Remaining Part of Singleton Pattern and Then State and Composite Patterns

Lect 21
10-10-2023

Singleton: Example Code

```
Public Class Singleton {
  private static Singleton uniqueInstance = null;

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

Exercise 1

- Objects in a certain application:
  - Get a Database Manager: `DBMgr.getDBMgr();`
- It needs to be ensured that:
  - There is only one Database Manager object.
  - 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
    publicstatic 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 not just use a static method?

- Why have all the complexity of Singleton?
  - **Why is a static method with private constructor not enough?**

```
public class DBMgr{
private static DBMgr d= DBMgr();
public static DBMgr getDBMgr(){
return d;}  }
```

- **Problem 1: lacks flexibility**
  - 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.

Exercise 1: Random Number Generator

```
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 Possible problem?**
  - Always creates the instance, even if it isn't used…

Singleton Pattern: Some 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.

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

Singleton Pattern: Some Insights

- **Singleton subclassing:** Singleton needs to implement a protected constructor:
  - Clients cannot directly instantiate Singleton instances through new.
  - But, protected constructor can be called by subclasses.

```
public abstract class MazeFactory {
        private static MazeFactory
                        uniqueInstance = null;
    protected MazeFactory(){}
```

Can Multiple Singletons Still Appear?

- In certain situations, two or more Singletons 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;
  }
```

Handling Concurrency Issue

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

      return  instance;
  }
```

Refined code: without unnec

```
public class RandomGenerat
  private static RandomGe

  public static RandomGene                    ) {
    if (gen == null) {
      synchronized (RandomGener.or.class) {
        // must test again -- can you see why?
        if (gen == null)
          gen = new RandomGenerator(); }
    }
    return gen;
  }
}
```

synchronized (RGen.class) is used here to make sure that there is exactly one Thread in the block..

Singleton Pattern: Variations

- **Multiton:**
  - We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
  - **Operation that grants access to the Singleton instance needs to change.**

```
final class  Multiton{
    private static final int COUNT = 10; // change to vary the number of instances
    private static final Multiton[] INSTANCES = new Multiton[COUNT];
    private static int index = 0;
    static {
        for (int i = 0; i < INSTANCES.length; i++)  {
            INSTANCES[i] = new Multiton();
        }
    }
      private Multiton() {}
      public synchronized Multiton getInstance(){// synchronized because of index
      Multiton result = INSTANCES[index];
      index = (index + 1) % INSTANCES.length;
      return result;
    }
}
```

Simple Multiton

Usage of Singleton in Other Patterns

- Several patterns use Singleton Pattern for implementation.
  - For instance AbstractFactory needs a singleton object for maintaining a single instance of the factory.

State Pattern

Introduction

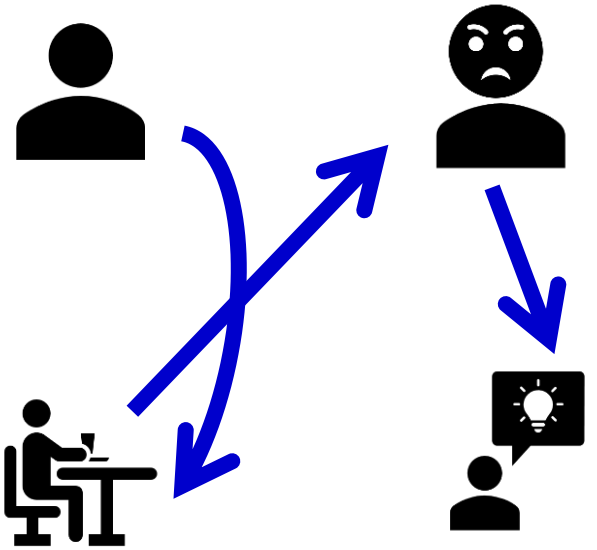
- An object is behaving differently to the same message. What could be the reason?
  - **Possibly the object is transiting to different states.**

- **Example:** Consider the responses to the "renew" request on a book:

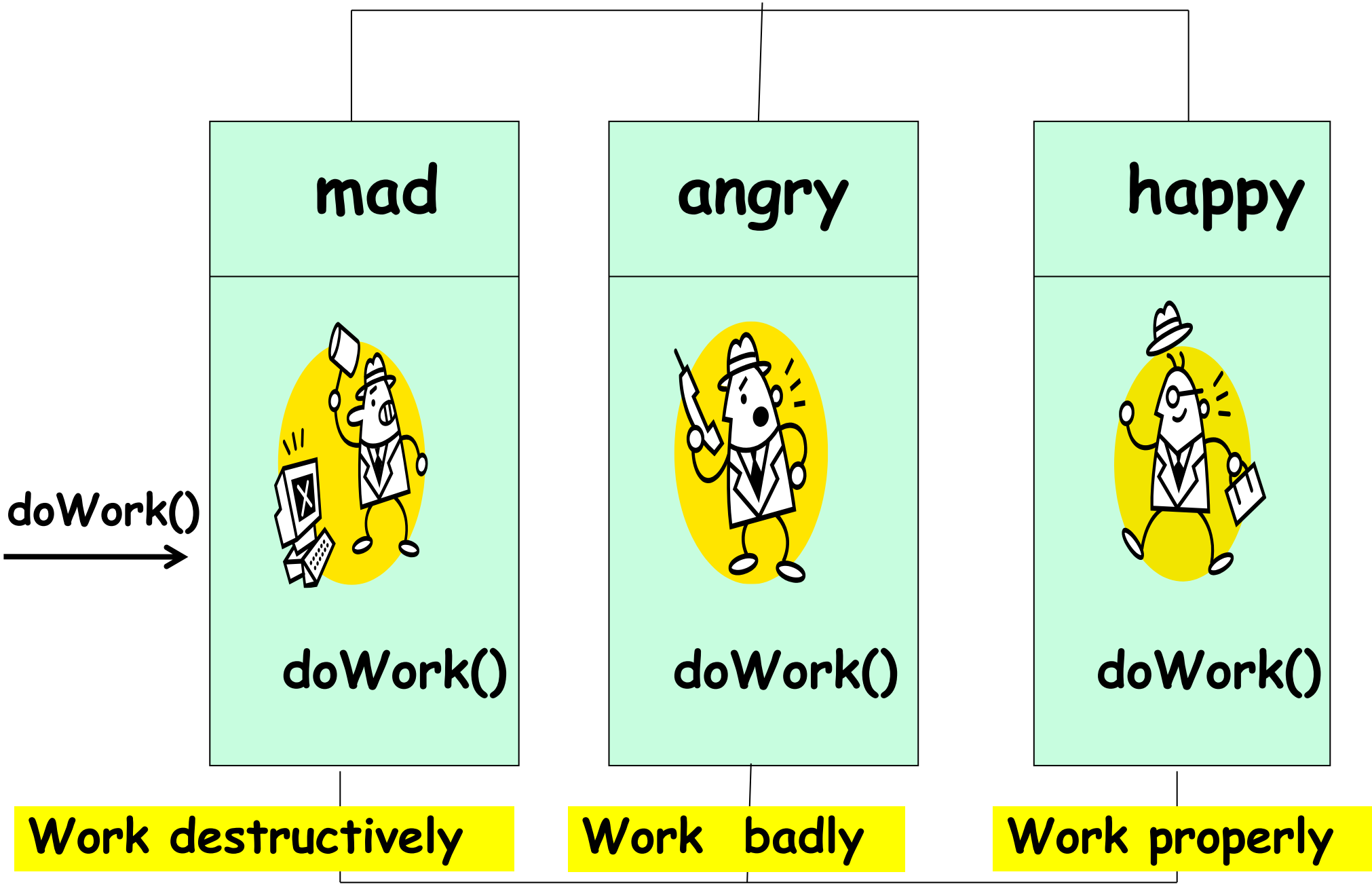
  - Successfully renewed
  - Reserved, cannot be renewed
  - Book needed for stock taking, cannot be renewed
  - Already renewed five times, cannot be renewed

