

Singleton Pattern

Singleton Pattern: Problem

- o **Ensure that a class has only one instance.**
- o **Provide a global point of access to it.**

Singleton Pattern: Example 1

- **Problem: An object needs to maintain an application's configuration.**
- Many objects may wish to read and update the configuration:
 - How to restrict only one instantiation?
 - A singleton class will ensure that all objects of the application can get the same copy of configuration...

Singleton Pattern: Example 2

- **Problem:**
 - How to control access to resources such as database connections or sockets?
- **Example:** You have a license for only one connection for your database:
 - A Singleton makes sure that only one connection is made.
 - If you buy more licenses or use a JDBC driver that allows multithreading, the Singleton can be easily adjusted to allow more connections.

Stateful Singleton: Example 3

- Suppose you need a **counter** that gives out unique numbers (e.g. token numbers in a restaurant):
 - First, the counter needs to be unique.
 - A Singleton can generate the numbers and synchronize access.

More Singleton Examples

- One session manager per user session
- One file system
- One shopping basket per customer
- One logger
- One configuration object
- One account per user,
- Abstract factory and factory method, etc.

Singleton Pattern

- A Singleton is accessed by many objects:
 - Therefore should be easily and globally accessible --- Provide a global point of access to the object.
 - Ensure that only one instance of a class is created --- Allow multiple instances when required.

Singleton Solution: Main Idea

- Create an object with operation:
–**getInstance()**
- On first call to **getInstance()**:
–Relevant object instance is created and object identity is returned.
- On subsequent calls to **getInstance()**:
–No new instance is created,
–Id of existing object is returned.

Singleton Pattern

- A Singleton class itself is responsible for keeping track of its sole instance.
 - **How?** --- intercept all requests to create new objects.
- Singletons maintain a static reference to the sole singleton instance:
 - Return a reference to that instance from a static instance() method.

Singleton Structure

Singleton

-uniqueInstance

Object identifier for singleton instance, class scope i.e. static

-singletonData

+getInstance()

Returns object identifier for unique instance

+getSingletonData()

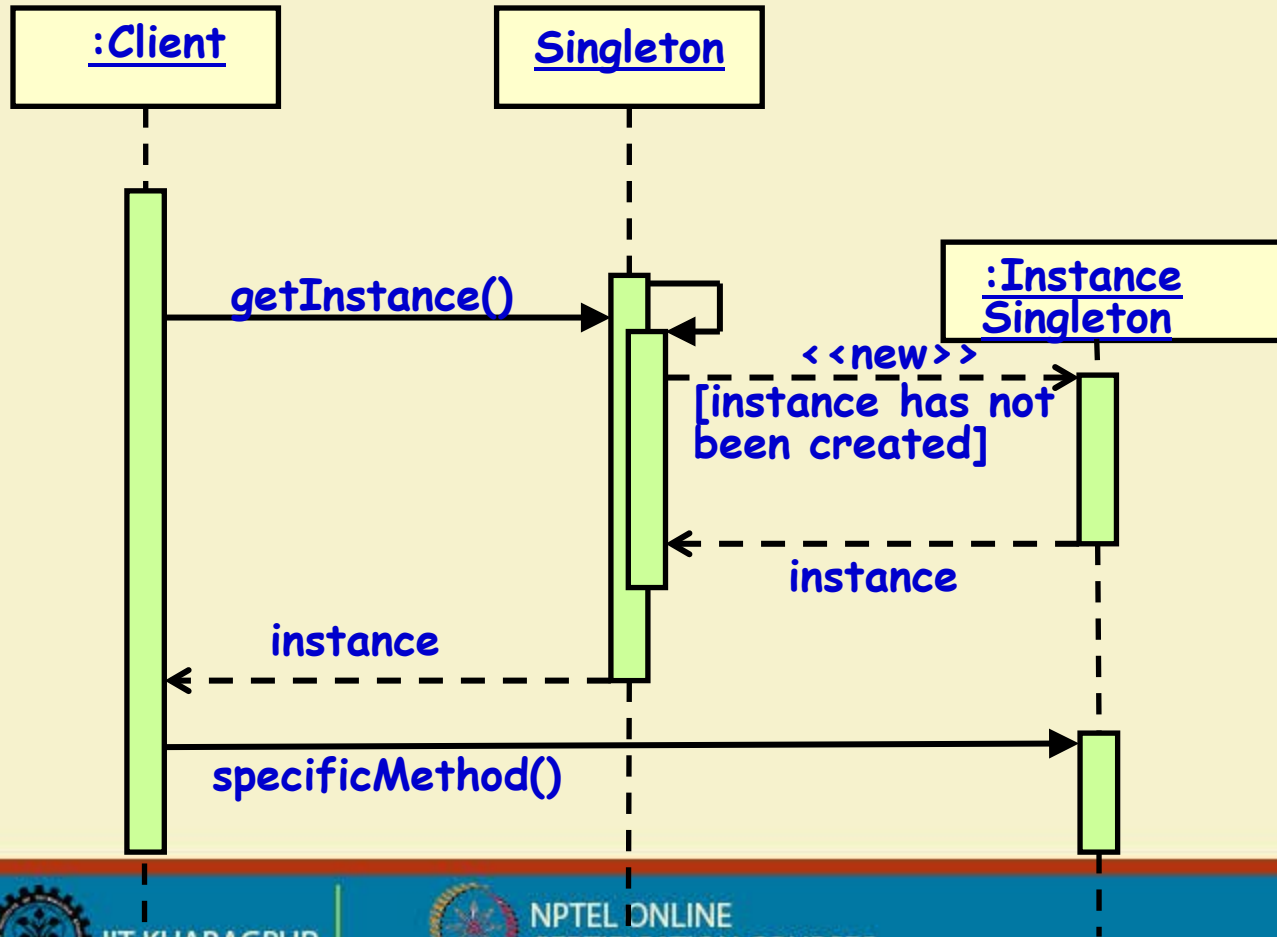
+singletonOperation()

Private constructor only accessible via getInstance()

-Singleton()

```
getInstance( ) {  
    if ( uniqueInstance == null )  
    { uniqueInstance = new Singleton( ); }  
    return uniqueInstance;  
}
```

Singleton: Sequence Diagram



Example Code

```
Public Class Singleton {  
  
    private static Singleton uniqueInstance = null;  
    // Why is uniqueInstance static?  
  
    private Singleton( ) { .. } // private constructor  
  
    public static Singleton getInstance( ) {  
        if (uniqueInstance == null)  
            uniqueInstance = new Singleton();  
        return uniqueInstance;  
    }  
}
```

Exercise 1

- o In a certain application, various classes get a Database Manager using the call:

DBMgr.getDBMgr();

- o It needs to be ensured that:
 - o There is only one Database Manager object.
 - o Need to disallow creation of more than one object of this class.

Exercise 1 Solution

```
class DBMgr {  
    private static DBMgr pMgr=null;  
    private DBMgr() { } // No way to create outside of this Class  
    public static DBMgr getDBMgr() { // Only way to create.  
        if (pMgr == NULL) pMgr = new DBMgr();  
        return pMgr;    }  
    public Connection getConnection(); }  
}
```

Usage:

```
DBMgr dbmgr = DBMgr.getDBMgr();  
//Created first time called
```

- Why have all the complexity of Singleton?

Why not just use
a static method?

- Is a static method not enough?

- **Problem 1: lacks flexibility**

```
public class DBMgr{  
    public static getConnection();  
}
```

- Static methods can't be passed as an argument to a method, nor returned

- **Problem 2: cannot be extended**

- Static methods can't be subclassed and overridden like a singleton's could be.

```
public class RandomGenerator {  
    private static RandomGenerator gen = new  
        RandomGenerator();  
    public static RandomGenerator getInstance() {  
        return gen;  
    }  
    private RandomGenerator() {}  
    public double nextNumber() {  
        return Math.random();  
    }  
}
```

Fill Code Here...

Any problems?

Always creates the instance,
even if it isn't used...

Exercise 2: Random
Number Generator

Singleton Pattern: Insights

- **Singleton with lazy instantiation:**
 - The singleton instance is not created until the instance() method is called for the first time.
 - Ensures that singleton instance is created only when needed.
- **If subclassing is required: Singleton needs to implement a protected constructor:**
 - Clients cannot directly instantiate Singleton instances through new.
 - Protected constructors can be called by subclasses.

Multiple Singletons?

- In certain situations, two or more Singletons can mysteriously materialize:
 - **Disrupting the very guarantees that the Singleton is meant to provide.**
- Consider a web application being executed in a browser.
 - Each servlet uses its own class loader.
 - Static blocks are executed during the loading of class and even before the constructor is called.

Concurrent Executions...

// error as no synchronization on method

```
public static MySingleton getInstance() {  
    if (instance==null) {  
        instance = new MySingleton();  
    }  
    return instance;  
}
```

Concurrency

// Correct solution

```
public static synchronized MySingleton getInstance(){  
    if (instance==null) {  
        instance = new MySingleton();  
    }  
    return instance;  
}
```

Singleton Pattern: Variations

- **Multiton:**

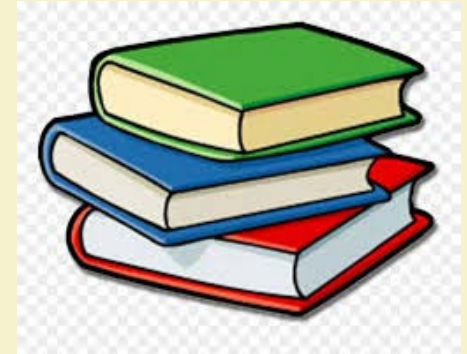
- We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
- **Of course, the operation that grants access to the Singleton instance needs to change.**

State Pattern

Introduction

- An object is behaving differently to the same message. What could be the reason?

- **Example:** consider the responses to the “renew” request for the same book:
 - Successfully renewed
 - Reserved, cannot be renewed
 - Book needed for stock taking, cannot be renewed
 - Already renewed five times, cannot be renewed ...



State of an Object

- One or more attributes of a class acts as state variable.
- Depending on the values of the state variables.
 - **Some methods exhibit different behavior.**
- An example:
 - A person may behave differently depending on his mood.

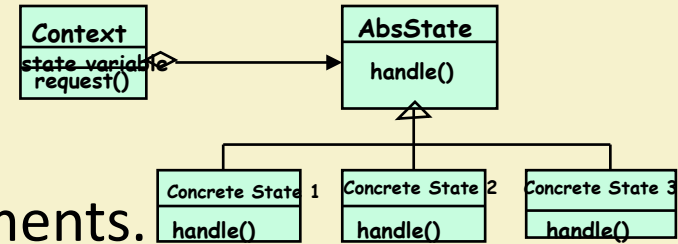
State Pattern

- A Behavioral pattern.
- Allows an object to alter its behavior when its state changes.
- **The object will appear to change its class.**
- To realize different behavior for different states of an object.
- **Polymorphism is used.**

Why Use A State Pattern?

- The state pattern creates a separate class for each conditional branch.

–This eliminates if/else or switch/case statements.



- Code is modular:

–Allows to easily change state behavior.

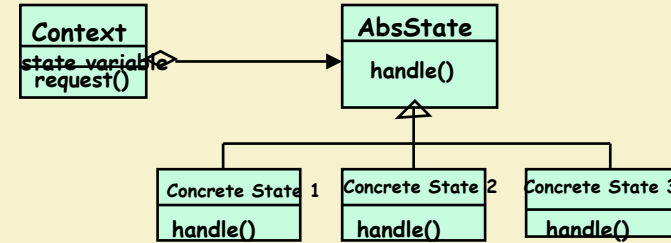
The State Design Pattern

- **Idea:** Value of an internal state variable determines Object behavior.
- **Intent:**
 - Allow an object to alter its behaviour when value the internal state variable changes.
 - The object will appear to change its class.

• Context class:

- It is the class which changes state.
- Context class maintains a reference to the current state object.
- To change the state,
 - The referenced object needs to be changed.

How is State pattern implemented?

