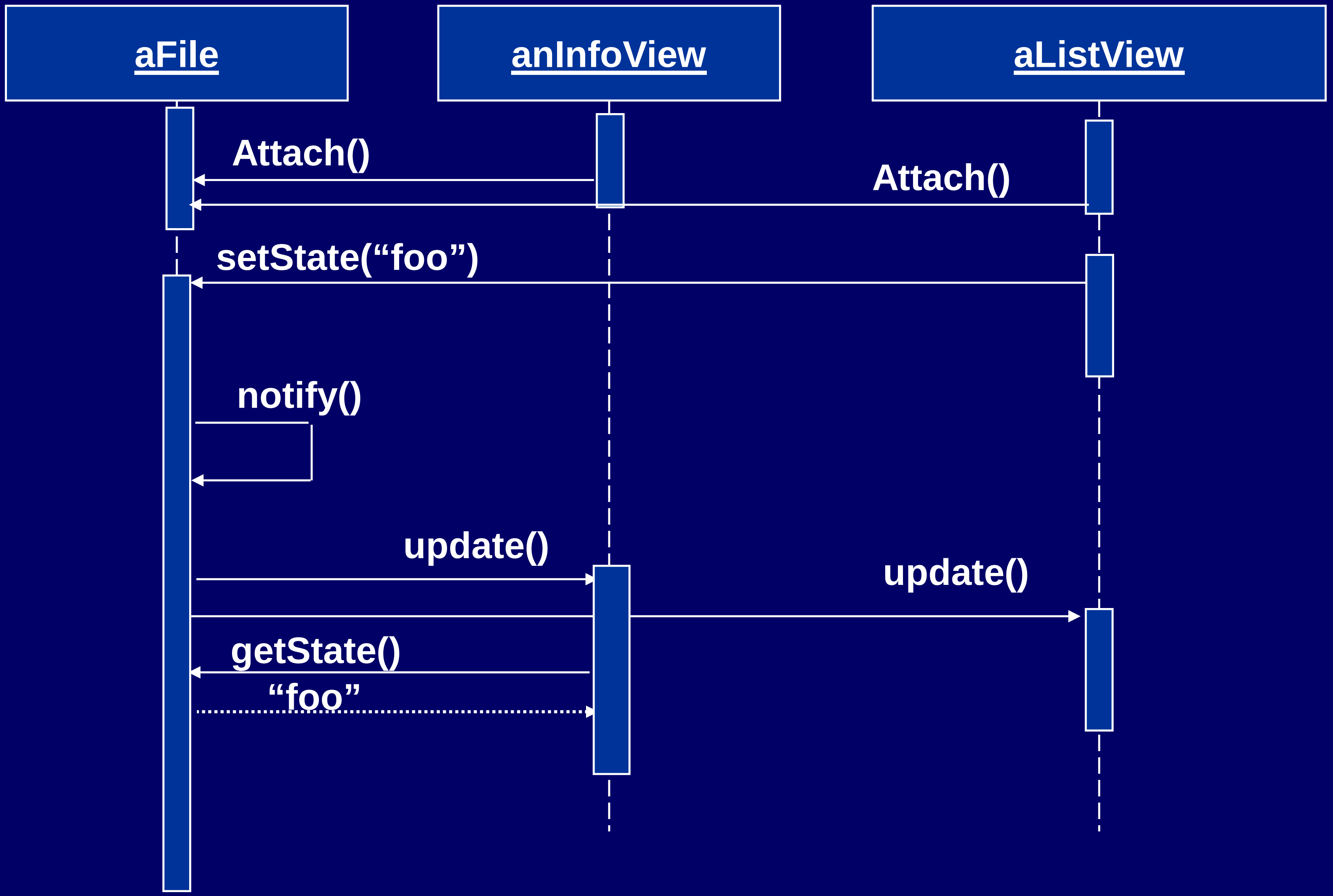
Observer pattern (cont'd)



- The Subject represents the actual state, the Observers represent different views of the state.
- Observer can be implemented as a Java interface.
- Subject is a super class (needs to store the observers vector) *not* an interface.



Subject goes through all its observers and calls update() on them, asking for the new state is decoupled from the notification


aFile
anInfoView
aListView
Attach()
Attach()
setState("foo")
notify()
update()
update()
getState()
"foo"