State of an Object

- One or more attributes of a class acts as state variables.
- 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 Example: Person

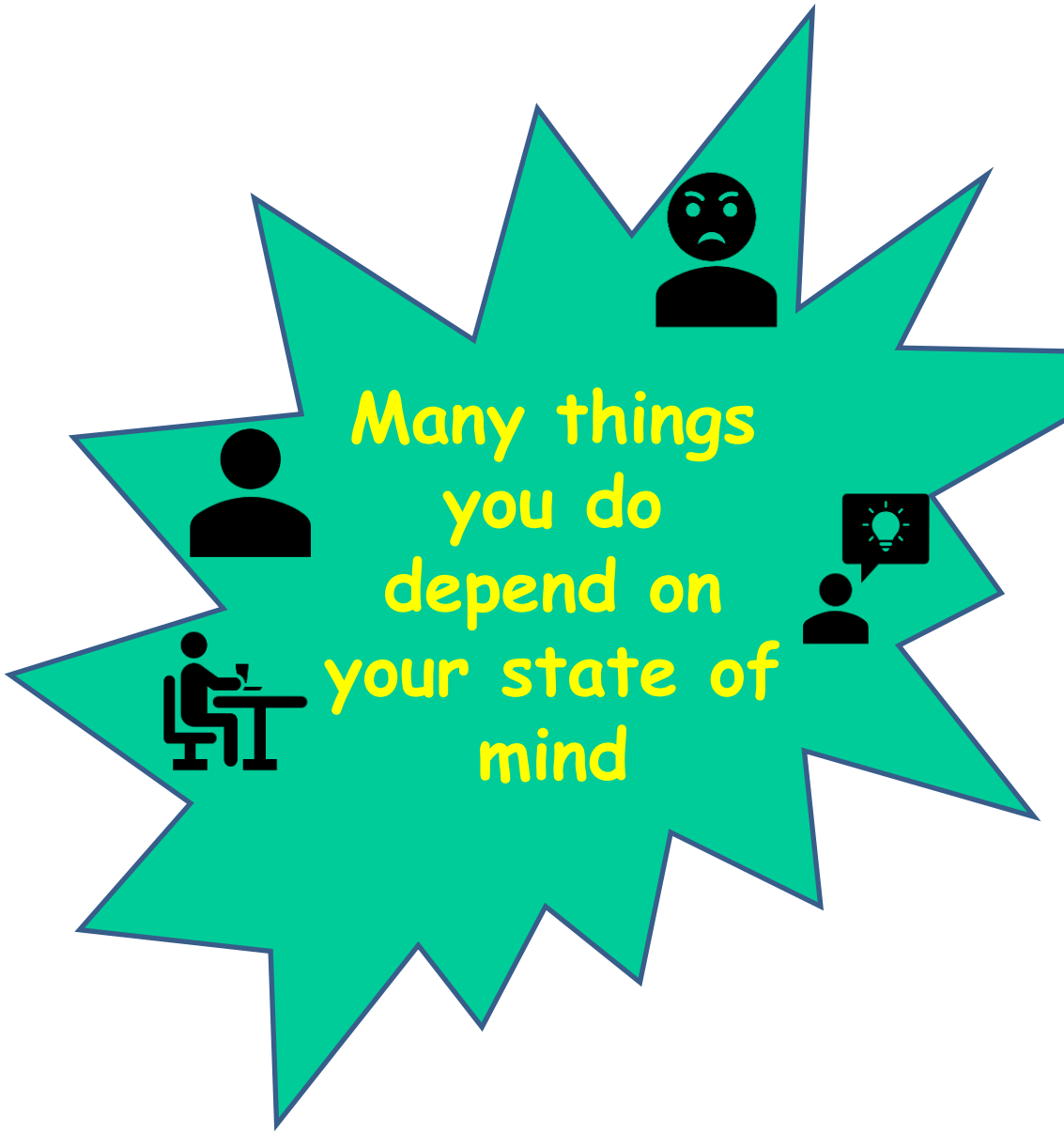


State Pattern: Overview

- 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.**

Typical State Behavior Implementation

```
Class Person {

  Mood  state;

     doWork() {
       if (state == angry)

          doWorkBadly();
       else if (state == happy)
          doWorkProperly();
     }

     pleaseHelpMe() {
          if (state == angry)

            doScold();
        else if (state == happy)
            doHelp();
     }

      ….
}
```



State Design Pattern

- **Idea:** Value of an internal state var determines Object behavior.

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

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.