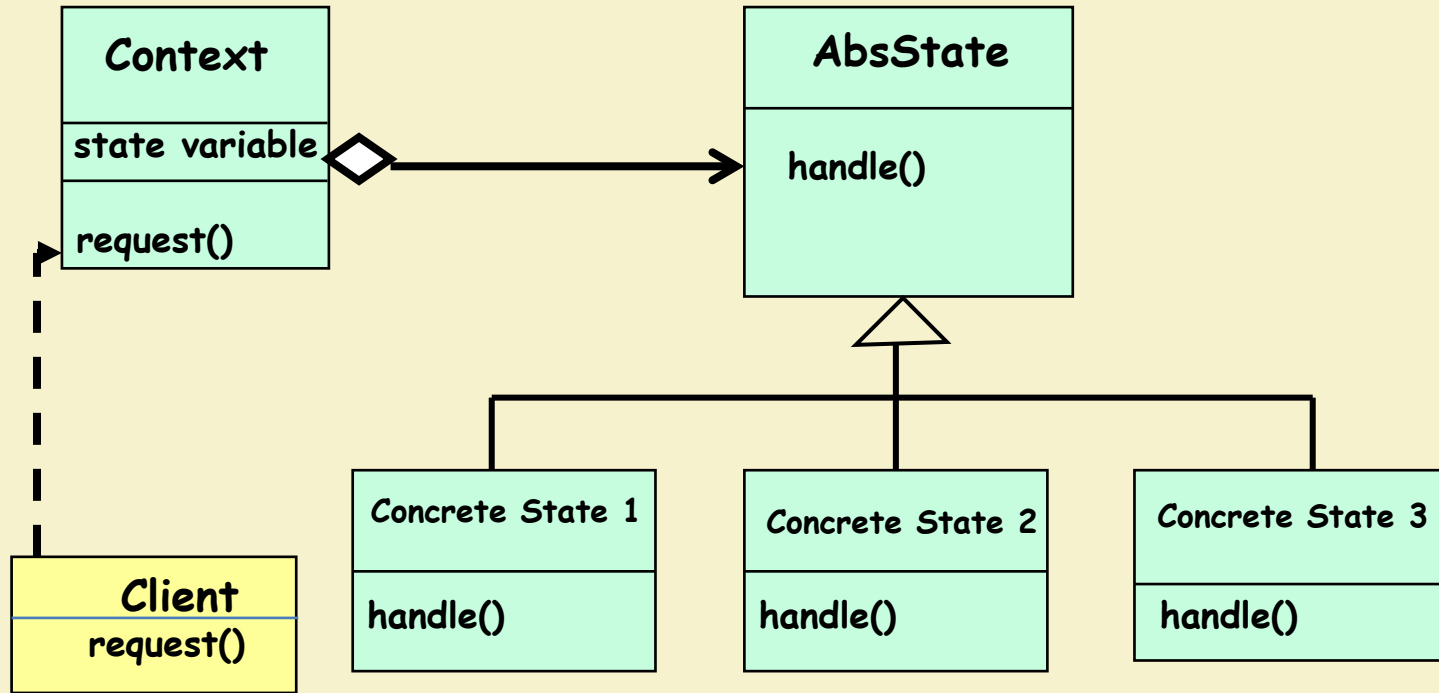


• Abstract State class

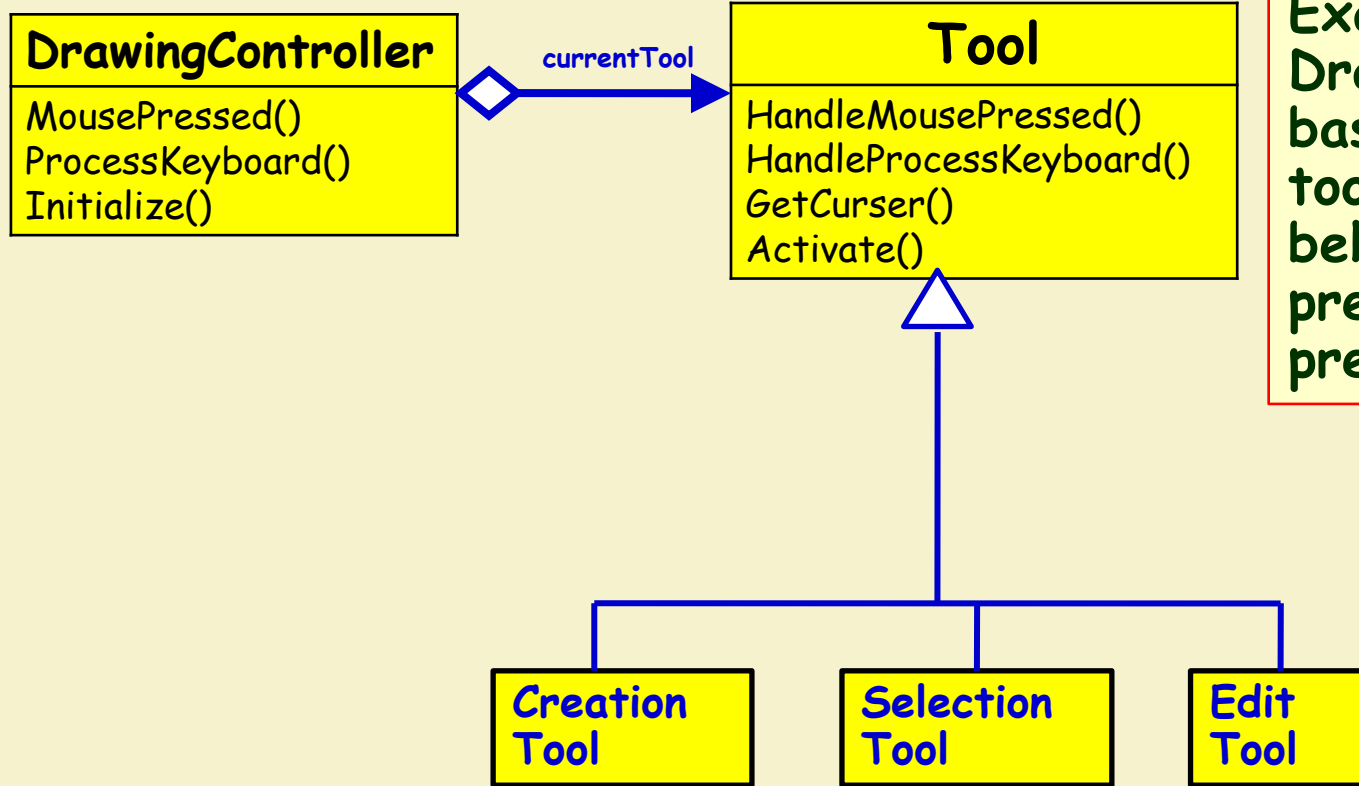
• Derived classes:

- Define the changed behavior in states.

State Pattern: Structure

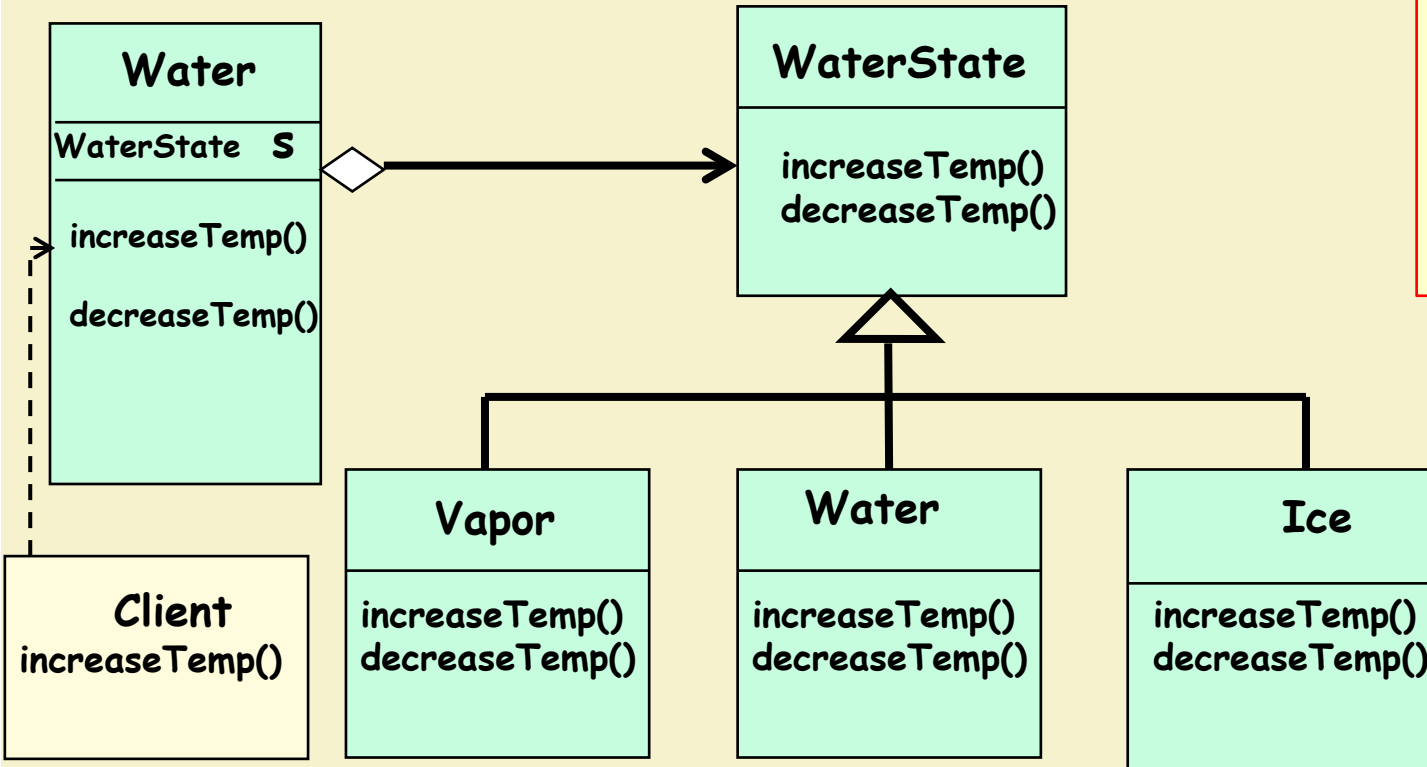


Allow an object to alter its behavior (bind to a different method) when its internal state changes.



Example 1: For a Drawing Package, based on the current tool selection, the behavior of mouse press, key board press, etc. vary...

Example 2: Behavior of Water Depends on Its State



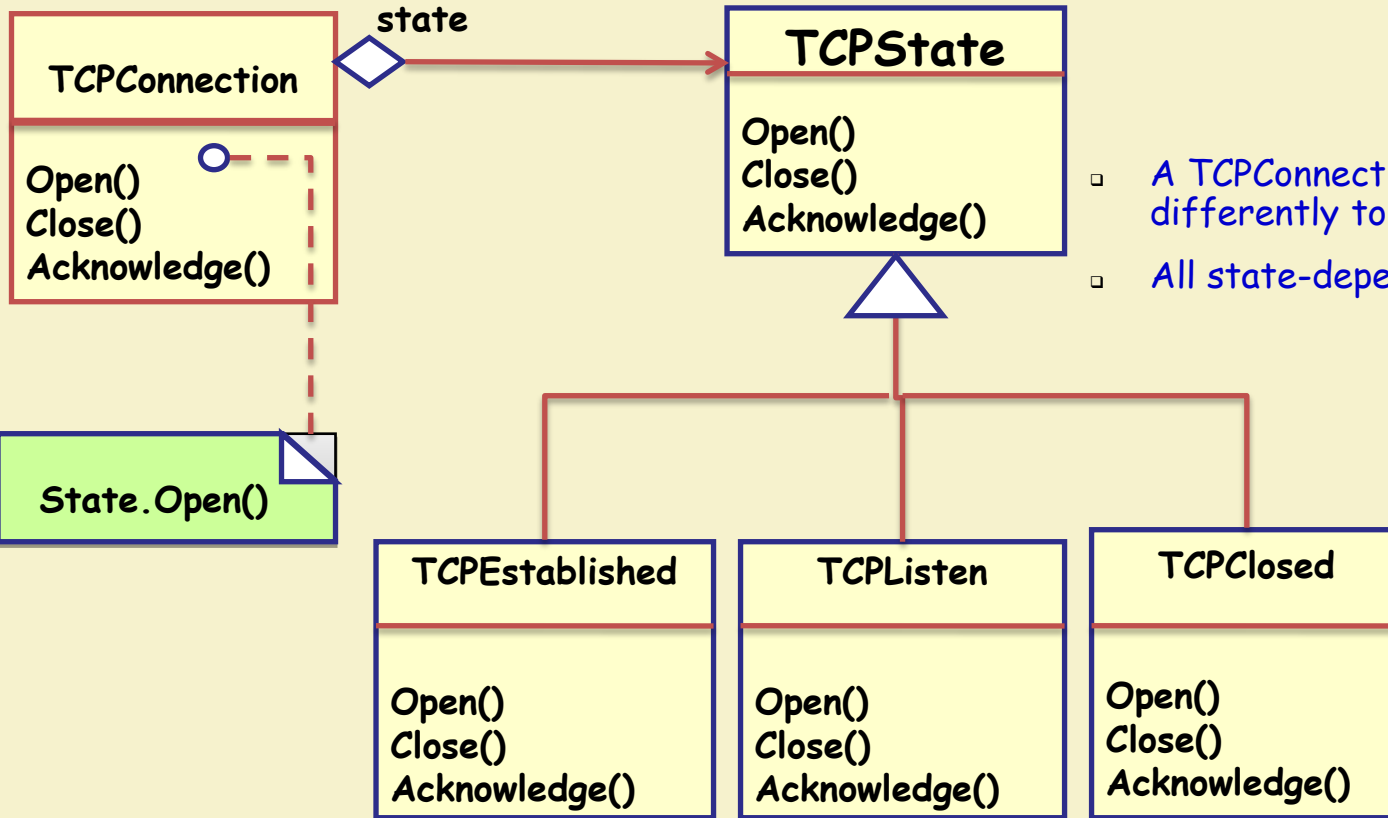
Exercise 1

- A TCP connection may be in any one of the following states:
 - Connection established
 - Listen
 - Close

- A `TCPConnection` object responds differently to requests at different states.
 - Open
 - Close

Exercise 1: Solution

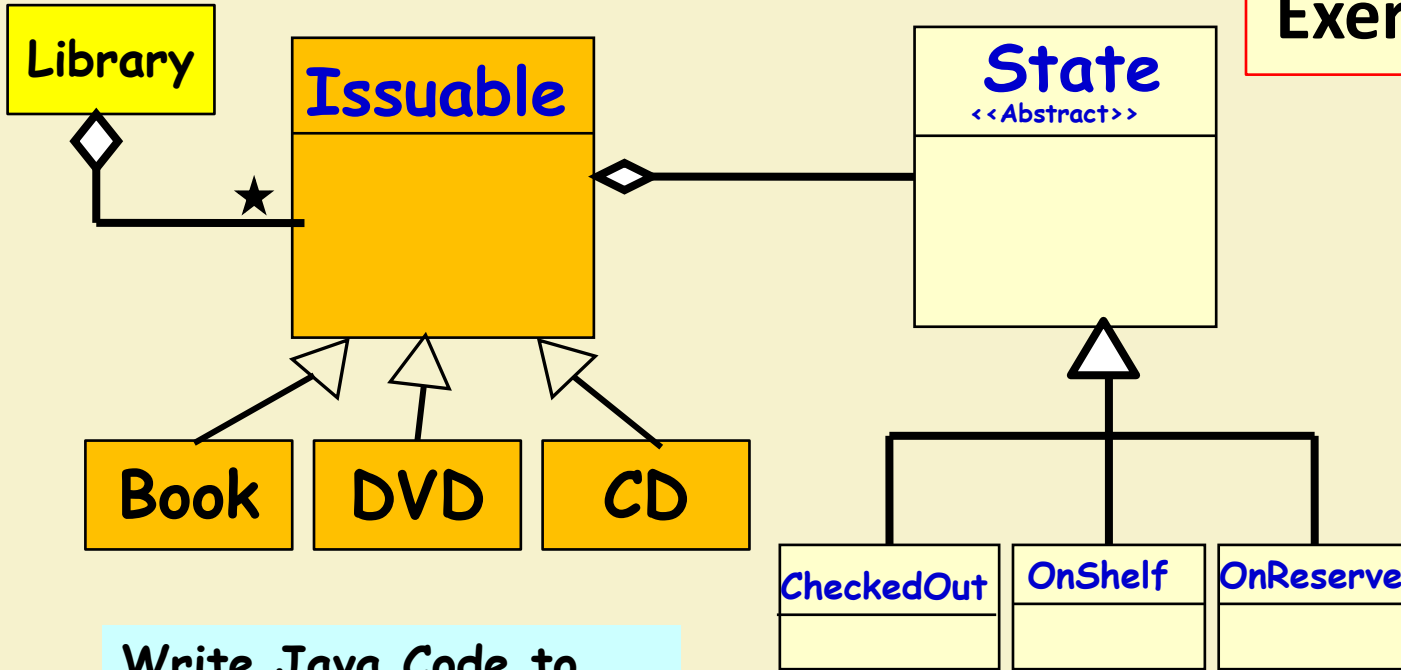
- A TCPConnection object responds differently to requests at different states.
- All state-dependent actions are delegated.



State Pattern: Exercise 2

- A Library has a large number of Issuables.
 - An issuable is either a book, a DVD, or Music CD.
- Each issuable can either be on shelf, issued out, or on reserve.
 - Response to request for issue and renew an issuable is different in different states.