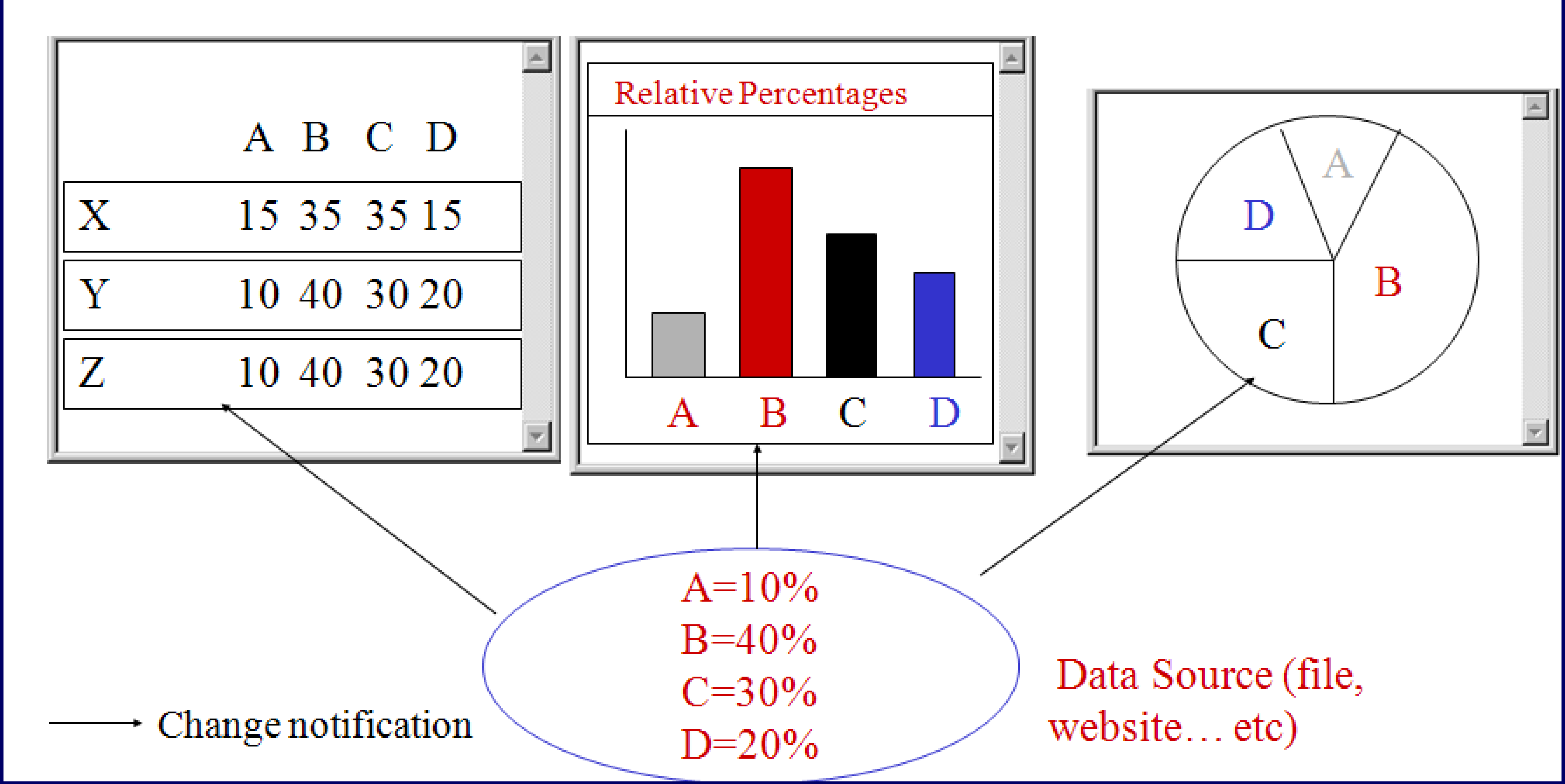
Observer pattern implementation in Java

```
// import java.util;

public class Observable extends Object {
      public void addObserver(Observer o);
      public void deleteObserver(Observer o);
      public boolean hasChanged();
      public void notifyObservers();
      public void notifyObservers(Object arg);
}

public abstract interface Observer {
      public abstract void update(Observable o, Object arg);
}
public class Subject extends Observable{
      public void setState(String filename);
  public string getState();
}
```

Observer Pattern



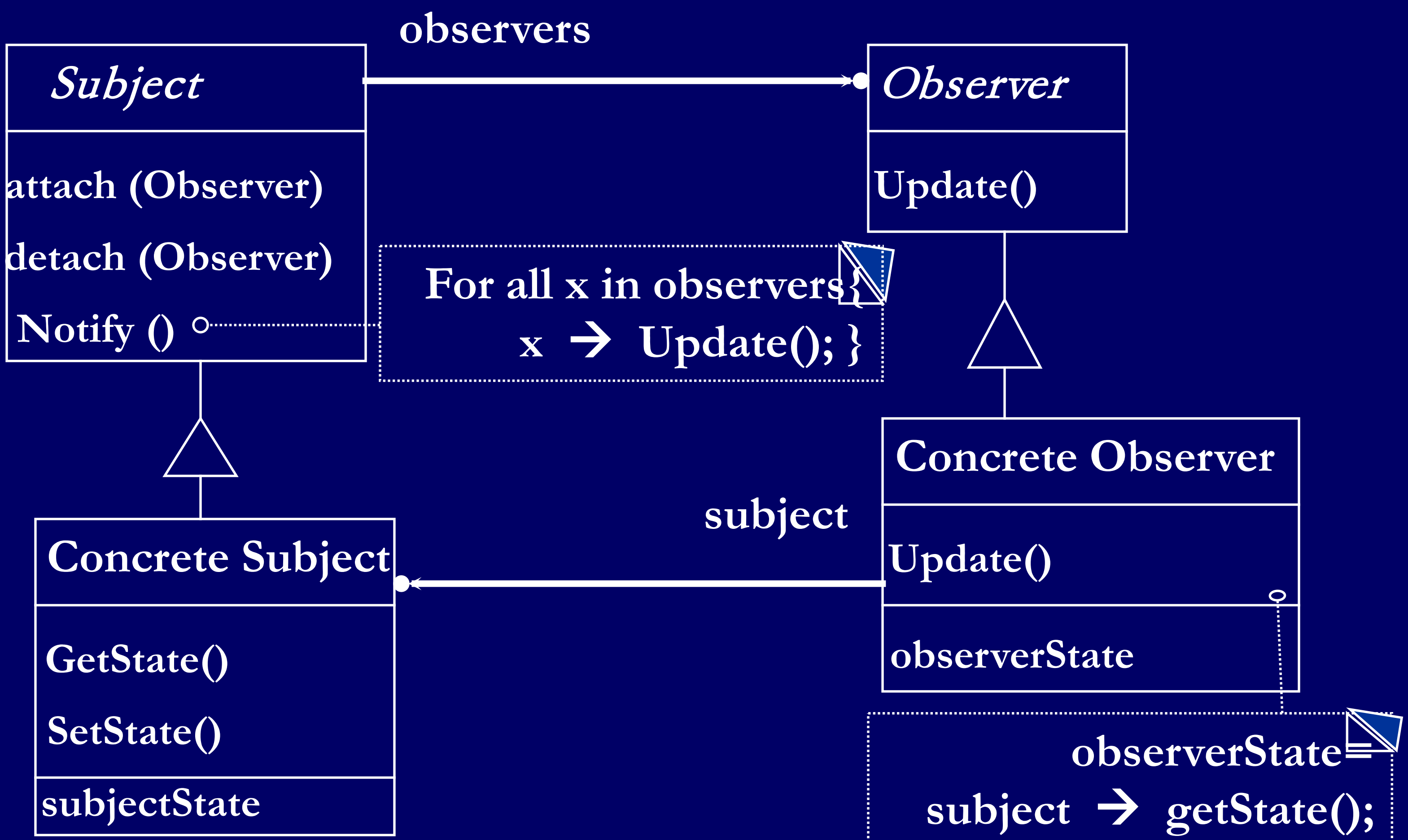
	A	B	C	D
X	15	35	35	15
Y	10	40	30	20
Z	10	40	30	20