State Pattern Basic Approach:
Replace Conditionals by Inheritance

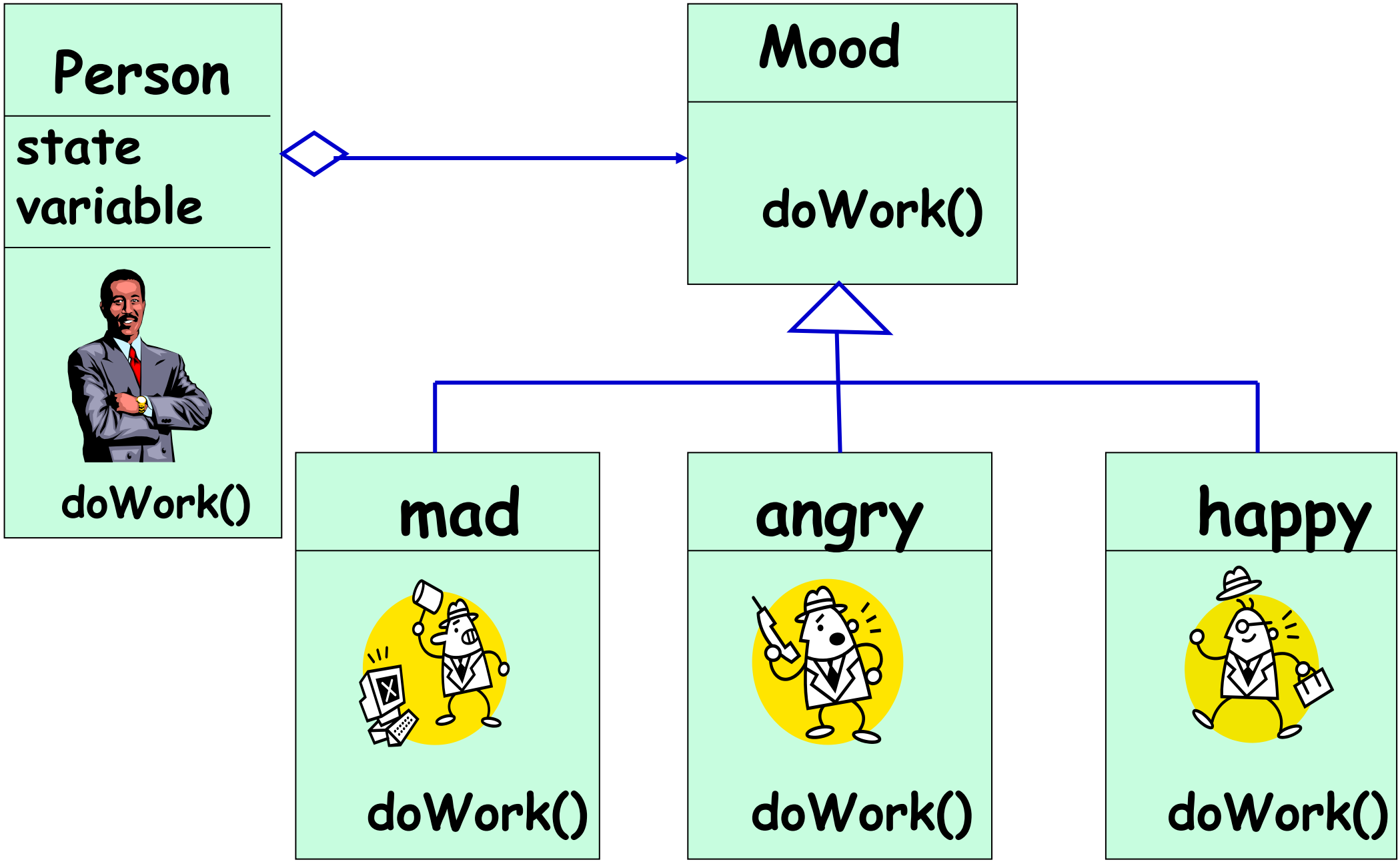
```
doWork() {
    If (condition)
        doWorkProperly()
    else
        doWorkBadly();
}
```

```
abstract class A{
    doWork();
}
```

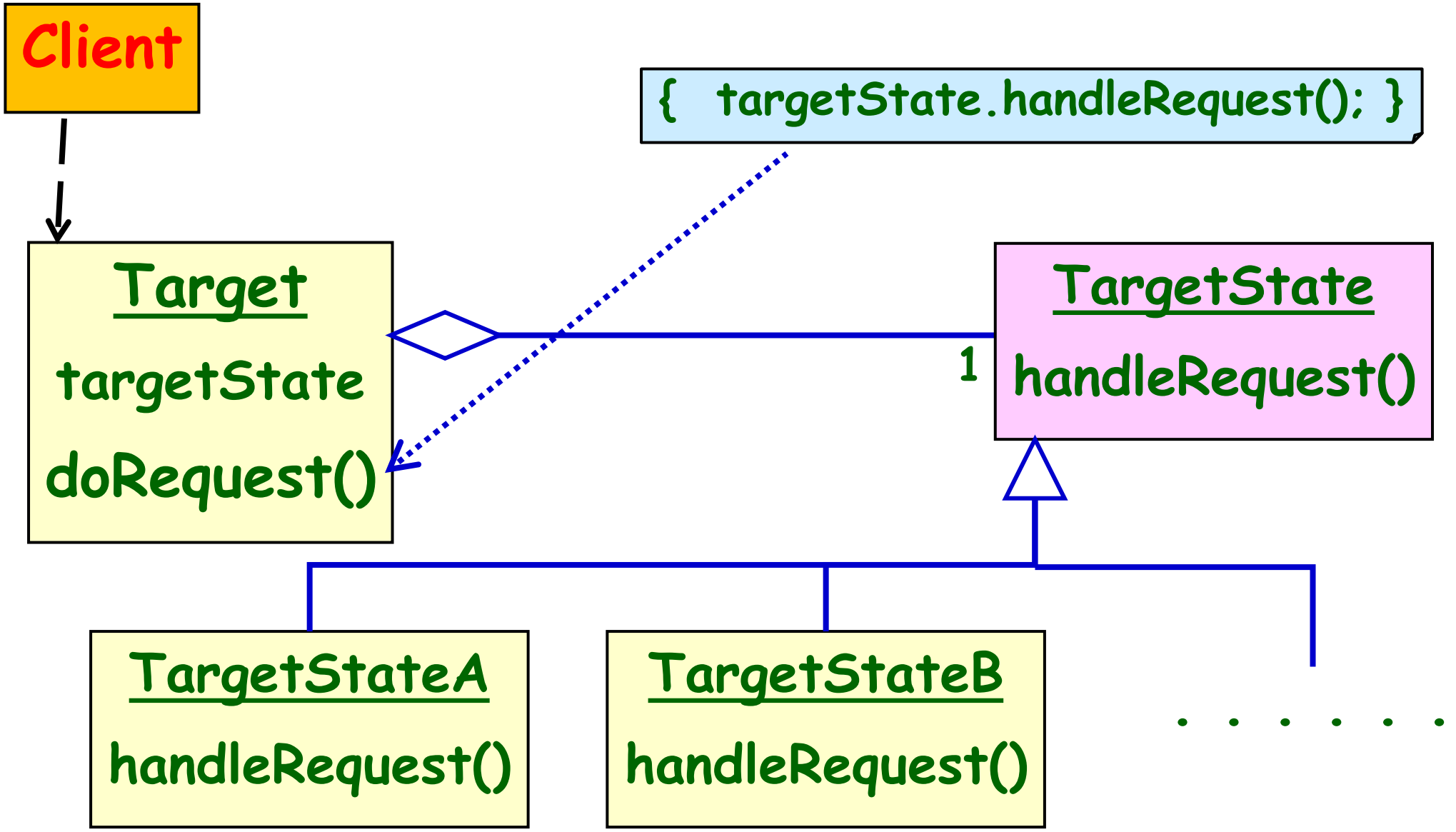
```
****IfCondition****
  doWork(){
doWorkProperly();}
```

```
****ElseCondition****
    doWork(){
        doWorkBadly();}
```