Exercise 2: Solution



Write Java Code to Implement the State Pattern.

```
abstract public class State {
```

```
    protected boolean checkOut() { return false; }
```

```
    protected boolean putOnShelf() { return false; }
```

```
    protected boolean putOnReserve() { return false; }
```

```
}
```

```
public class CheckedOut extends State{
```

```
    private boolean putOnShelf() { return true; }
```

```
}
```

```
public class OnShelf extends State {
```

```
    private boolean checkOut() { return true; }
```

```
    private boolean putOnReserve() { return true; }
```

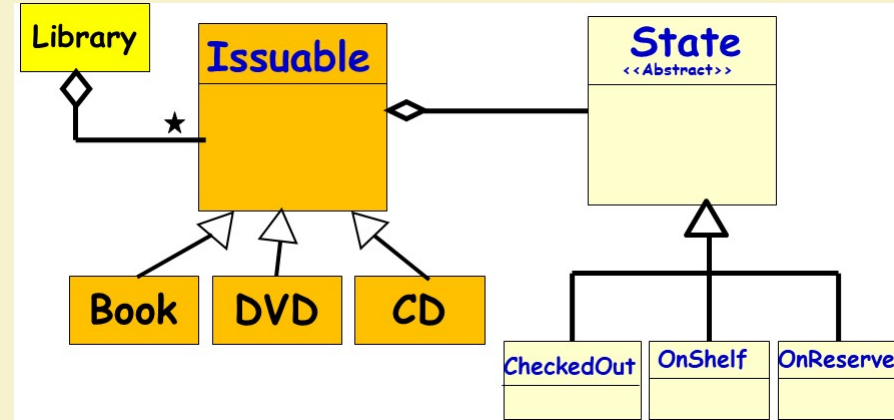
```
}
```

```
public class onReserve extends State {
```

```
    private boolean checkOut() { return true; }
```

```
}
```

Exercise 2: Code



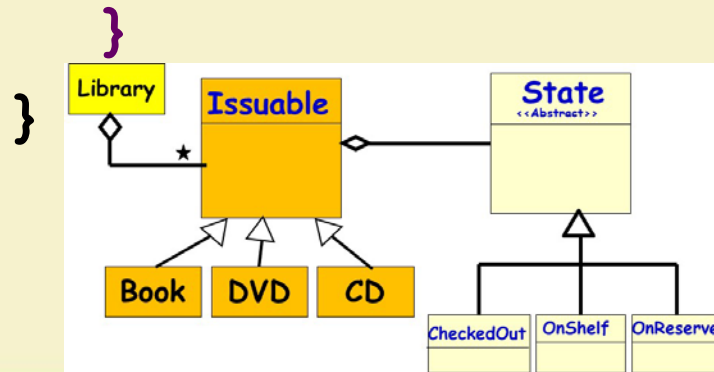
```
abstract public class Issuable{  
    State state=new onShelf();
```

```
    public boolean checkOut(){  
        if(state.checkOut() == true) {  
            changeState(new CheckedOut());  
            return true;}  
        else return false;  
    }  
}
```

Exercise 2: State Pattern Code

```
    public boolean putOnShelf() { ... }  
    public boolean putOnReserve() { ... }
```

```
    public void changeState(State newState){  
        state = newState;  
    }  
}
```



Advantages

- Encapsulates behavior of a state into an object
- Eliminates long lines of code involving if/else or switch/case statements with state transition logic
- State changes occur using one object, therefore avoids inconsistent states.

Disadvantages

- Increased number of objects.

Trade-offs

- **Advantages**

- Switch-case statement is not efficient.
On the average, half of the options need to be examined.
- Code becomes modular:
 - Otherwise, behavior for all states get handled in one place (switch-case):
 - Any changes to state logic would need recompilation and retesting.

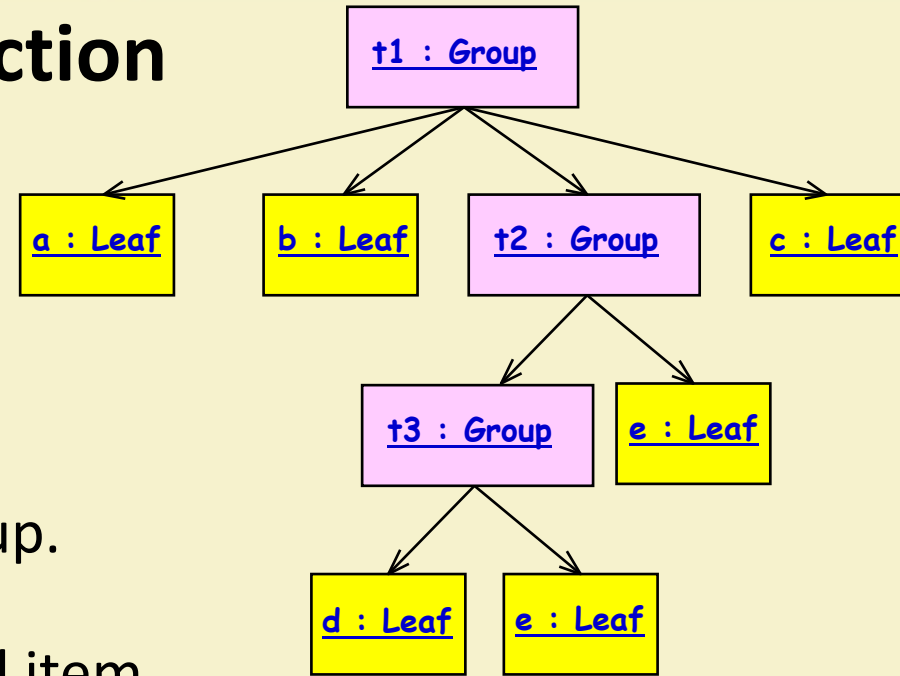
Food for thought...

- **Where to define the state transitions ?**
 - For simple cases, transition can be defined in the context.
 - More usable if transition is specified in the State subclass.
- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
 - First one is desirable if state changes are infrequent.
 - Later one is desirable if the state changes are frequent.

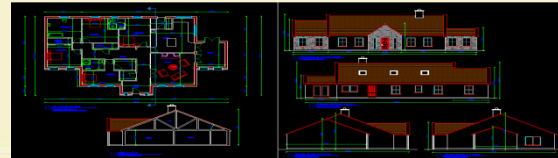
Composite Pattern

Composite: Introduction

- A composite is a group of objects in which some objects contain others:

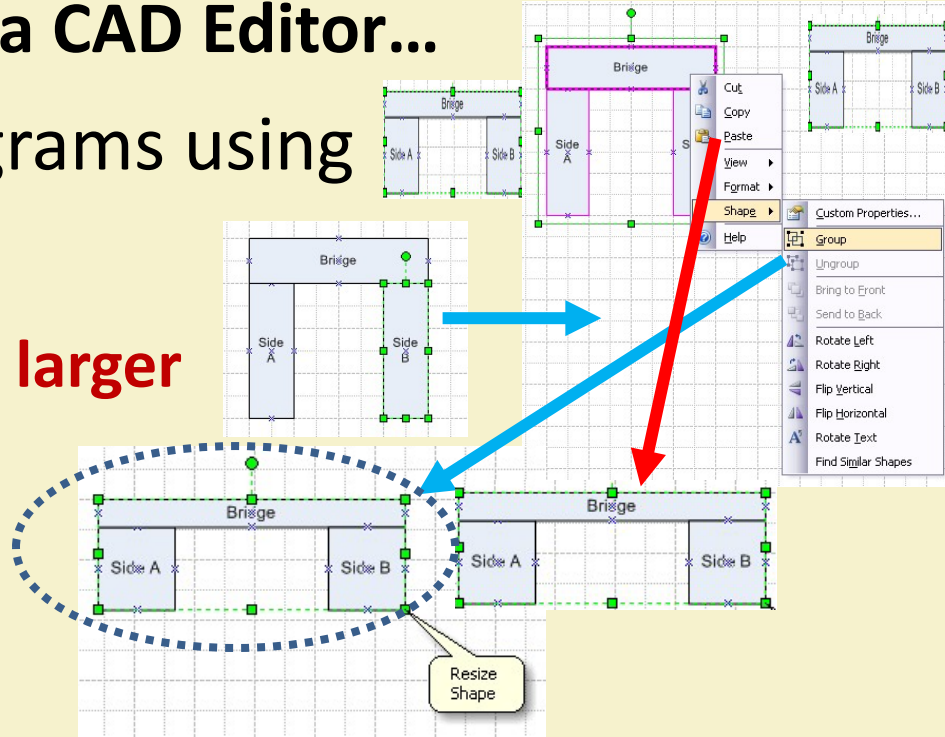


- An object may represent a group.
- Or may represent an individual item.
- Example: A CAD Design ---



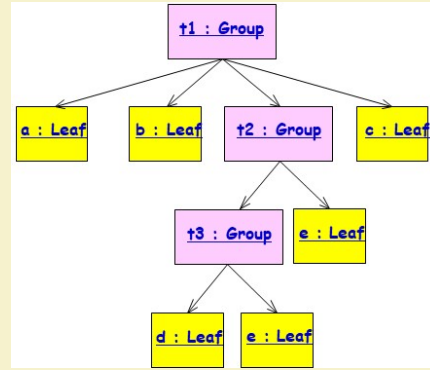
Example: Consider a CAD Editor...

- You can build complex diagrams using simple components
 - Group components to form larger components...
 - ...which in turn can be grouped to form still larger components



Composite Pattern: Intent

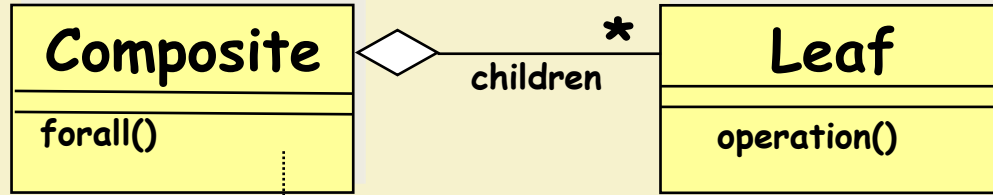
- Compose nested groups of objects into a tree structure to represent part-whole hierarchies.
 - **Clients should be able to treat individual objects and composites in the same way.**



Why Composite Pattern? (Motivation)

- What problems would occur if composite pattern is not used?
 - Client code might have to treat primitive and container classes differently...
 - Makes the application more complex.
 - Additions of new types of components becomes troublesome...

Problem: handling nested group of objects



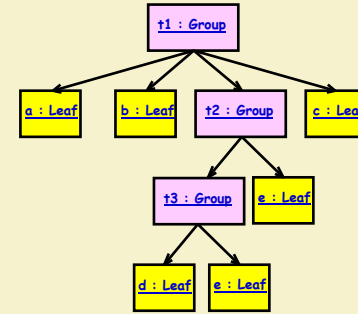
For all children c:
c.operation()

Lets see how a typical
Mark=1/100
from scratch...

Client Code:

```
...
if X is
  Composite
  then
    X.forall()
  else
    X.operation();
...
```

Composite Solution: First Attempt



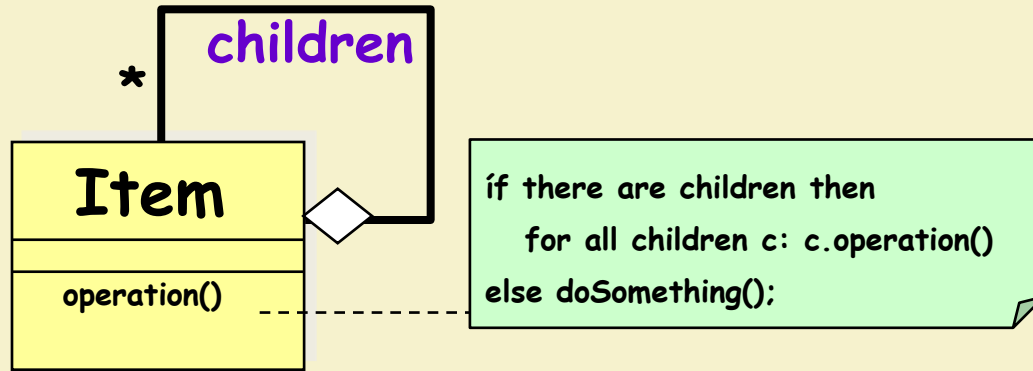
Analysis of Solution: Naive solution...

What are the problems?

- Only single nesting level (depth =1)
- Composite and leafs always treated differently, Difficult to extend.

Composite:
Attempt 2

Mark=
40/100



Surely there are improvements...

Unified treatment in client **and** unrestricted depth of parts...

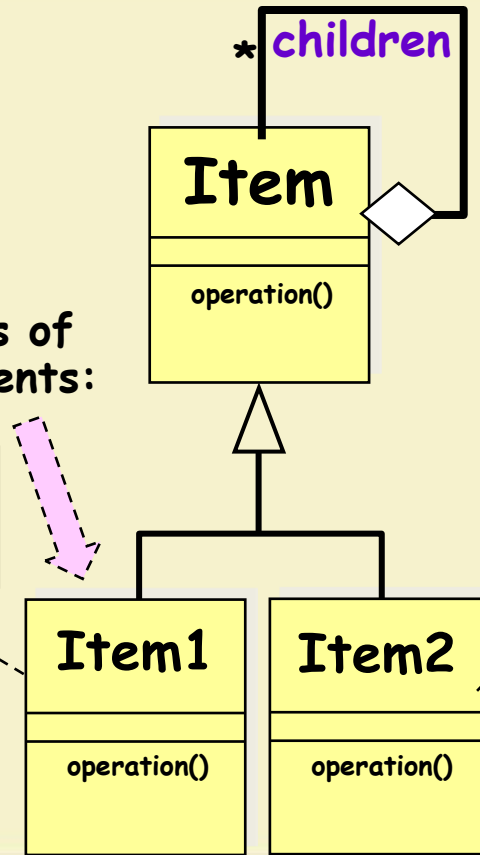
What are the problems?

- Does not handle different item types: primitive and composite
- Difficult to extend with new kinds of leafs or composites.