A=10%
B=40%
C=30%
D=20%

Data Source (file, website… etc)

→ Change notification

Observer Pattern



Observer Pattern

- Need to separate presentational aspects with the data, i.e. separate views and data.
- **Classes defining application data and presentation can be reused.**
- **Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.**
- **Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.**

Observer Pattern

- GUI programming example

Pattern: Template Method

Pizza Machine Program: what's Wrong With This?

```
public class PizzaMaker {
  public void cookPizzas(List pizzas) {
    for (int i=0; i<pizzas.size(); ++i) {
      Object pizza = pizzas.get(i);
      if (pizza instanceof ThinCrustPizza) {
        ((ThinCrustPizza)pizza).cookInWoodFireOven();
      }
      else if (pizza instanceof PanPizza) {
        ((PanPizza)pizza).cookInGreasyPan();
      }
      else {

      }
    }
  }
}
```

The Open-Closed Principle

- *Classes should be open for extension, but closed for modification*
  - .e., you should be able to extend a system *without* modifying the existing code
- The type-switch in the example violates this
  - Have to edit the code every time the marketing department comes up with a new kind of pizza

Abstraction is the Solution

- Solve the problem by creating a `Pizza` interface with a `cook` method
  - Or an abstract base class whose cook method must be overridden by every child
- Simple, right?

How Open Should You Be?

```
public abstract class Pizza {
  public final void cook() {
    placeOnCookingSurface();
    placeInCookingDevice();
    int cookTime = getCookTime();
    letItCook(cookTime);
    removeFromCookingDevice();
  }
  protected abstract void placeOnCookingSurface();
  protected abstract void placeInCookingDevice();
  protected abstract int  getCookTime();
  protected abstract void letItCook(int min);
  protected abstract void removeFromCookingDevice();
}
```

Template Method Design Pattern

- The *Template Method* design pattern is used to set up the skeleton of an algorithm
  - Details then filled in by concrete subclasses
- But what if someone wants to do something you didn’t anticipate?
  - E.g., wants to add a `PancakePizza` that has to be flipped over halfway through the cooking process

Override the Template Method?

```
public final void cook() {
  placeOnCookingSurface();
  placeInCookingDevice();
  int cookTime = getCookTime();
  letItCook(cookTime/2);
  flip();
  letItCook(cookTime/2);
  removeFromCookingDevice();
}
```

- But `cook` was `final`
- And it's storing up trouble for the future

Squeeze It Somewhere Else?

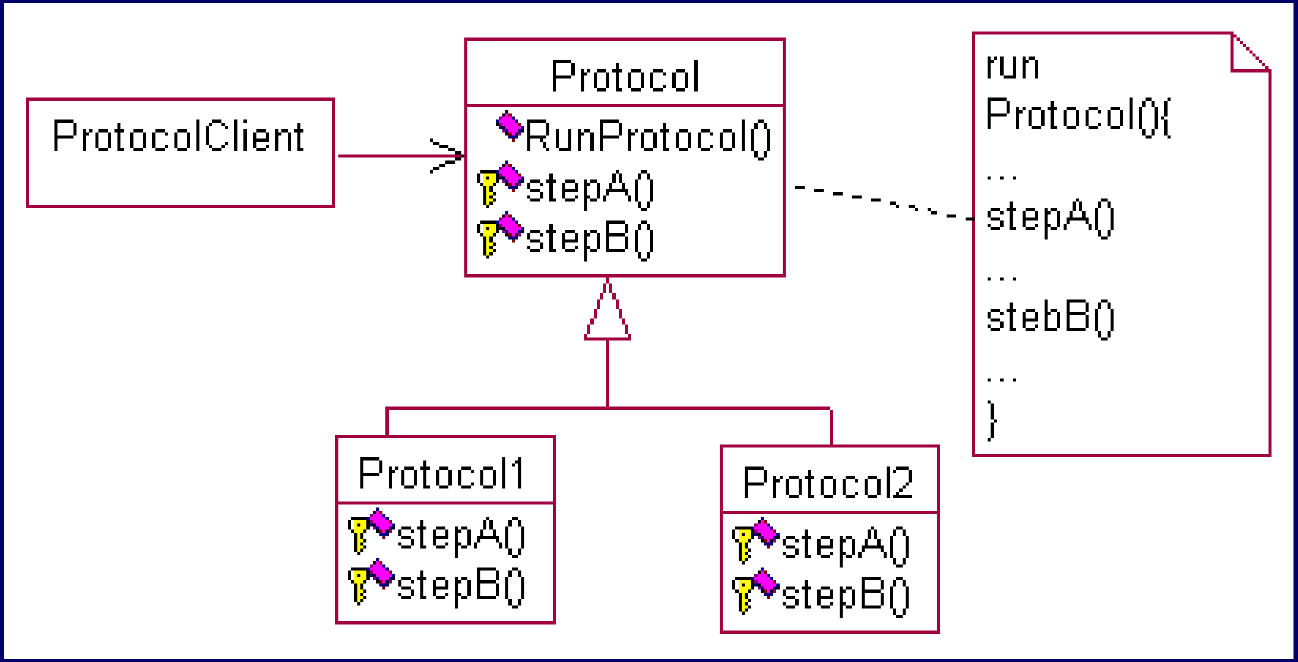
```
protected void removeFromCookingDevice() {
  flip();
  letItCook(cookTime);
  …remove from skillet…
}
```