Example: State Pattern Structure

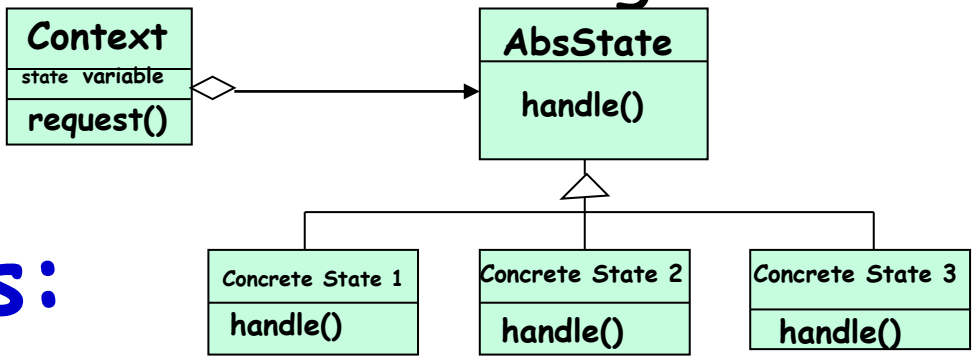


State Pattern: Working



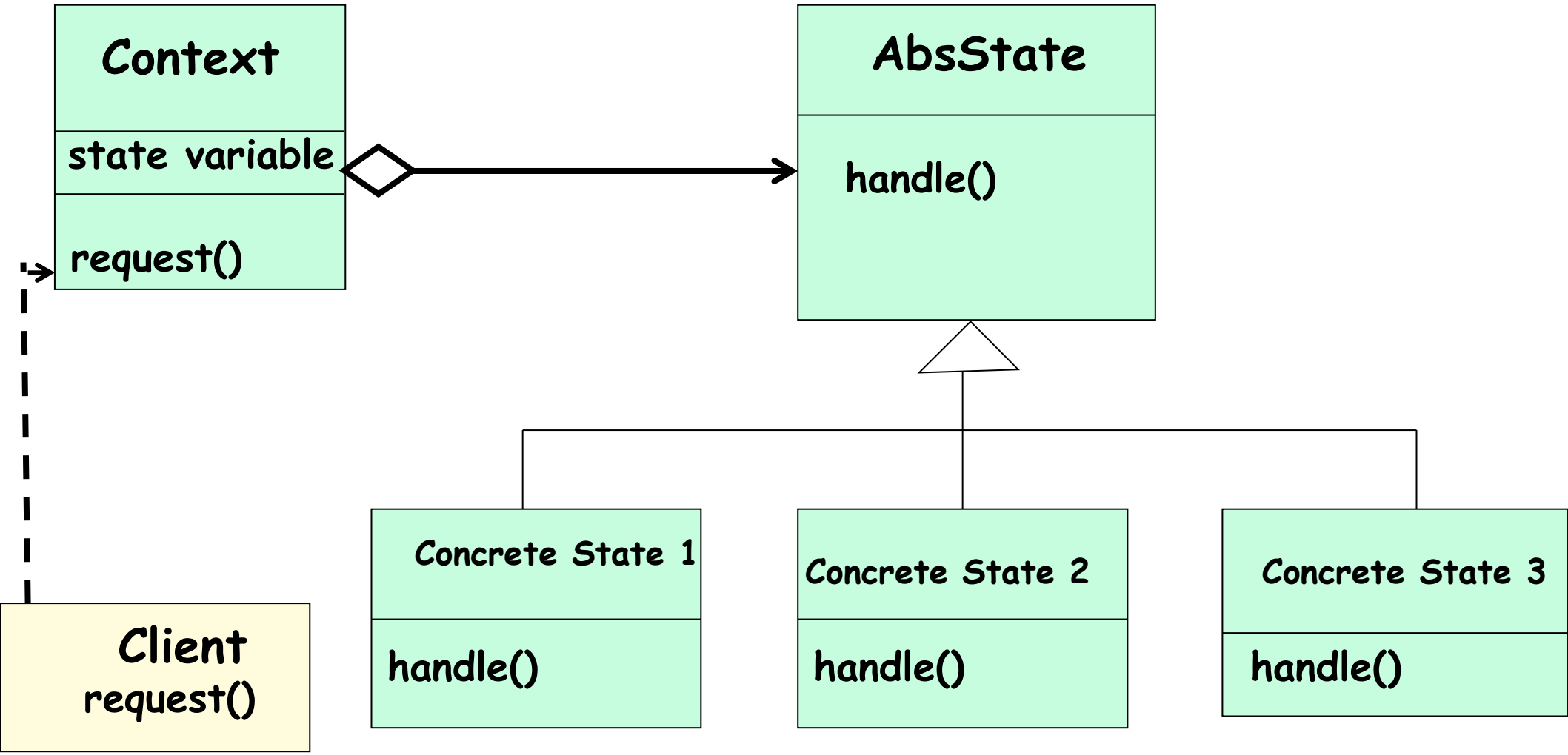
State Pattern: Players

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



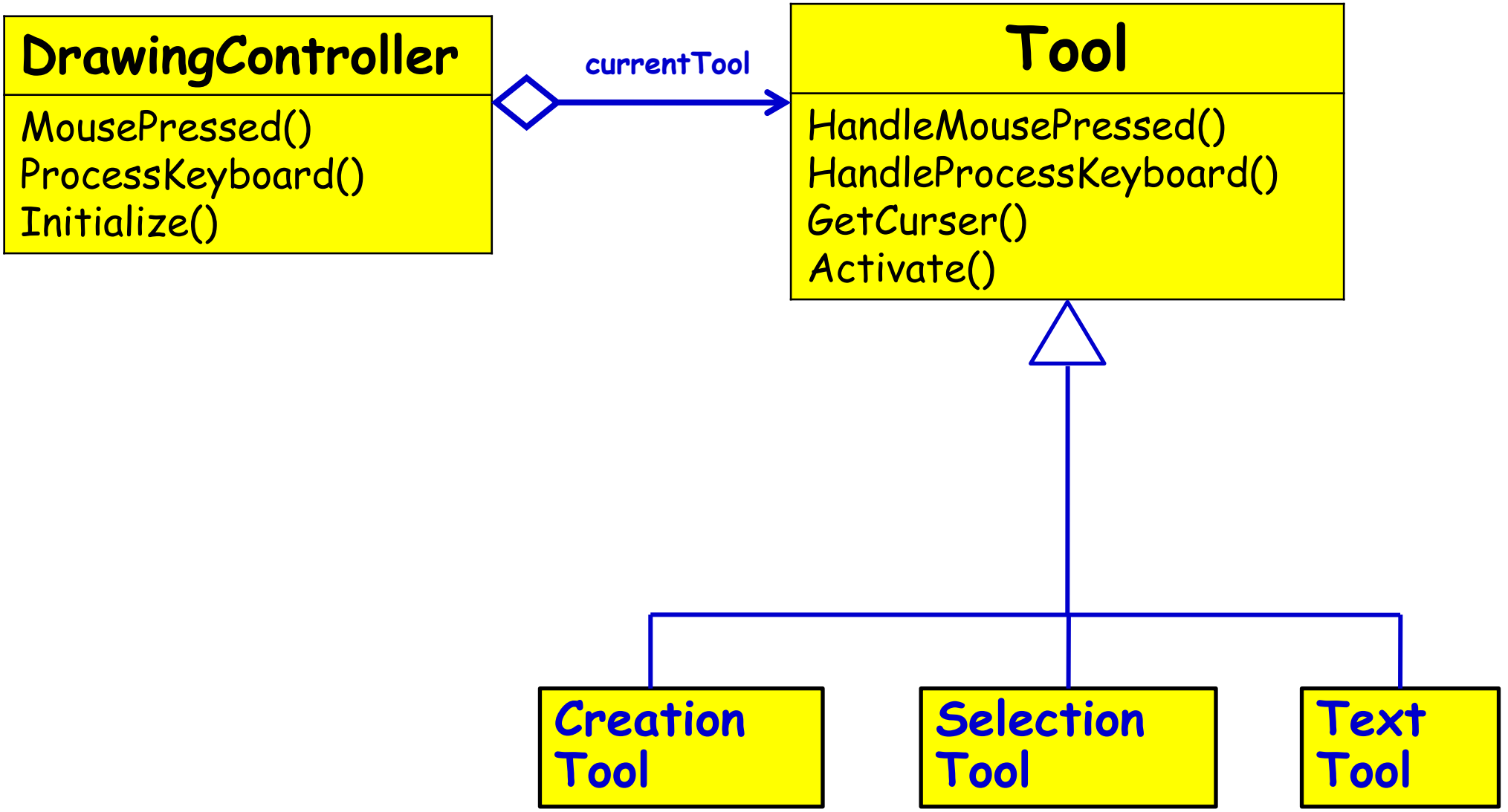
- **Abstract State class**
- **Concrete State classes:**
  - Define the changed behavior in states.