Attempt 3: Handling different Types of Items

Mark=
50/100

New
types of
elements:

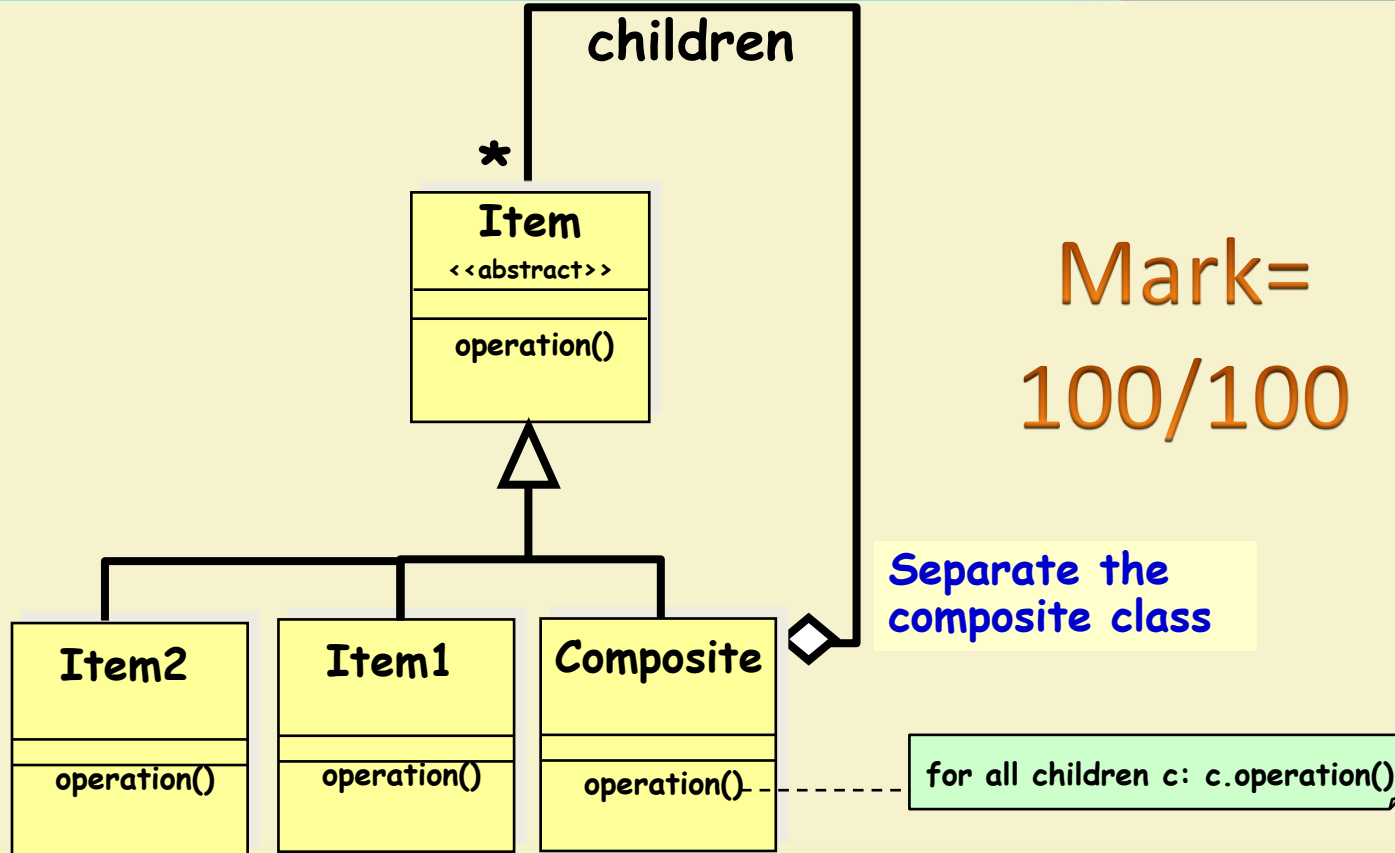


If there are children then
for all children c: c.operation()
else doOperation1()

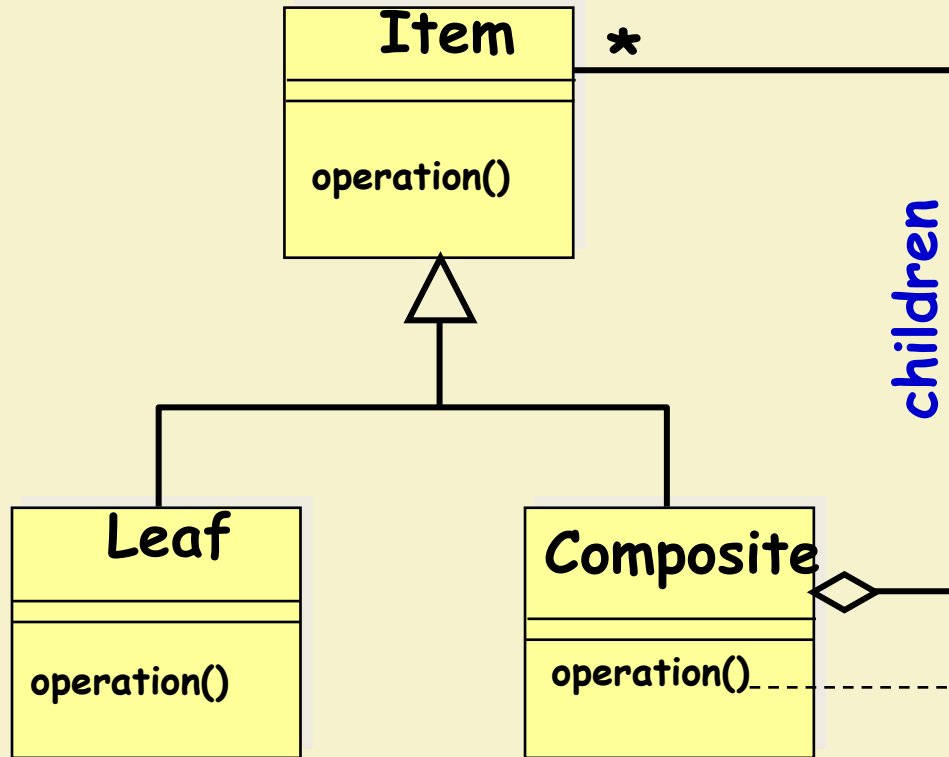
if there are children then
for all children c: c.operation()
else doOperation2()

Finally:
Composite
Pattern

Mark=
100/100



Composite Design Pattern



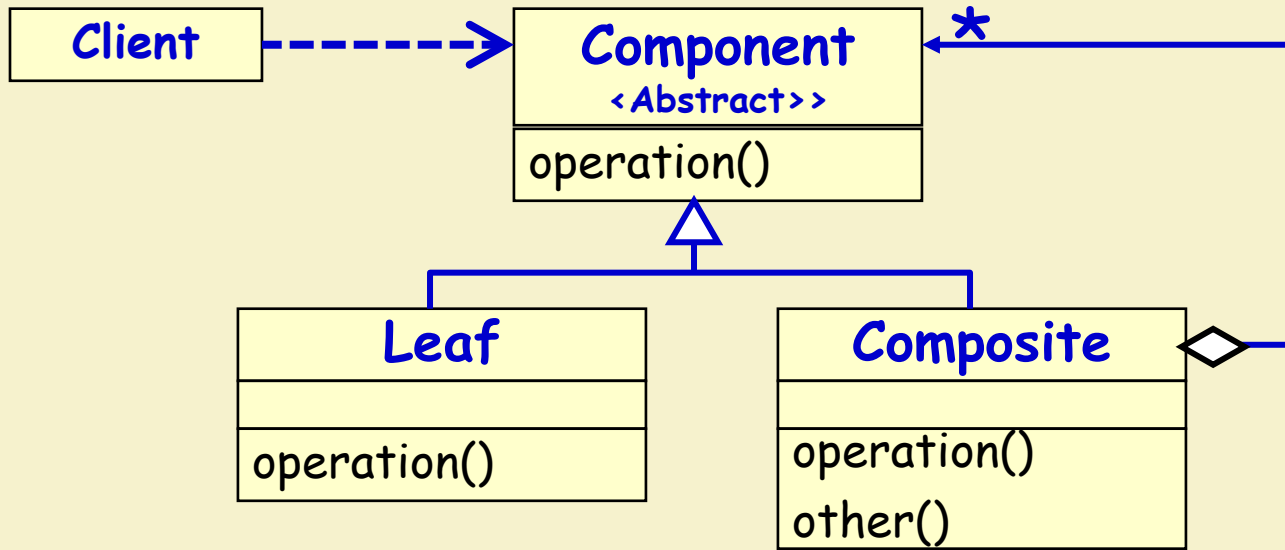
Problem: How to organize a hierarchical object structure so that the clients are not required to be aware of the hierarchy?

For all children *c*:
c.operation()

Composite Pattern: Issues

- What is the class diagram?
- How does the client interact?
- What operations are defined for:
 - The component, the composite, and the leaf?
 - How are they carried out?
- How is the design implemented?

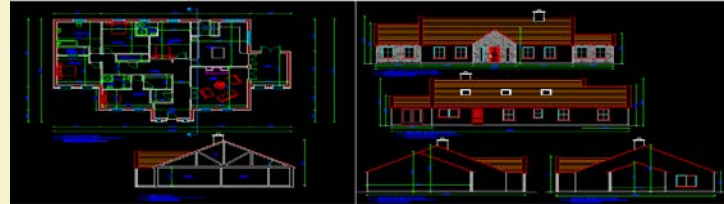
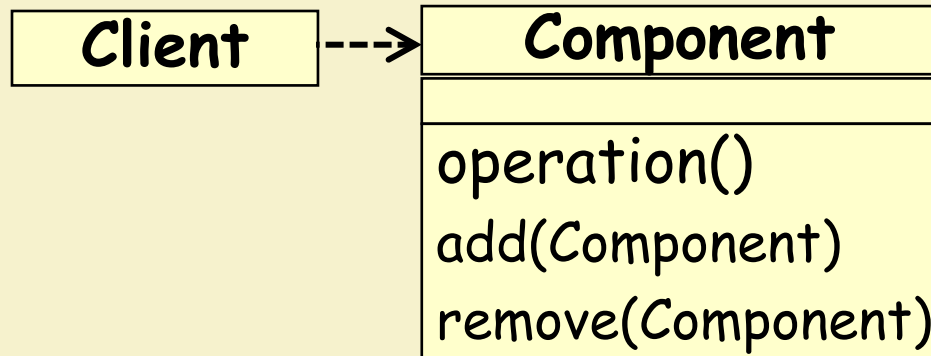
Composite Pattern



- Each node of the Component structure should respond to some common operation(s).
- The client can call operation of the Component and the structure responds “appropriately”.

Client

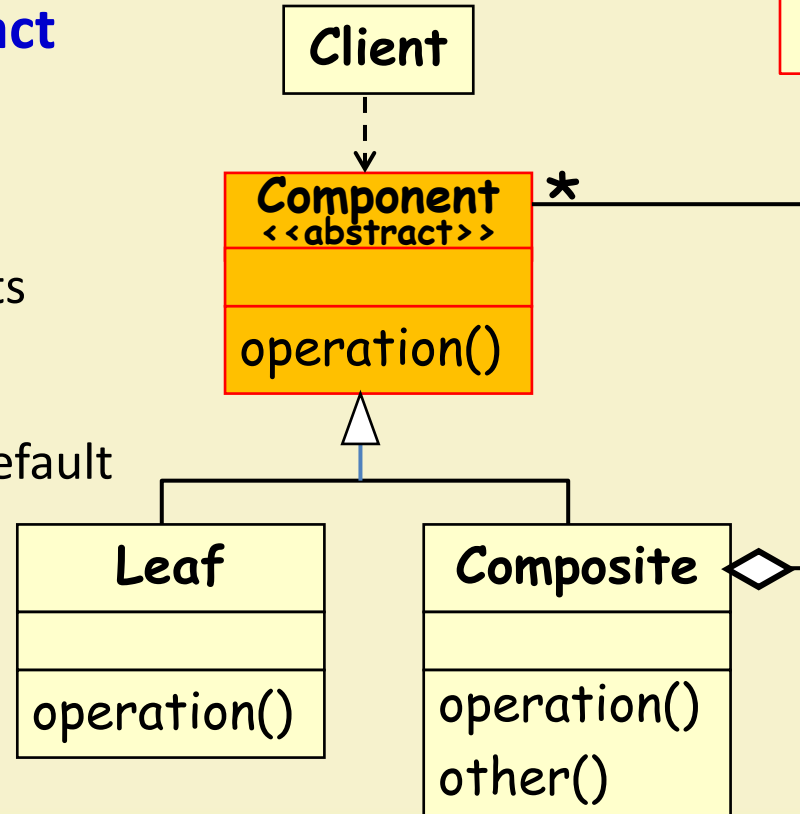
- The client is a class invokes composite:
 - Manipulates objects in the composition through the Component's interface
- Example: A CAD Drawing



Component

- Component is an abstract class:**

- Declares the interface for accessing and managing its child components
- Defines an interface for default behavior.
- Optionally provides access to the parent component

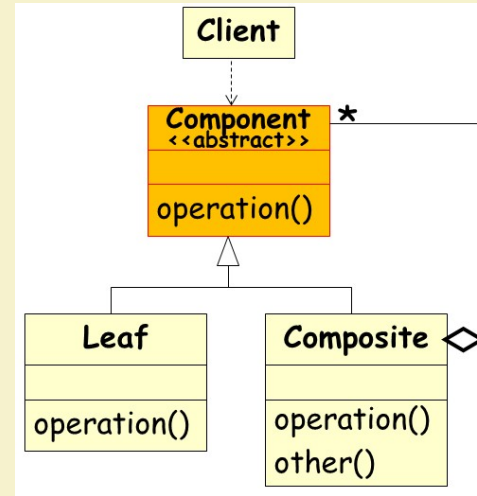


- **Leaf**

- A leaf has no children.
- Defines behavior for primitive objects in the composition.

- **Composite**

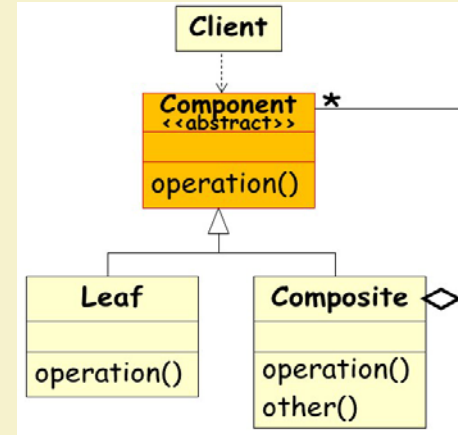
- Defines behavior for components having children.
- Stores child components.
- Implements child-related operations in the Component interface.