- `removeFromCookingDevice` shouldn’t be doing other things
  - Think about the documentation
- And once again, we’re storing up trouble for the future

Leave Space for Future Growth?

```
public final void cook() {
  beforePlacingOnCookingSurface();
  placeOnCookingSurface();
  beforePlacingInCookingDevice();
  placeInCookingDevice();
  beforeCooking();
  for (int i=0; i<getCookingPhases(); i++) {
    letItCook(getCookTime(i));
    afterCookingPhase(i);
  }
  beforeRemovingFromCookingDevice();
  removeFromCookingDevice();
  afterRemovingFromCookingDevice();
}
```

Template Method Design Pattern



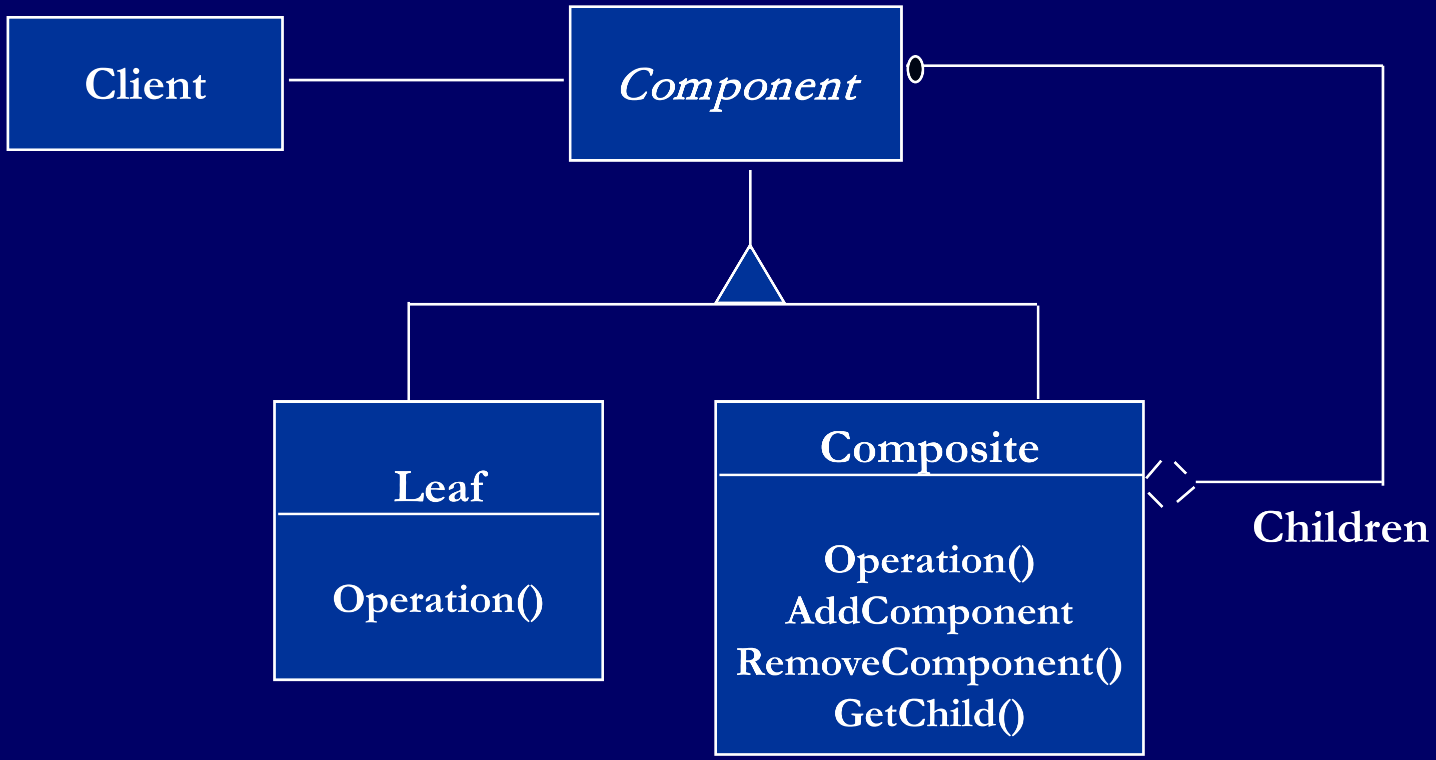
Pattern: composite

What is common between these definitions?