State Pattern Structure

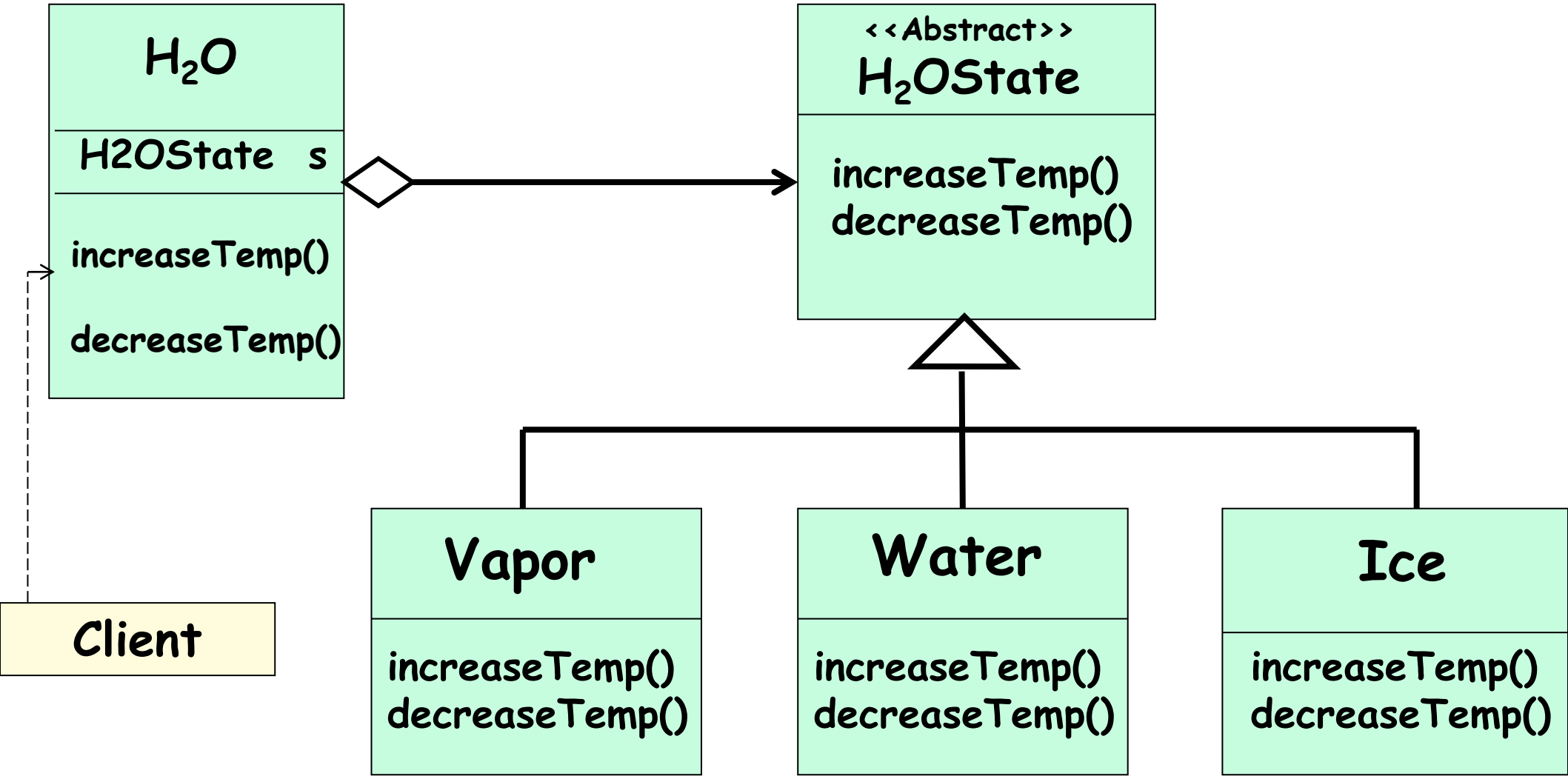


Allow an object to alter its behavior (dynamically bind to a different method) when its internal state changes.

Example 1: Based on the current tool selection, the behavior of mouse press, key board press, etc. vary…