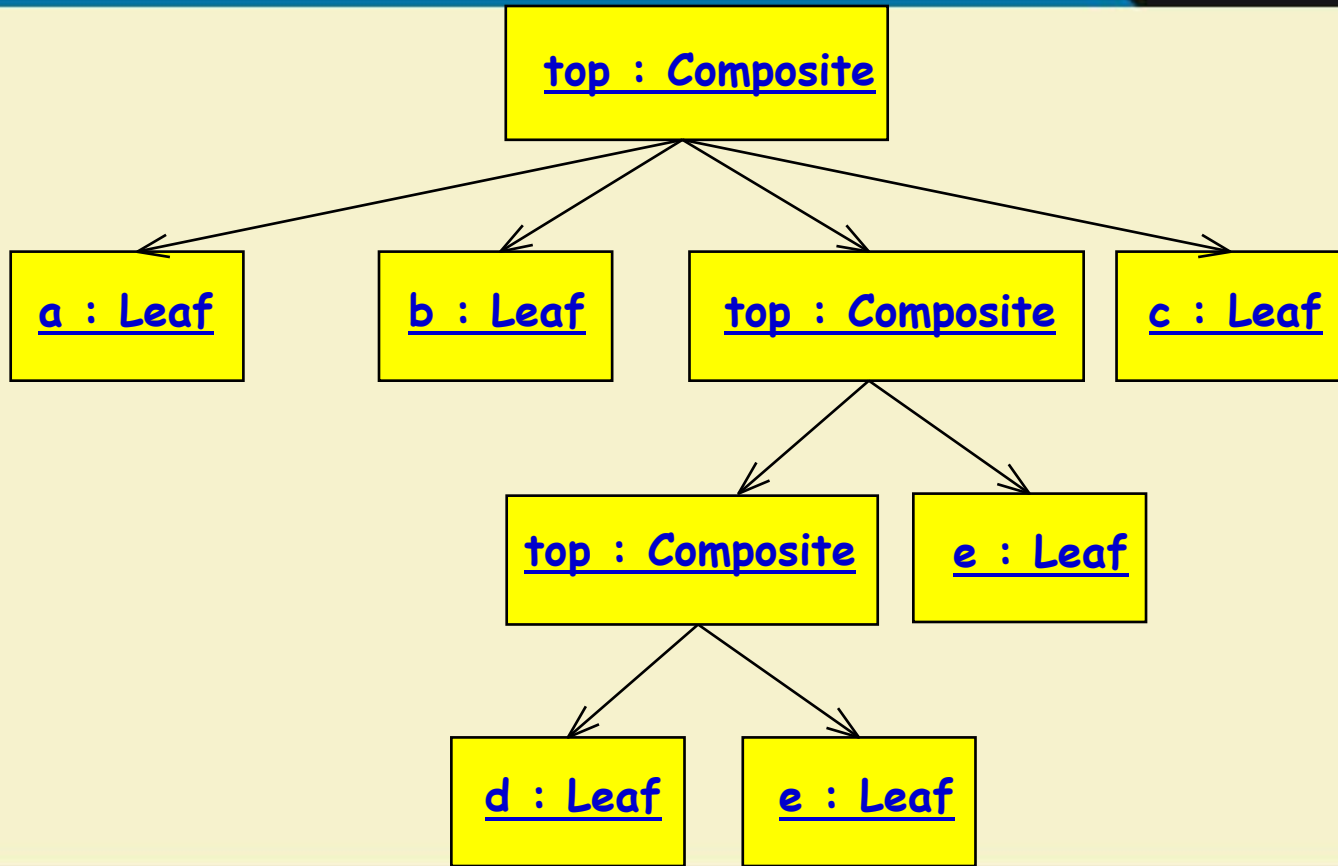
**Other
Participants**

Collaborations

- Clients use the Component class interface, which in turn interacts with objects.
- If the recipient is a Leaf:
 - **Handles the request directly...**
- If the recipient is a Composite:
 - **Forwards the request to its child components...**

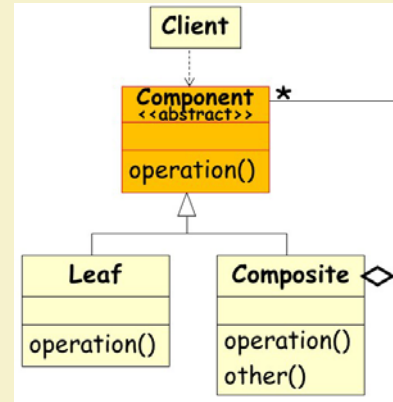


Composite: Object Diagram



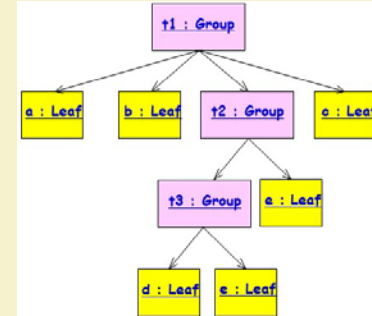
Consequences

- **Makes it possible to define recursive composition of primitive and composite objects.**
- Makes invocations by client simpler.
 - Client doesn't need to know whether it is dealing with leaves or composites.
- Makes it easier to add new kinds of components.



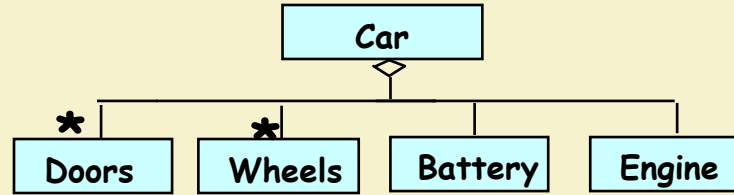
Applicability

- Use the Composite pattern when:
 - You need to represent part-whole hierarchies of objects
 - You want clients to ignore the differences between parts and wholes
 - **The parts should be created dynamically – at run time:**
 - Example: to build a complex system from primitive components and previously defined subsystems.
 - **This is especially important when the construction process will reuse subsystems defined earlier.**

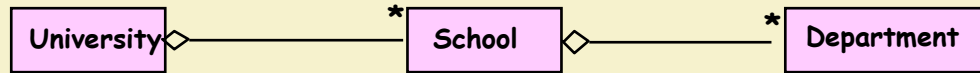


The Composite Patterns model dynamic aggregates

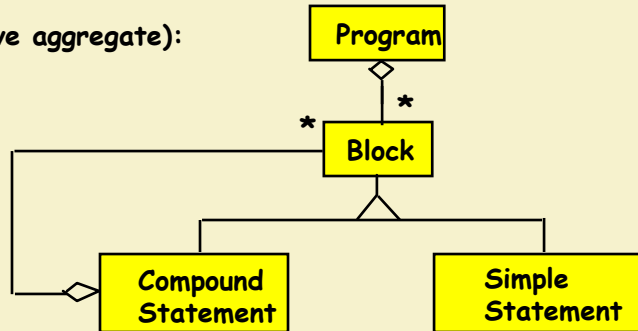
Fixed Structure:



Organization Chart (variable aggregate):



Dynamic tree (recursive aggregate):



Composite example: Jpanel



JPanel, a part of Java **Swing** package. It is a container that can store a group of components. The main task of **JPanel** is to organize components

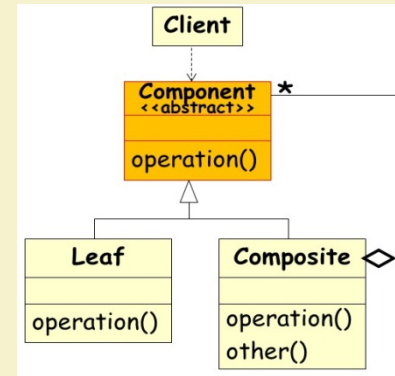
```
Container north = new JPanel(new FlowLayout());
north.add(new JButton("Button 1"));
north.add(new JButton("Button 2"));

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

// overall panel contains the smaller panels (composite)
JPanel overall = new JPanel(new BorderLayout());
overall.add(north, BorderLayout.NORTH);
overall.add(new JButton("Center Button"), BorderLayout.CENTER);
overall.add(south, BorderLayout.SOUTH);

frame.add(overall);
```

Some Insights

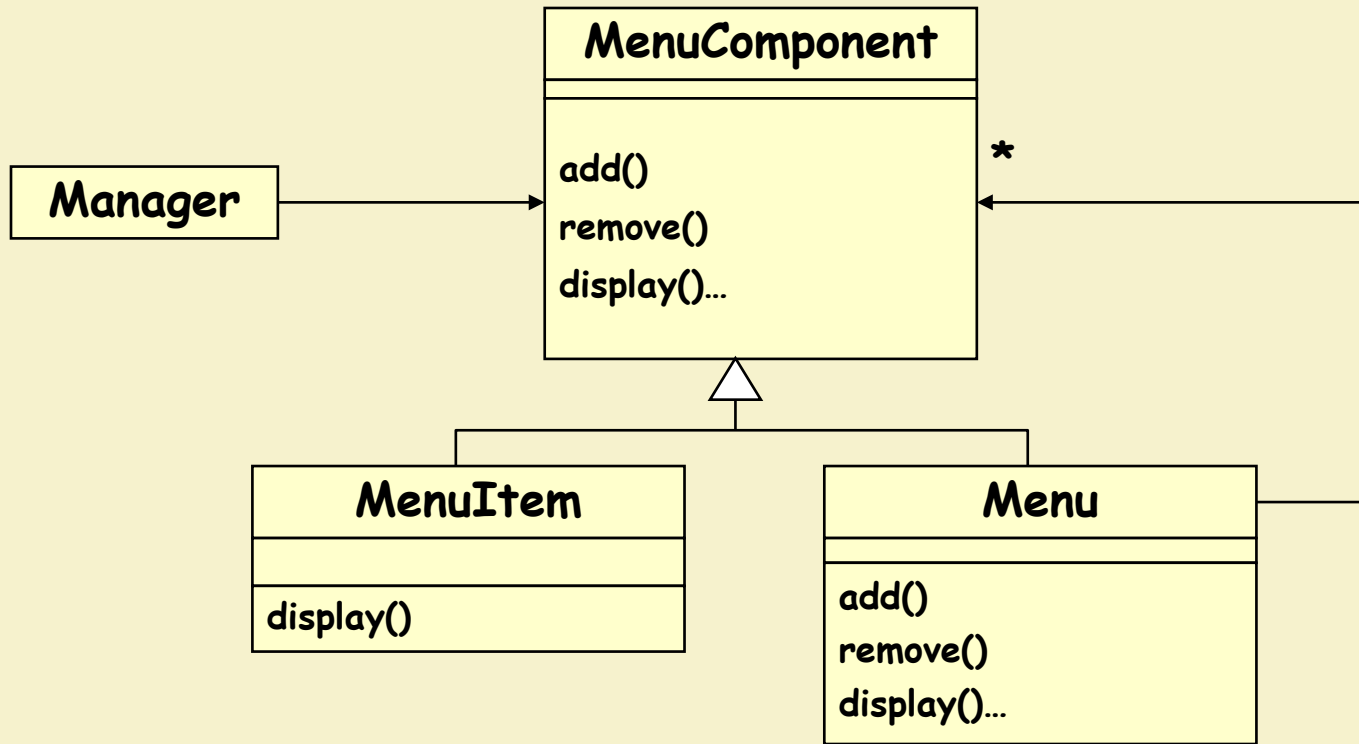


- Why do you declare the methods to handle children in the abstract class?
 - Only the composite class has any use for them?
 - Is it not poor programming practice to have these methods inherited by primitive classes, which have no use for them?
- **There is a tradeoff here between safety and transparency**

- If the child management methods are moved from the abstract class to the composite:
 - The client can no longer call these methods on primitive objects, improving elegance.
- However, this gives primitive and composite objects different interfaces:
 - Which is what the design patterns attempt to avoid

Composite Pattern: Example 1

Class Structure
to store
Restaurant
Menu



How can the entire menu be displayed?

Menu

menuComponents: ArrayList

add()
remove()
display()...

Example 1

1. Menu to MenuComponents association implemented with an array list data type.
2. Let us examine the implementation of print() in Menu and in MenuItem classes...

Example 1: Code

```
Class Menu implements MenuComponent{... ..  
public void display() {  
    System.out.print("\n" + getName());  
    System.out.println(", " + getDescription());  
    System.out.println("-----");  
    Iterator iterator= menuComponent.iterator();  
    while (iterator.hasNext()) {  
        MenuComponent menuComponent =  
            (MenuComponent)iterator.next();  
        menuComponent.print();  
    }  
}
```

}

```
class MenuItem implements MenuComponent{ ... ..
```

```
public void display() {
```

```
    System.out.print(" " + getName());
```

```
    if (isVegetarian())    System.out.print("(v)");
```

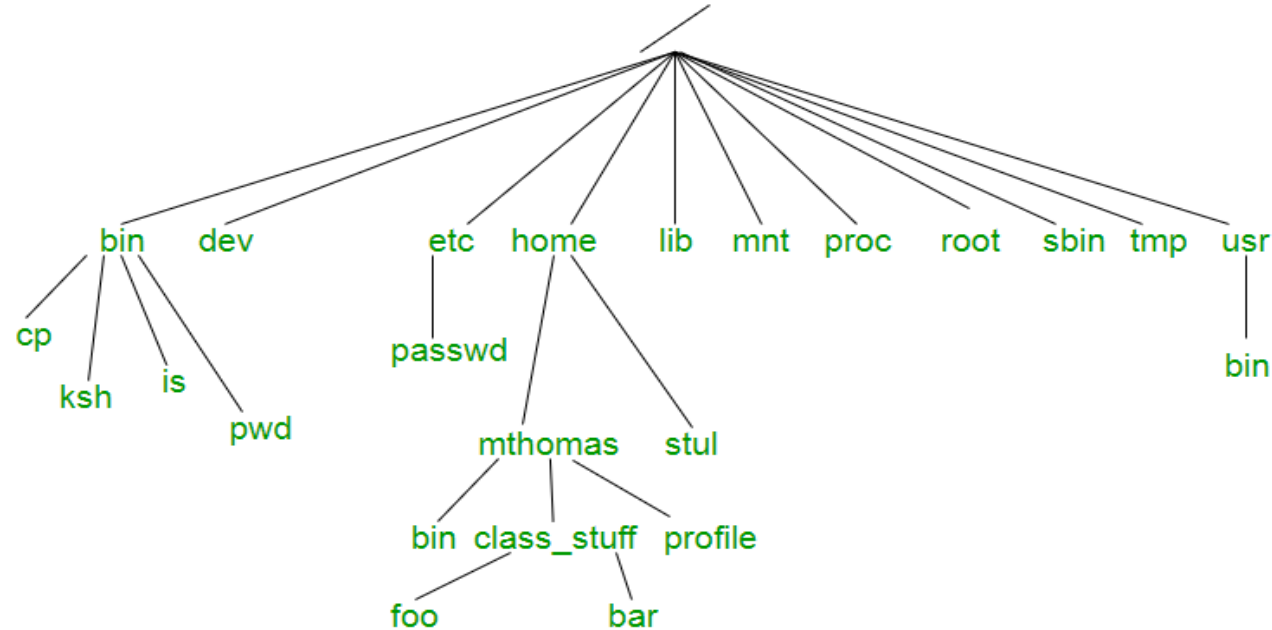
```
    System.out.println(", " + getPrice());
```

```
    System.out.println("- -" + getDescription());
```