- Definition - Software System
  - A software system consists of subsystems which are either other subsystems or collection of classes
- Definition - Software Lifecycle:
  - The software lifecycle consists of a set of development activities which are either other activities or collection of tasks

Introducing the Composite Pattern

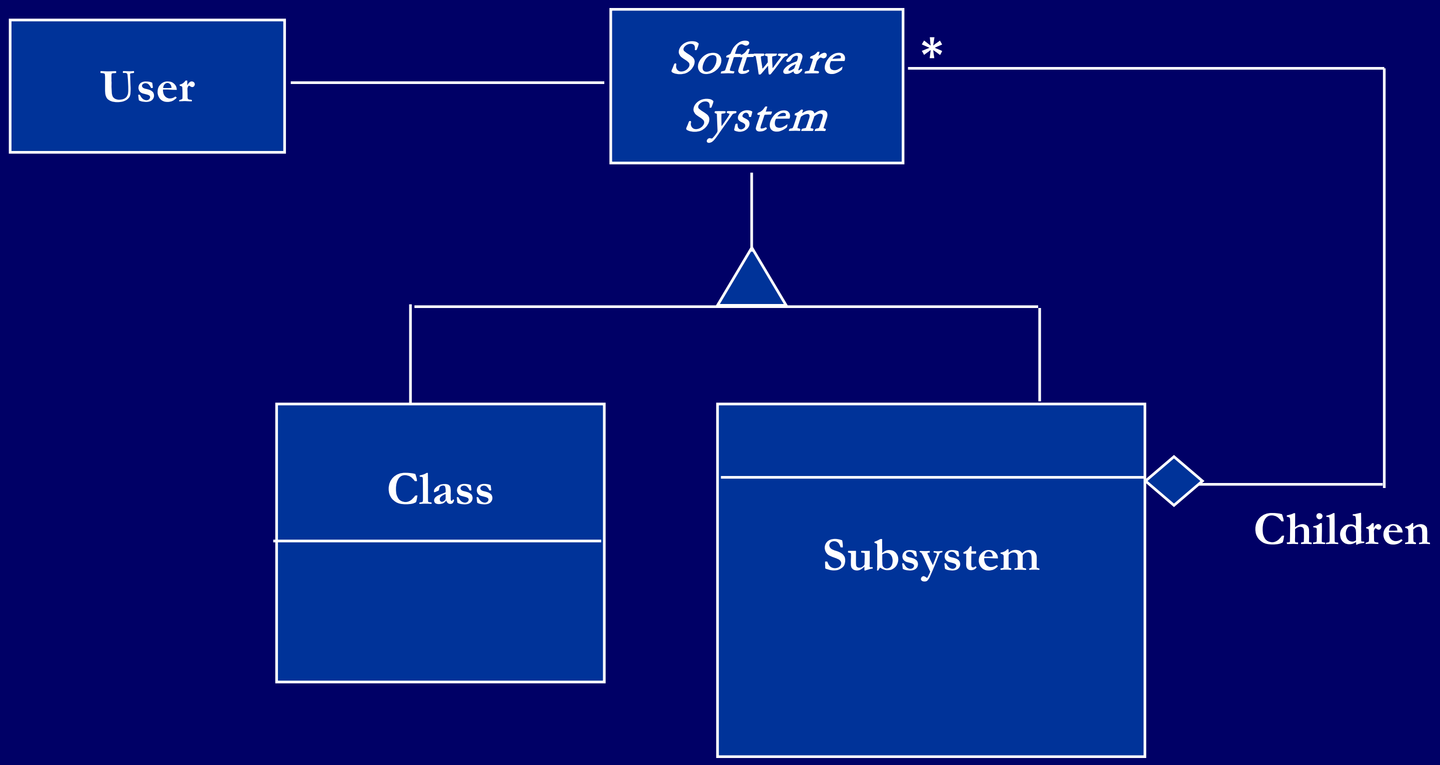
- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly



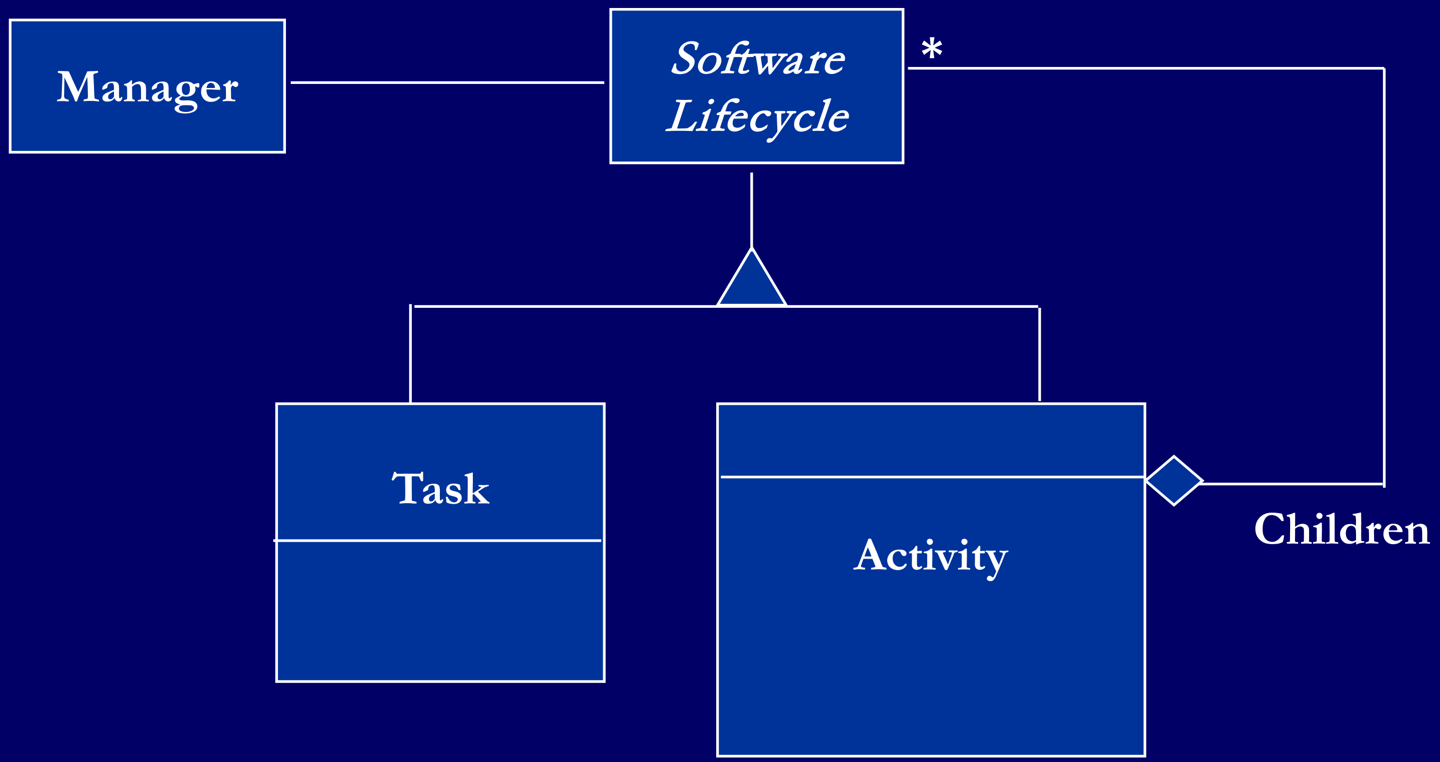
What is common between these definitions?