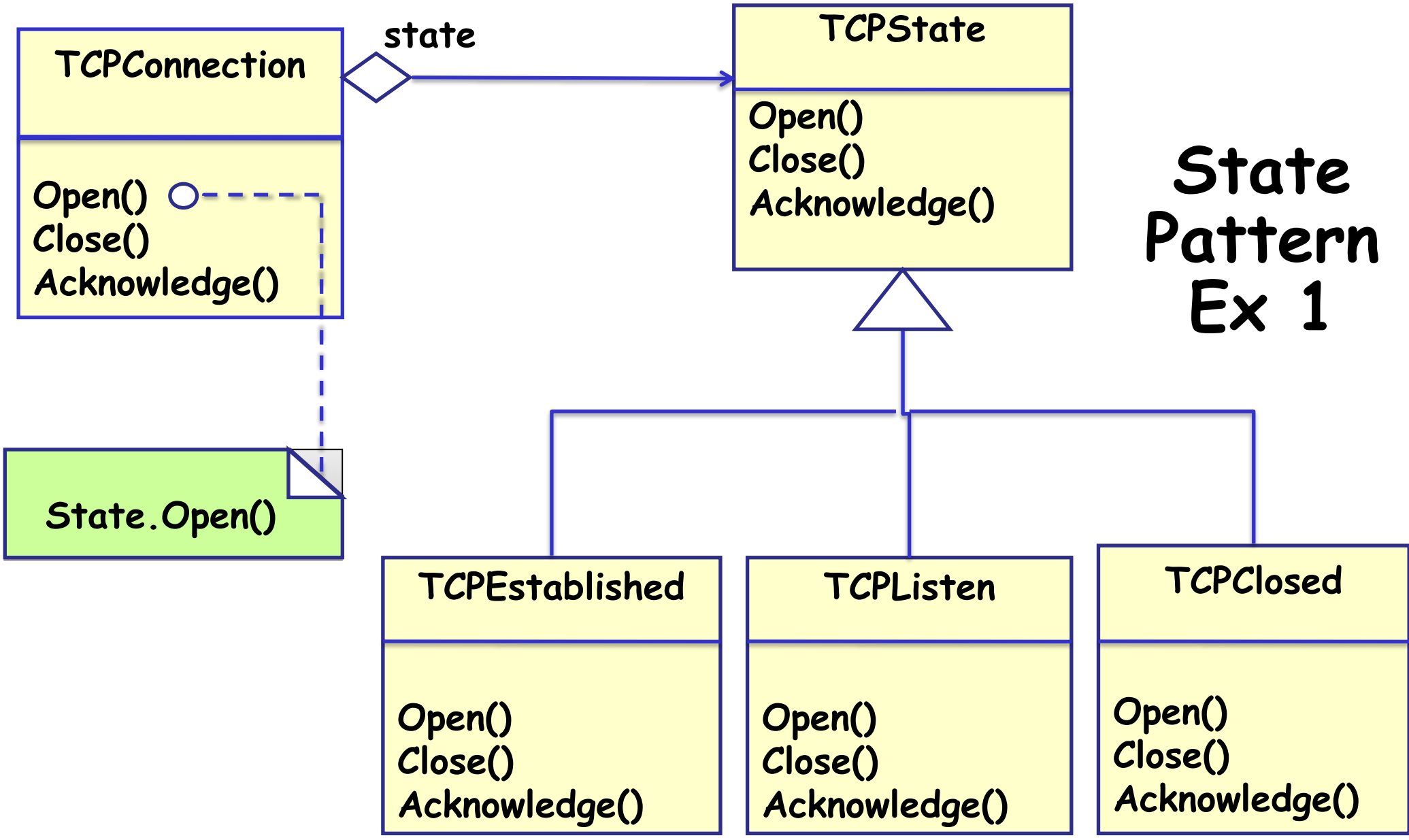


Example 2: Behavior of H_2O Depends on Its State



Exercise 1

- A TCP connection may be in any of the following states:
  - Connection established
  - Listen
  - Close
- Methods open, close, and ack, behave differently in these states



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

Trade-offs

Advantages

- Encapsulates behavior of a state into an object
- Eliminates massive 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.

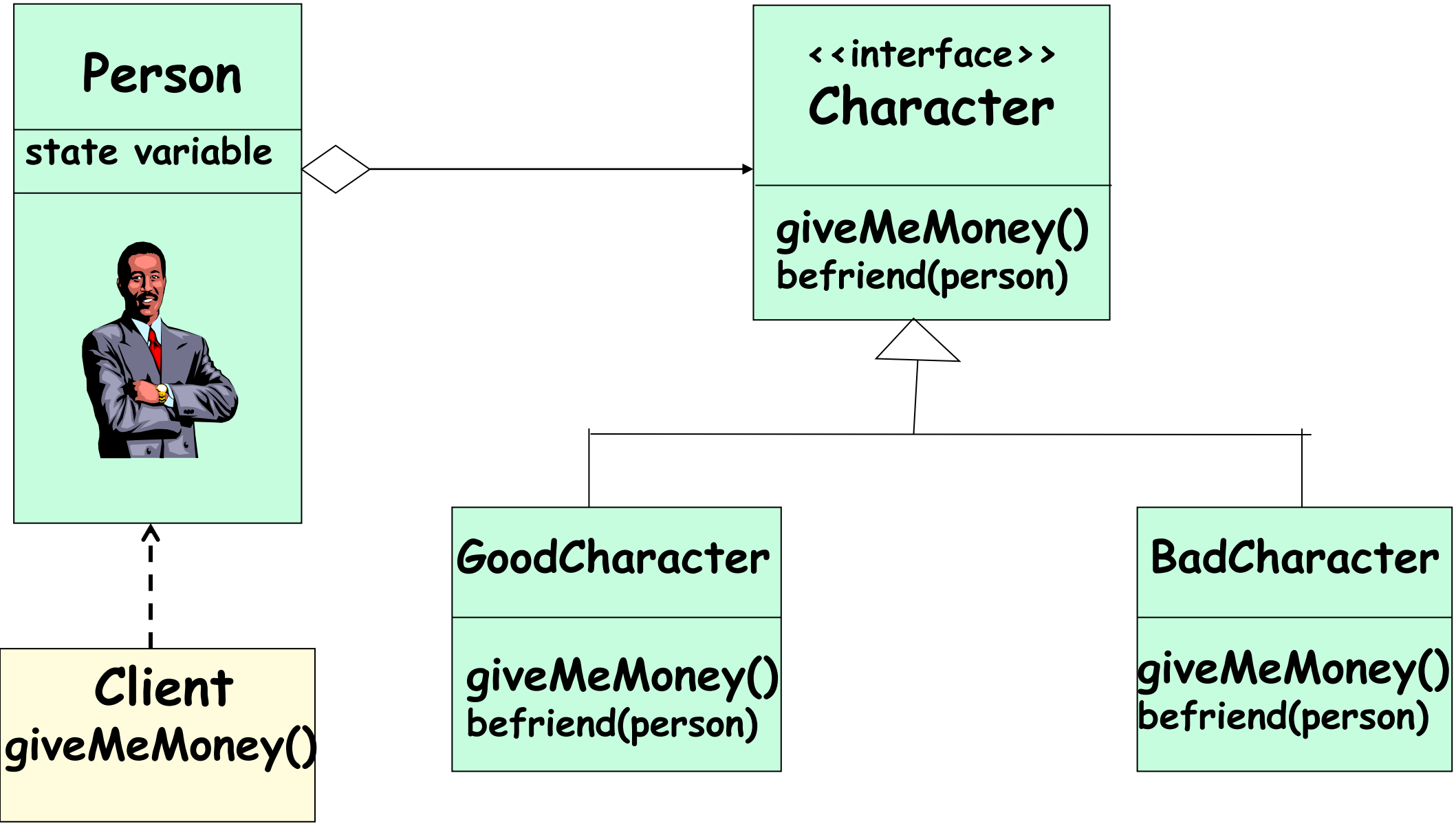
State Pattern

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

Exercise 3

- Draw the class diagram and write sample code to implement the following behavior of a class using state pattern.
- A person's character may be good, bad, or ordinary:
  - **When asked to give money, a good person gives all his money, a bad person says he does not have any money, an ordinary person says he would give money later. A good person becomes bad, if he befriends a bad person. A bad person becomes a good, if he befriends a good person.**