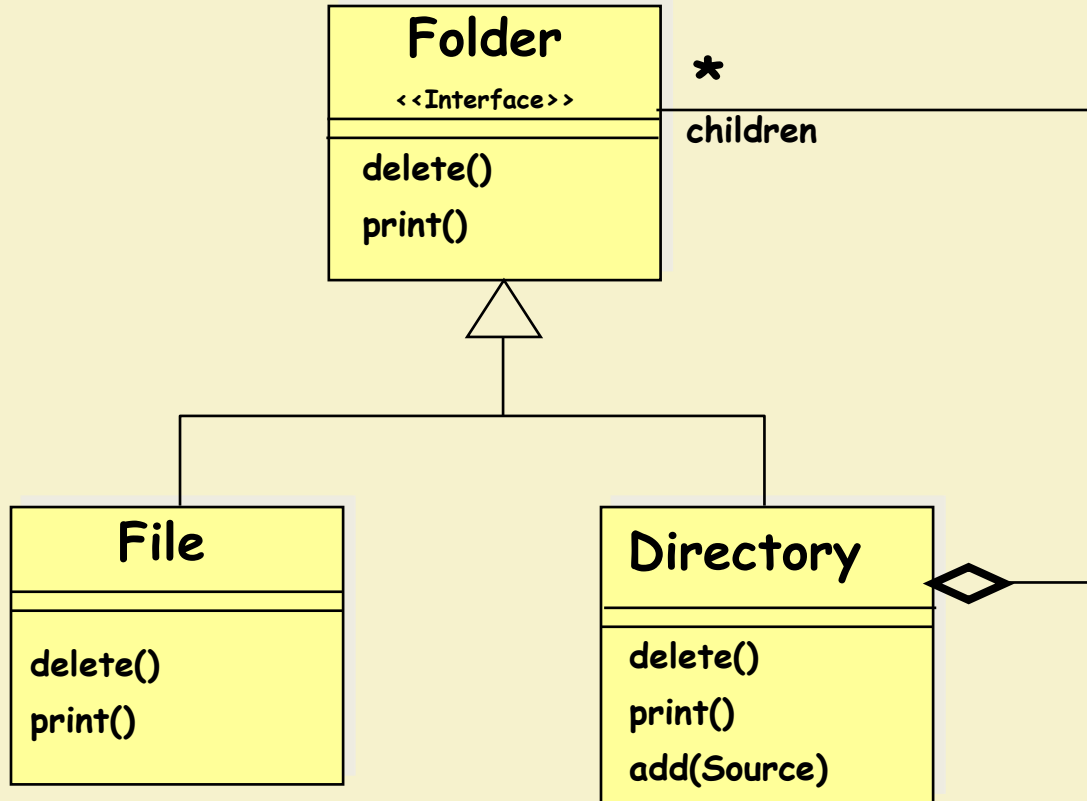
```
}
```

Exercise 1

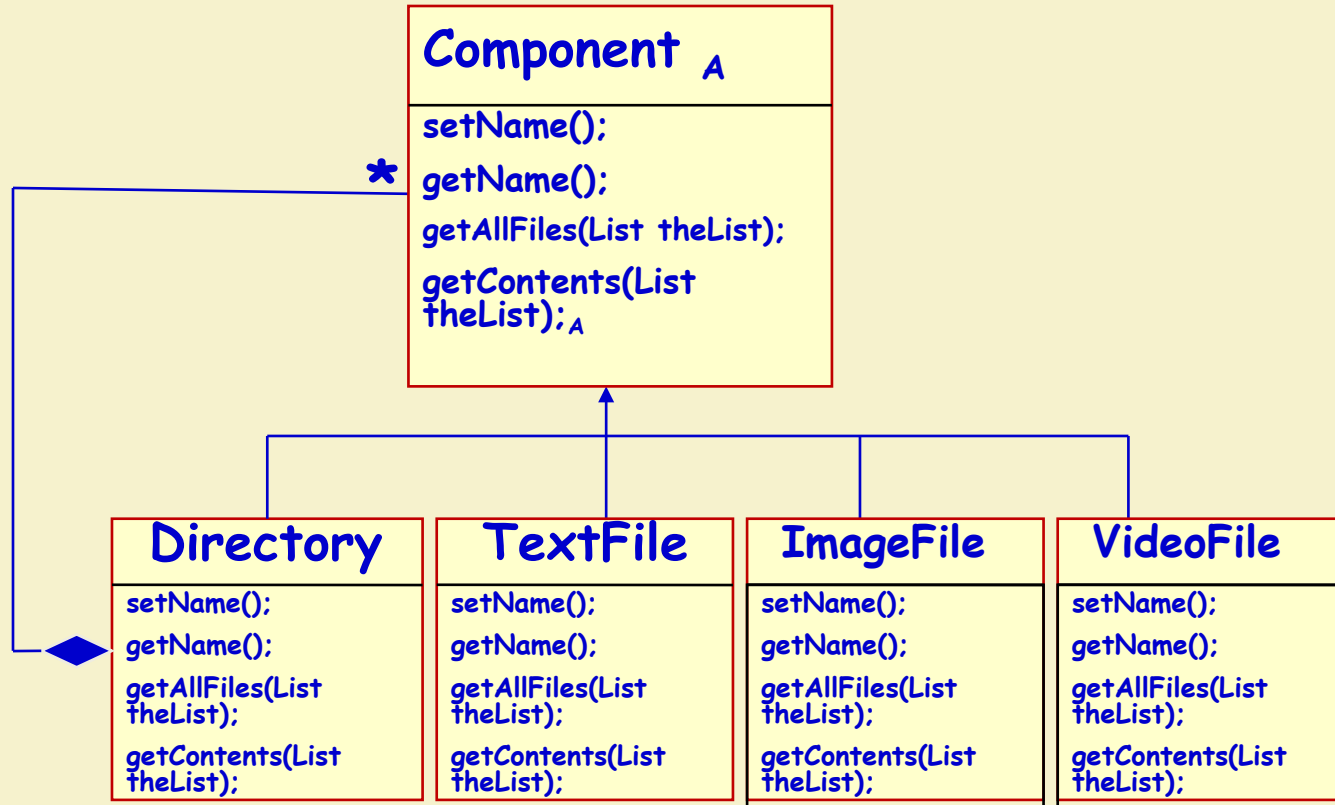
- Design a class structure to model a Unix type file hierarchy ...

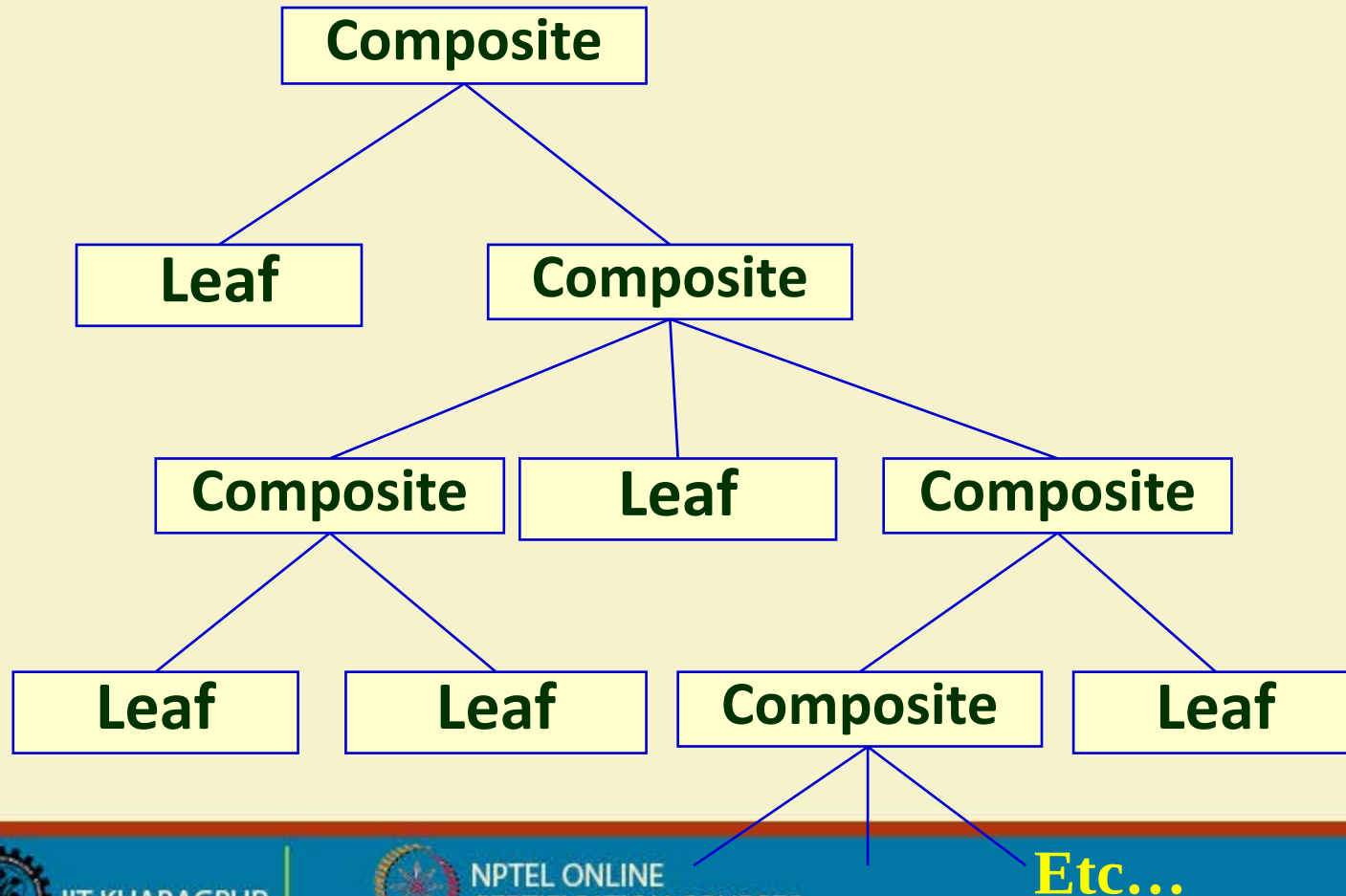


Exercise 1: Solution



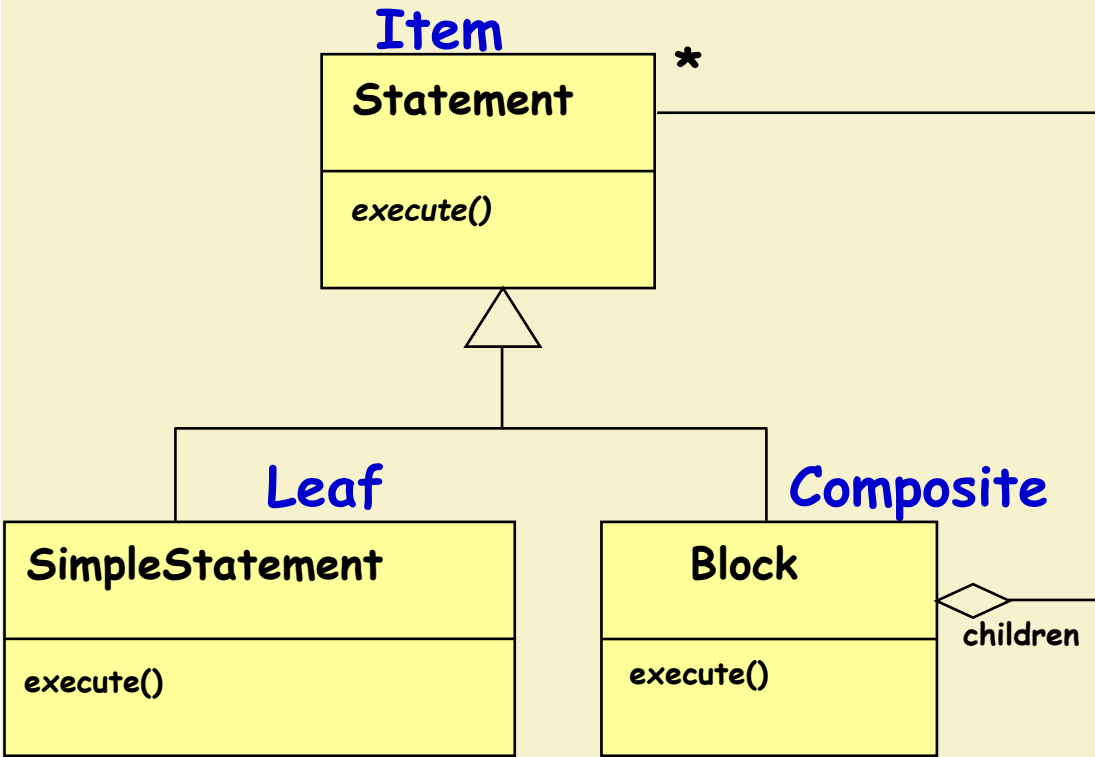
Now Suppose file types
are text, image,
Video...



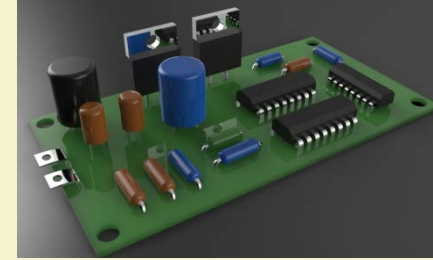


Exercise 2: Programs

A program block can contain simple statements or other program blocks...



Exercise 3

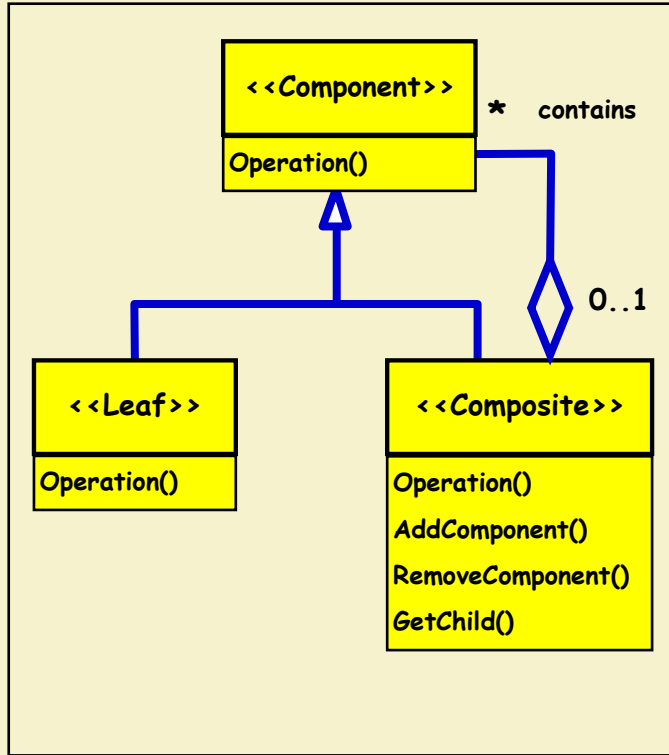


- Develop class design of components to be handled by an Electronic Circuit Diagram editor:
 - Should let users group simple components into larger components.
 - Which in turn can be grouped to form still larger components.
 - Larger components should behave similarly w.r.t. select, copy, paste, move, delete, resize, ...