- Software System:
  - Definition: A software system consists of subsystems which are either other subsystems or collection of classes
  - Composite: Subsystem (A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...)
  - Leaf node: Class
- Software Lifecycle:
  - Definition: The software lifecycle consists of a set of development activities which are either other actitivies or collection of tasks
  - Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which....)
  - Leaf node: Task

Modeling a Software System with a Composite Pattern

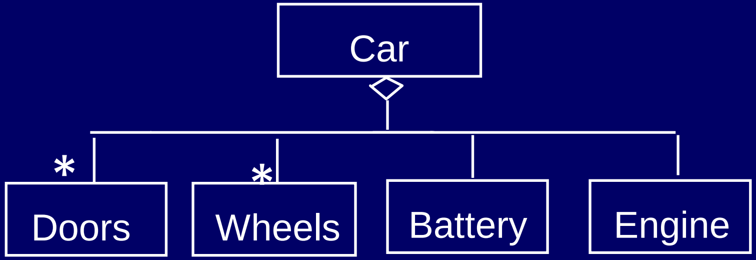


Modeling the Software Lifecycle with a Composite Pattern

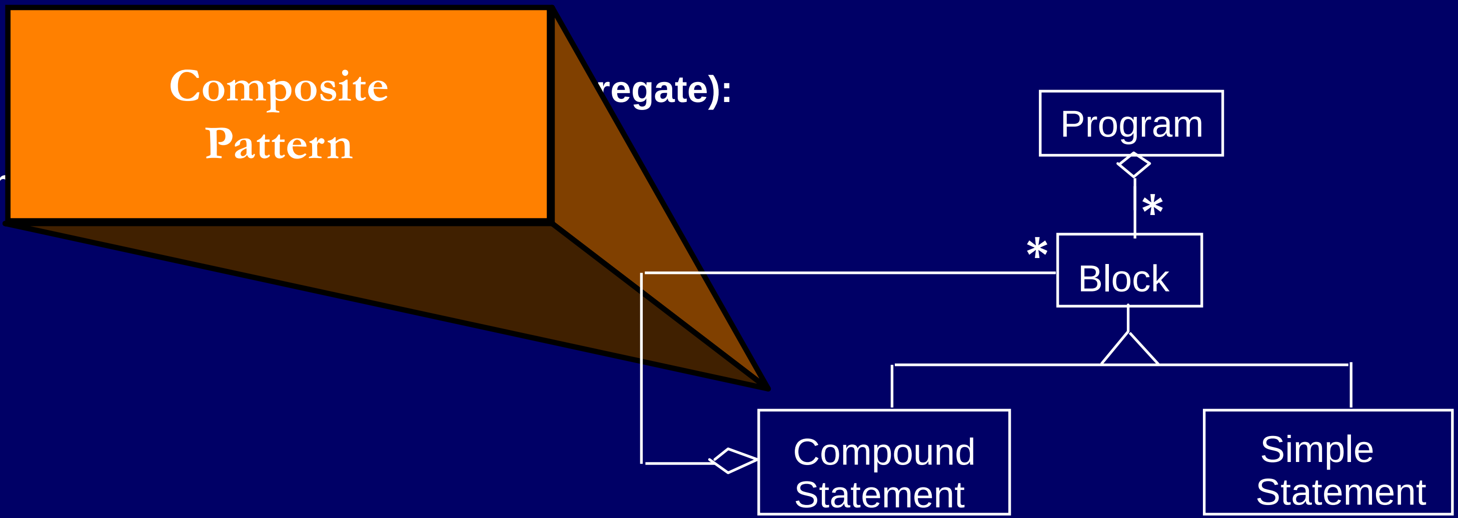


Composite Patterns models dynamic aggregates

Fixed Structure:

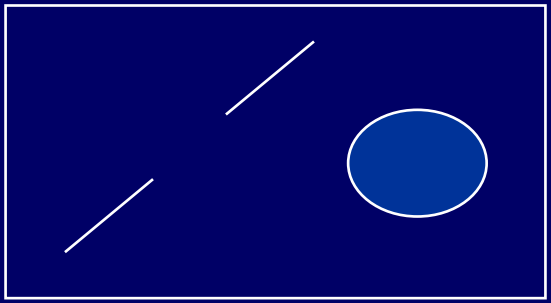


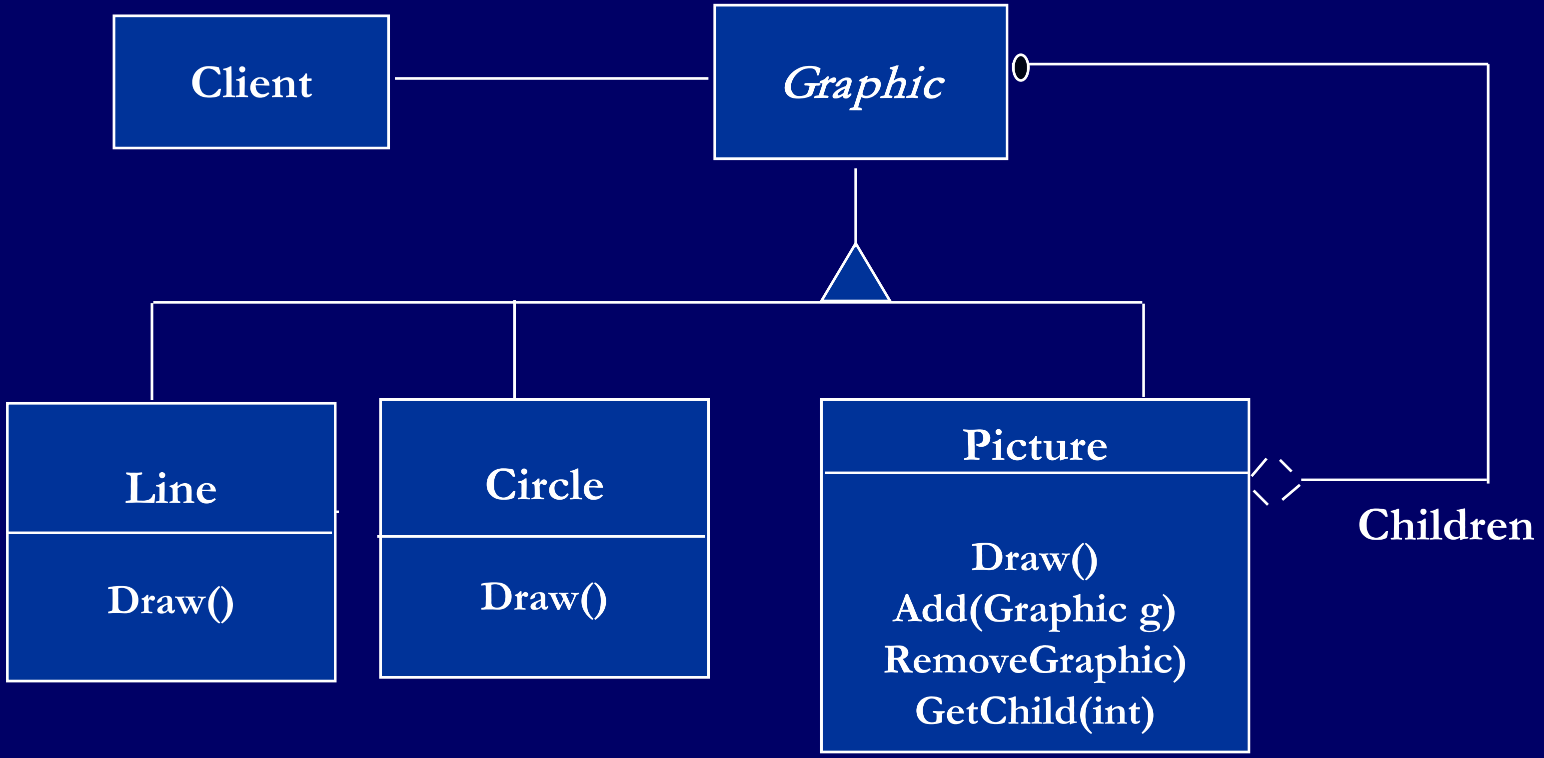
Organization Chart (variable aggregate):



Graphic Applications also use Composite Patterns

- The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)





Client

Graphic

Line

Draw()

Circle

Draw()

Picture

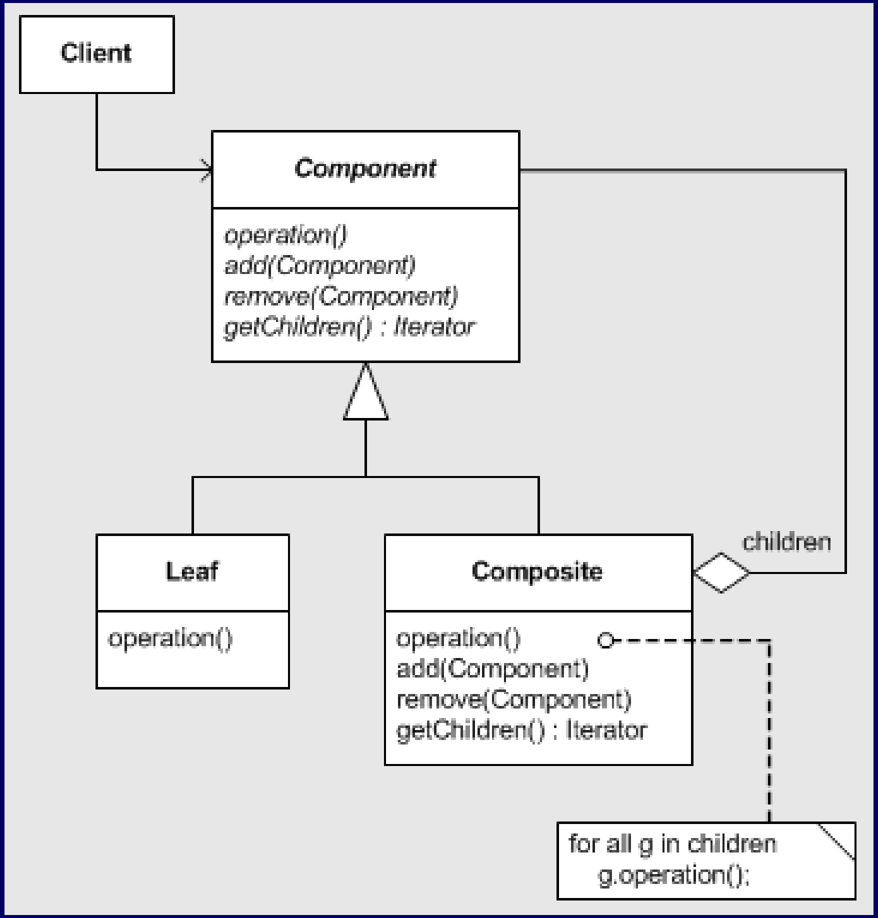
Draw()
Add(Graphic g)
RemoveGraphic)
GetChild(int)

Children

Composite Pattern

- Facilitates the composition of objects into tree structures that represent part-whole hierarchies.
- These hierarchies consist of both primitive and composite objects.