Exercise- Class Diagram



Example Java Code [Without Pattern]

```
public class Person {

private String name;

private String character;

public Person(String name, String character){

this.name = name; this.character = character; }

public void giveMeMoney(){

 if (character.equals("GoodCharacter")){
        System.out.println("Yes, take all my money"); }

else if (character.equals("BadCharacter")){
   System.out.println("No, I don't have anything with me"); }

else if (character.equals("OtherCharacter")){
   System.out.println("I will give the money tomorrow"); }

 } }
```

Example Java Code [With Pattern]

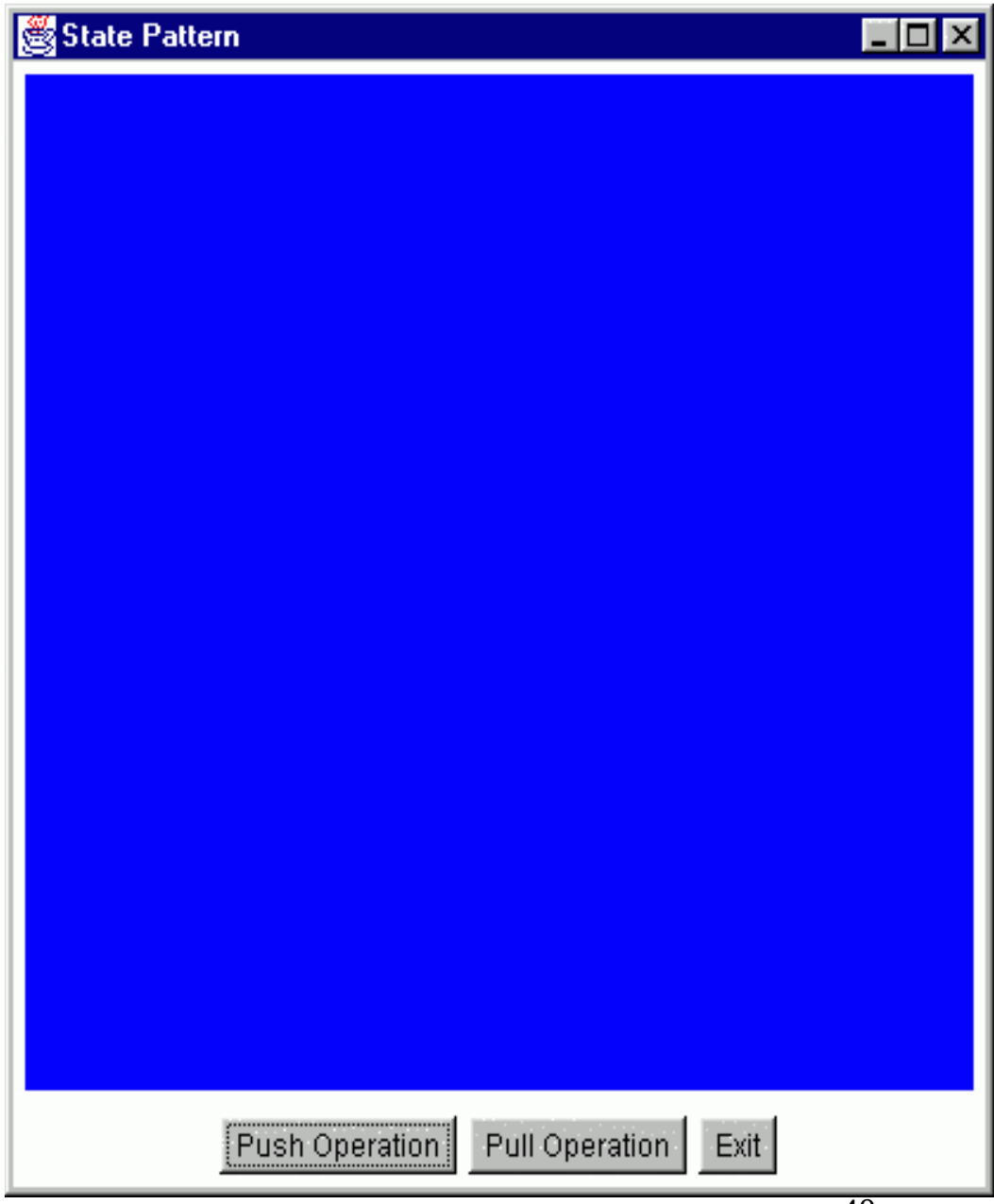
```
public interface Character {
public void giveMeMoney(); }

public class GoodCharacter implements Character {
public void giveMeMoney() { System.out.println("Yes,
  take all my money"); } }

public class BadCharacter implements Character{
  public void giveMeMoney() { System.out.println("No,
  I dont have anything with me"); } }
```

Example Java Code [With Pattern]

```
public class PersonPatternSol{

private String name;

private Character character;

public PersonPatternSol(String name, Character
  character){

this.name = name;  this.character = character; }

public void giveMeMoney(){ character.giveMeMoney(); }

public static void main(String[] args) {

PersonPatternSol object = new Person2("John", new
  GoodCharacter()); object.giveMeMoney(); object = new
  PersonPatternSol("John", new BadCharacter());
  object.giveMeMoney(); } }
```