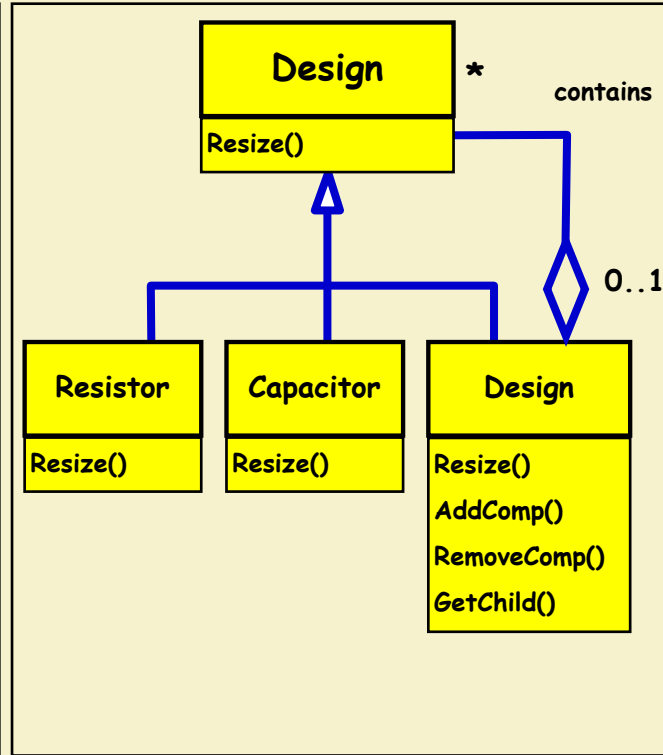
Solution

- The key to the Composite pattern:
 - An abstract class that represents both primitives and their containers.
 - The abstract class Design declares operations like Copy, Move, Delete, resize, etc. that are specific to graphical objects.
 - It also declares operations that all composite objects share, such as
 - Operations for accessing and managing its children, like Add, UnGroup.

Composite Pattern: Solution

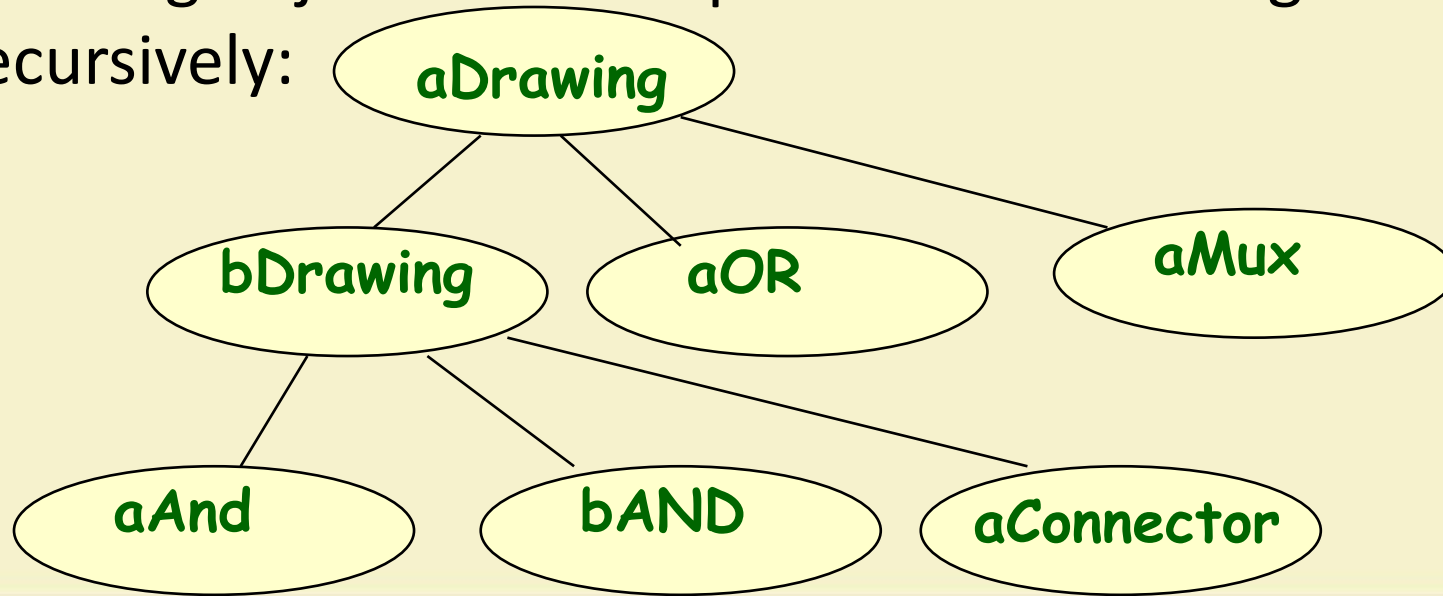


General Idea



Applied to the designing example

- Because the Drawing interface conforms to Design interface,
- Drawing objects can compose other Drawing recursively:



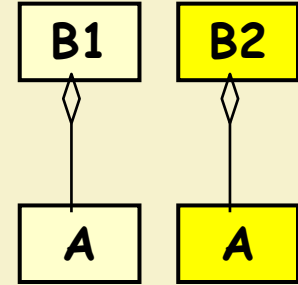
- Component does not know what it is a part of:

Alternatives...

- Component can be in many composites
 - Component can be accessed only through composite
-
- Component knows what it is a part of
 - Component can be in only one composite
 - Component can be accessed directly

Composite: Some Issues

- When components are part of a single composite:
 - A is a part of B if and only if B is the composite of A
 - However, duplicating information can be dangerous!
- **Problem:** How to ensure that references of components to composite and composite to components are consistent?



- The public operations on components and composites are:
 - Composite can enumerate components
 - Component knows its container
 - Add/remove a component to/from the composite
 - **The operation to add a component to a composite updates the container of the component**
 - There should be no other way to change the container of a component

addChild() in Composite

```
public void addChild(Component child) {  
    childArray.add(child);  
    child.setParent(this);  
}
```

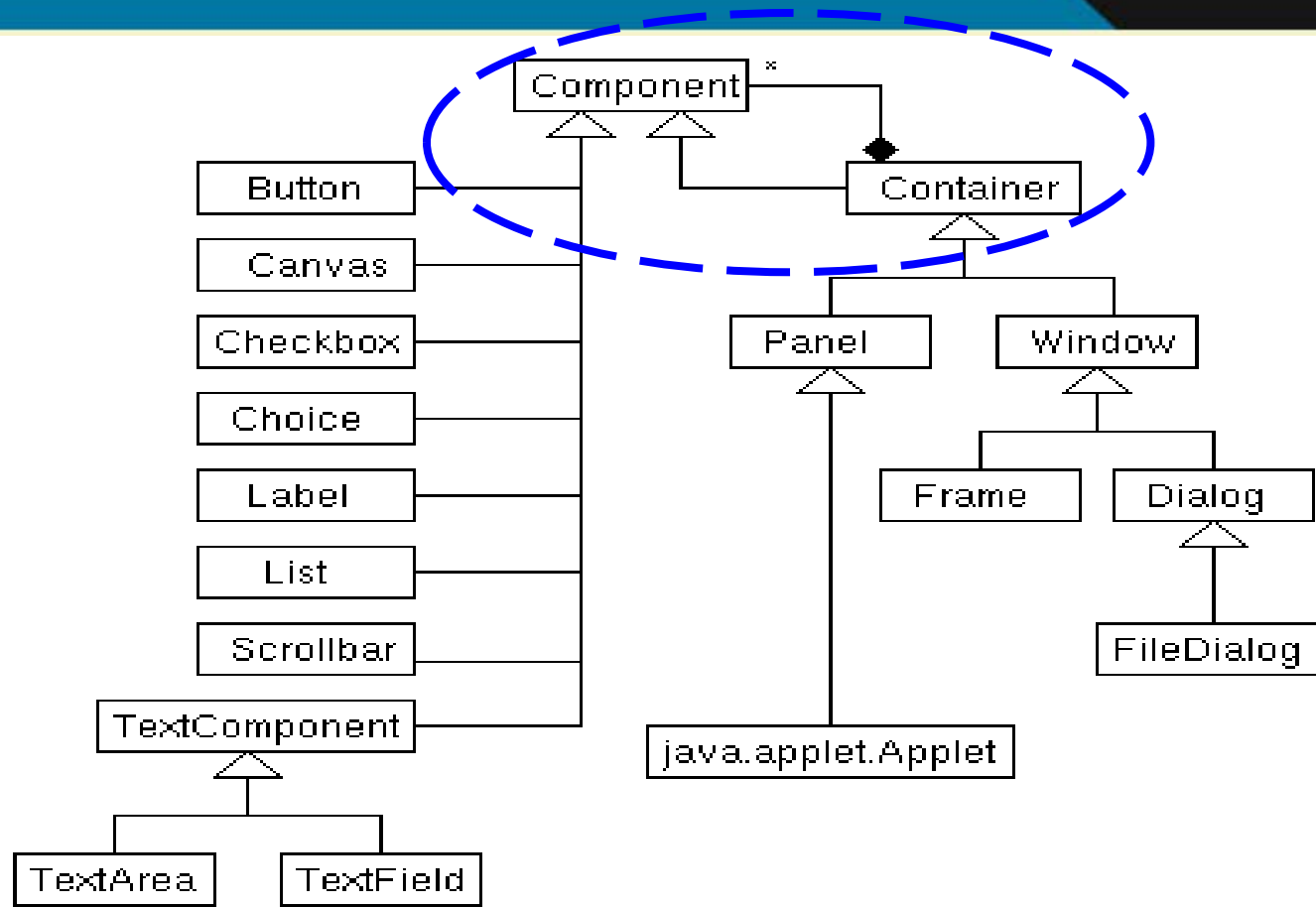
Exercise 4: Java GUI

Q: How can we add any widget to another, for example panels to an applet?

```
public class MyApplet extends java.applet.Applet {  
    public MyApplet() {  
        add(new Label("My label"));  
        add(new Button("My button"));  
        Panel myPanel = new Panel();  
        myPanel.add(new Label("Sublabel"));  
        myPanel.add(new Button("Subbutton"));  
        add(myPanel);  
    }  
}
```

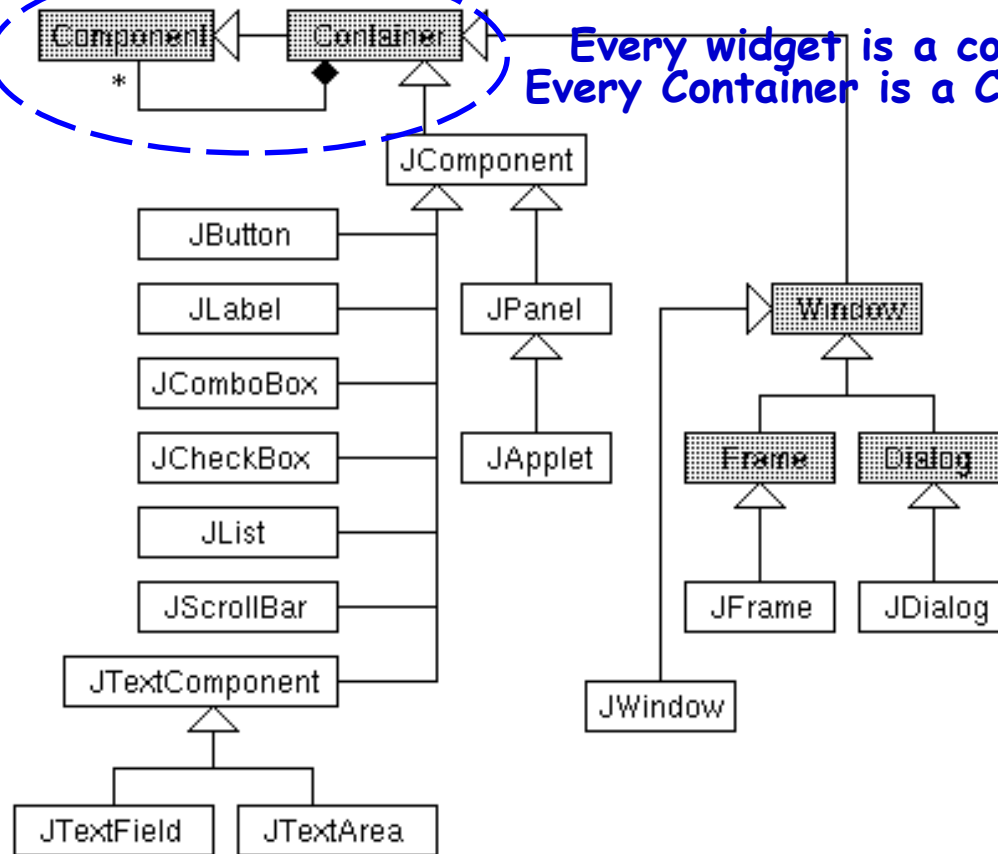
Solution:
Composing GUI

AWT Components



Swing Components

Every widget is a container!
Every Container is a Component!



Adapter Pattern



Intent:

- **Convert the interface of a class to the interface expected by the users of the class.**
- Allows classes to work together even when they have incompatible interfaces.



Example (non-software):

- You went to U.S.
- Had an Indian electrical appliance?
- How can you use it in U.S.?
- **Use Adapters!**



Adapter Pattern



Also universal adapters?

- A **wrapper pattern**

- **Problem: Convert the interface of a class into one that a client expects.**

- Lets classes work together --- that couldn't otherwise --- because of incompatible interfaces
 - Used to provide a new interface to existing legacy components.