Composite Design Pattern



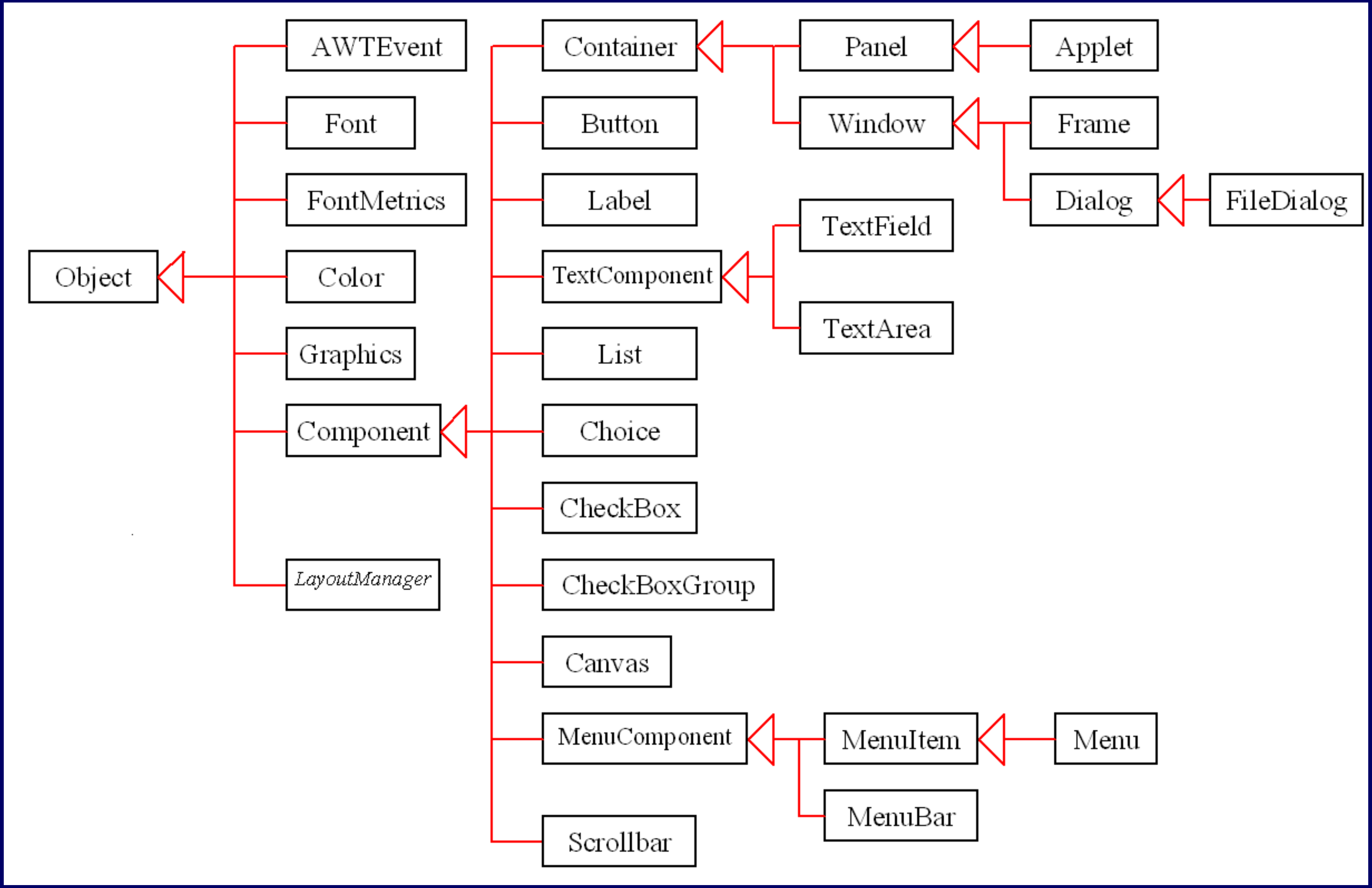
Composite Pattern – Participants

- Component
  - Declares interface for objects and for accessing children
  - Implements default behavior
- Leaf
  - No children; defines behavior for primitive objects
- Composite
  - Defines behavior for components with children
  - Stores children and implements children-related operations
- Client
  - Manipulates objects in the composition thru' Component interface.

Composite Pattern - Consequences

- Defines Class hierarchies for recursive composition.
- Makes clients simple (can treat composite structures and individual objects uniformly)
- Makes it easy to add new components (no code needed for components or for clients)
- Can make your design overly general – Harder to restrict the components of a composite.

Example: AWT Class Hierarchy



Composite example: layout

```
Container north = new JPanel(new FlowLayout());
north.add(new JButton("Button 1"));
north.add(new JButton("Button 2"));

Container south = new JPanel(new BorderLayout());
south.add(new JLabel("Southwest"), BorderLayout.WEST);
south.add(new JLabel("Southeast"), BorderLayout.EAST);

Container cp = getContentPane();
cp.add(north, BorderLayout.NORTH);
cp.add(new JButton("Center Button"), BorderLayout.CENTER);
cp.add(south, BorderLayout.SOUTH);
```