- Two main adapter variants:

- **Class adapter:**

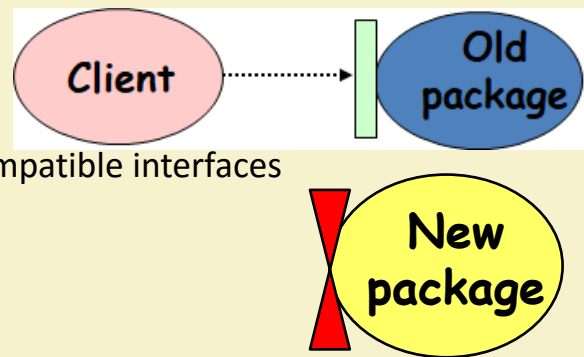
- Uses interface implementation and inheritance mechanisms

- **Object adapter:**

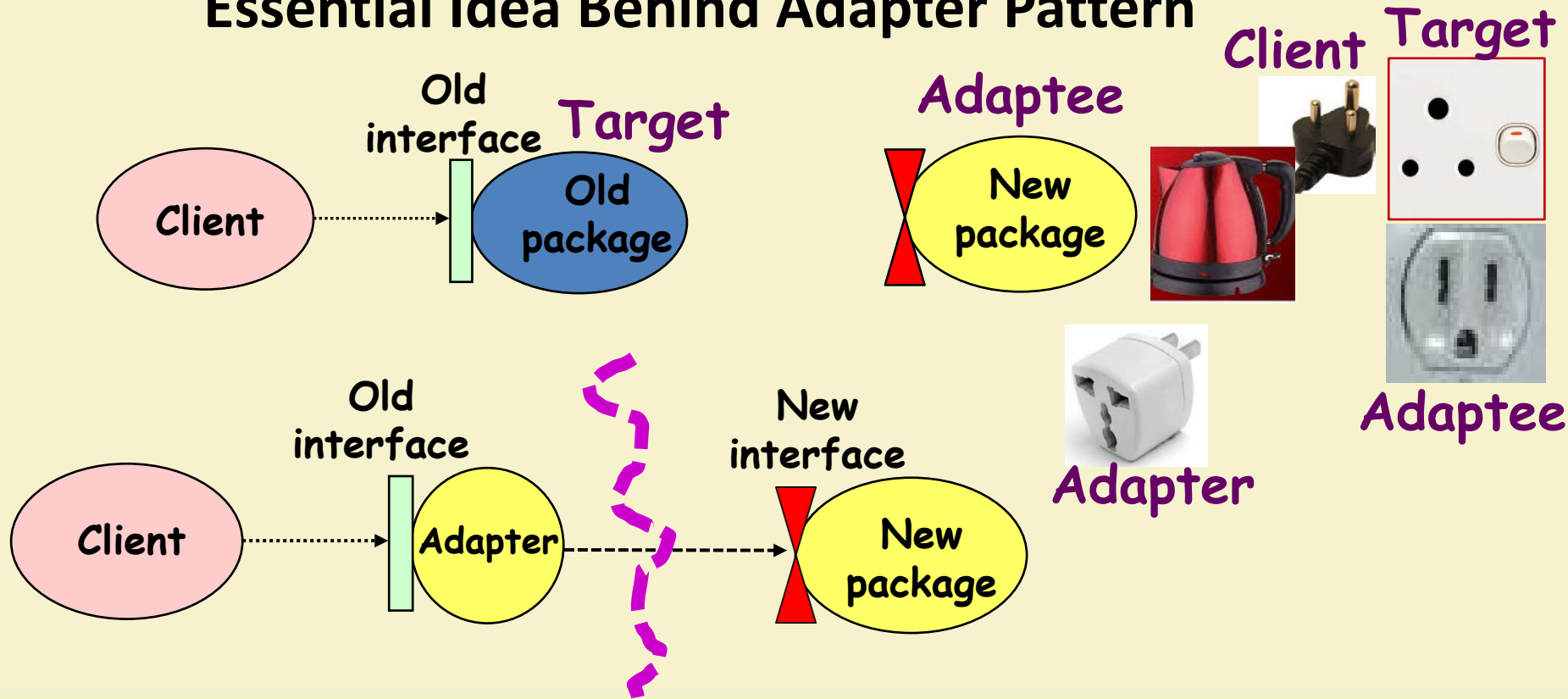
- Uses delegation to adapt one interface to another

- **Object adapters are much more common.**

Adapter Pattern



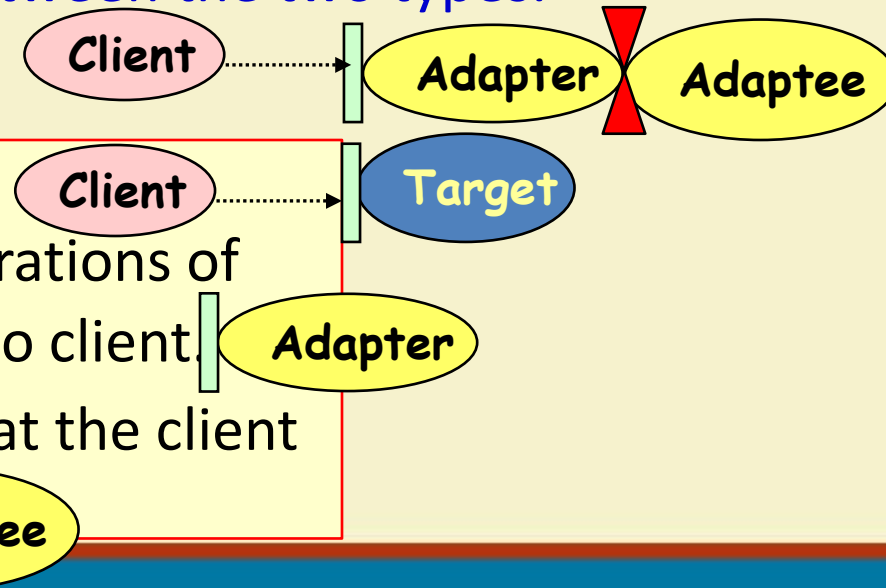
Essential Idea Behind Adapter Pattern



Adapter Pattern

- Helps two incompatible types to communicate.
 - When a client class expects an interface ---but that is not supported by a server class,
 - The adapter acts as a translator between the two types.
- 3 essential classes involved:

- **Target** – Interface that client uses.
- **Adapter** - class that wraps the operations of the Adaptee in interfaces familiar to client
- **Adaptee** - class with operations that the client class desires to use.



Recap: Terminology

Target



Adapter



Adaptee

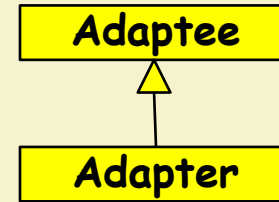


Class and Object Adapters

An adaptee may be given a new interface by an adapter in two ways:

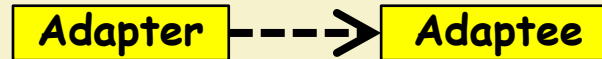
- **Inheritance**

- Known as **Class Adapter pattern**
- The adapter is a sub-class of adaptee;



- **Delegation**

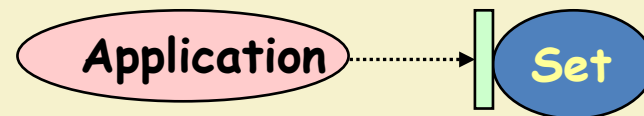
- Known as **Object Adapter pattern**
- The adapter holds a reference to an adaptee object and delegates work to it.



Example 1 – Sets

- There are many ways to implement a set
- Assume:
 - Your existing set implementation has poor performance.

- You got hold of a more efficient set class,
 - BUT: The new set has a different interface.
 - Do not want to change voluminous client code

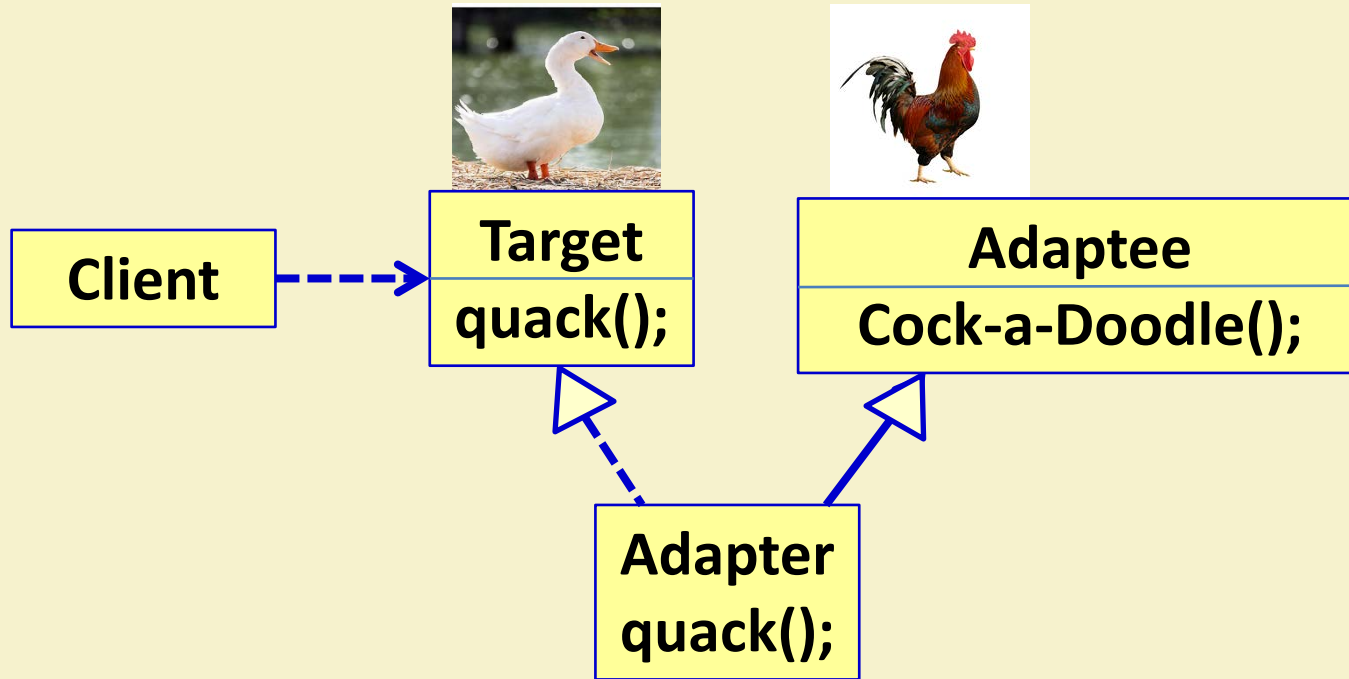


- **Solution: Design a setAdapter class :**

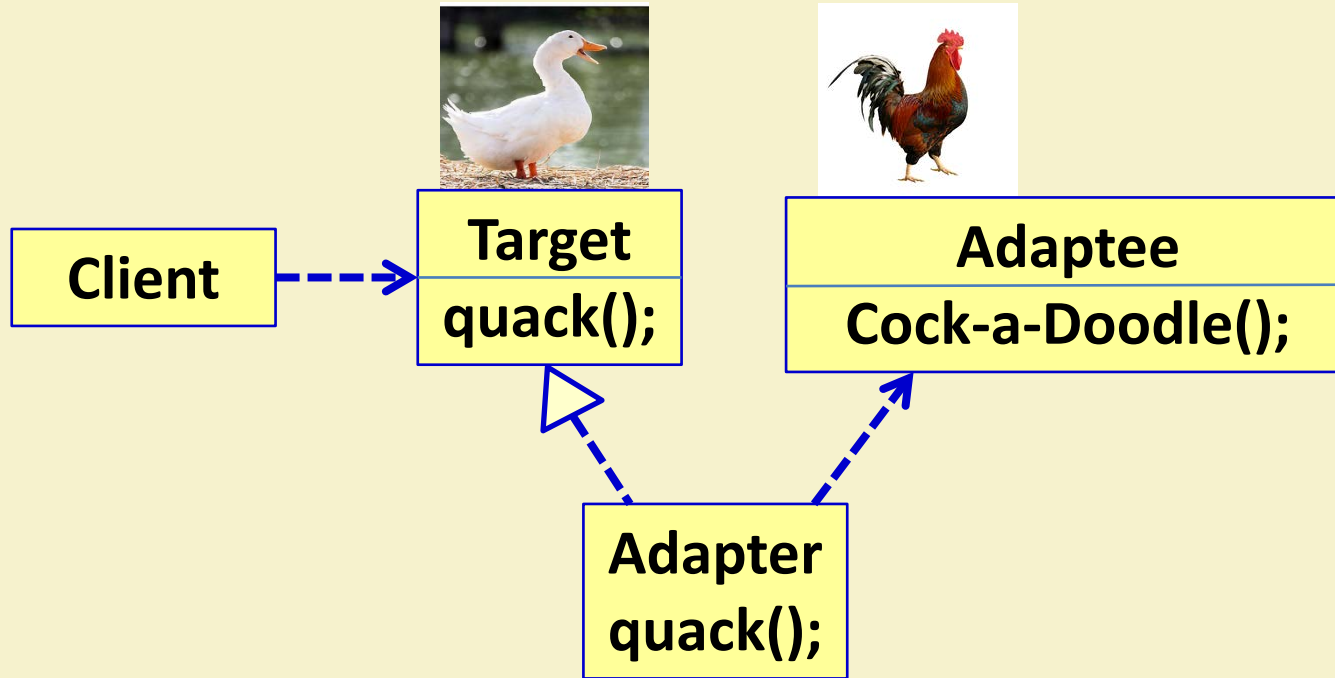
- Same interface as the existing set..
- Simply translates to the new set's interface.



Class Adapter: Main Idea



Object Adapter: Main Idea



Example: Problem

Client



OldSet

```
add(Object e)
del(Object e)
int cardinality()
contains(Object e)
```

Existing

Got hold of Newset...

Target

NewSet

```
insert(Object e)
remove(Object e)
int size()
contains(Object e)
```

Want use this with client...

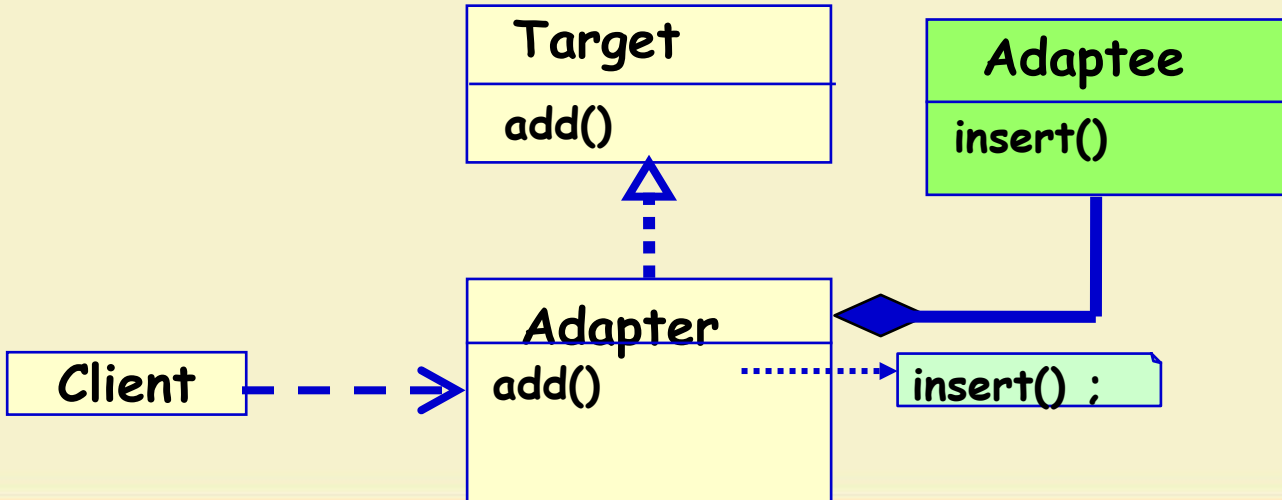
But, do not want
to change Client
code...

Adaptee

Object Adapter --- main idea delegation

- Adapter internally holds an instance of the Adaptee
- Uses it to call Adaptee operations from within operations supported by the Target.

Object Adapter Pattern



Object Adapter - Code

Client Code:

```
Adaptee a = new Adaptee(); Target t = new Adapter(a);  
public void test() { t.add(); }
```

Target Code:

```
interface Target {  
    public void add(){}  
}
```

Adaptee Code:

```
class Adaptee {  
    public void insert(){}  
}
```

Adapter Code:

```
class Adapter implements Target {  
    private Adaptee adaptee;  
    public Adapter(Adaptee a) { adaptee = a;}  
    public void add() { adaptee.insert();}  
}
```