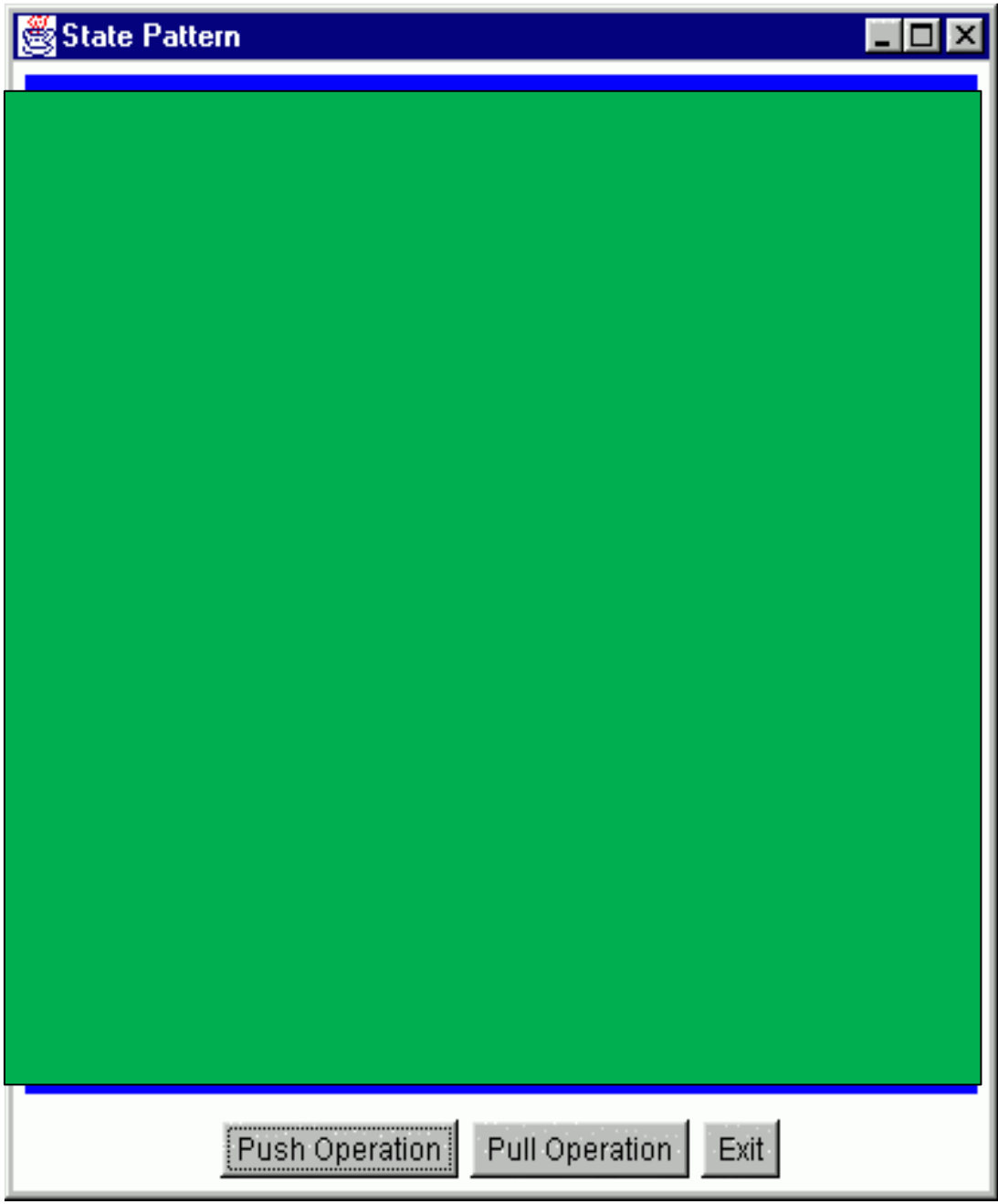
State Exercise 4: Color Changing Toy

- Toy has 4 states:
  - blue, black, green and red.
- Two functions:
  - push() and pull() changes the current state.
- **Blue state:**
  - push -> green state, pull -> red state
- Black state:
  - push -> red state, pull -> black state
- Green state:
  - push -> black state, pull -> blue state
- Red state:
  - push -> blue state, pull -> green state



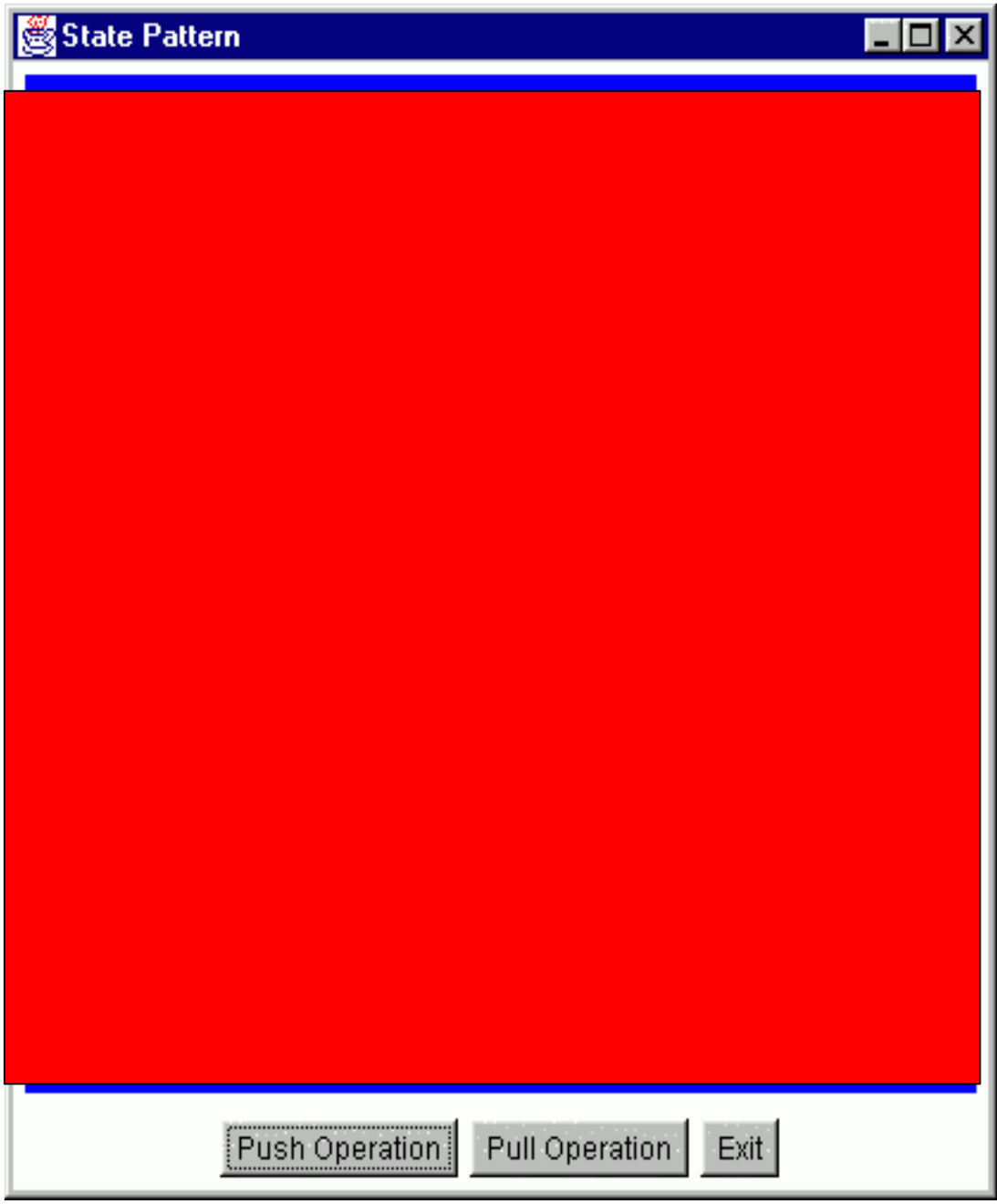
State Exercise 4: Color Change Toy

- We have 4 states:
  - blue, black, green and red.
- Two functions:
  - push() and pull() changes the current state.
- **Blue state:**
  - **push -> green state**, pull -> red state
- Black state:
  - push -> red state, pull -> black state
- Green state:
  - push -> black state, pull -> blue state
- Red state:
  - push -> blue state, pull -> green state

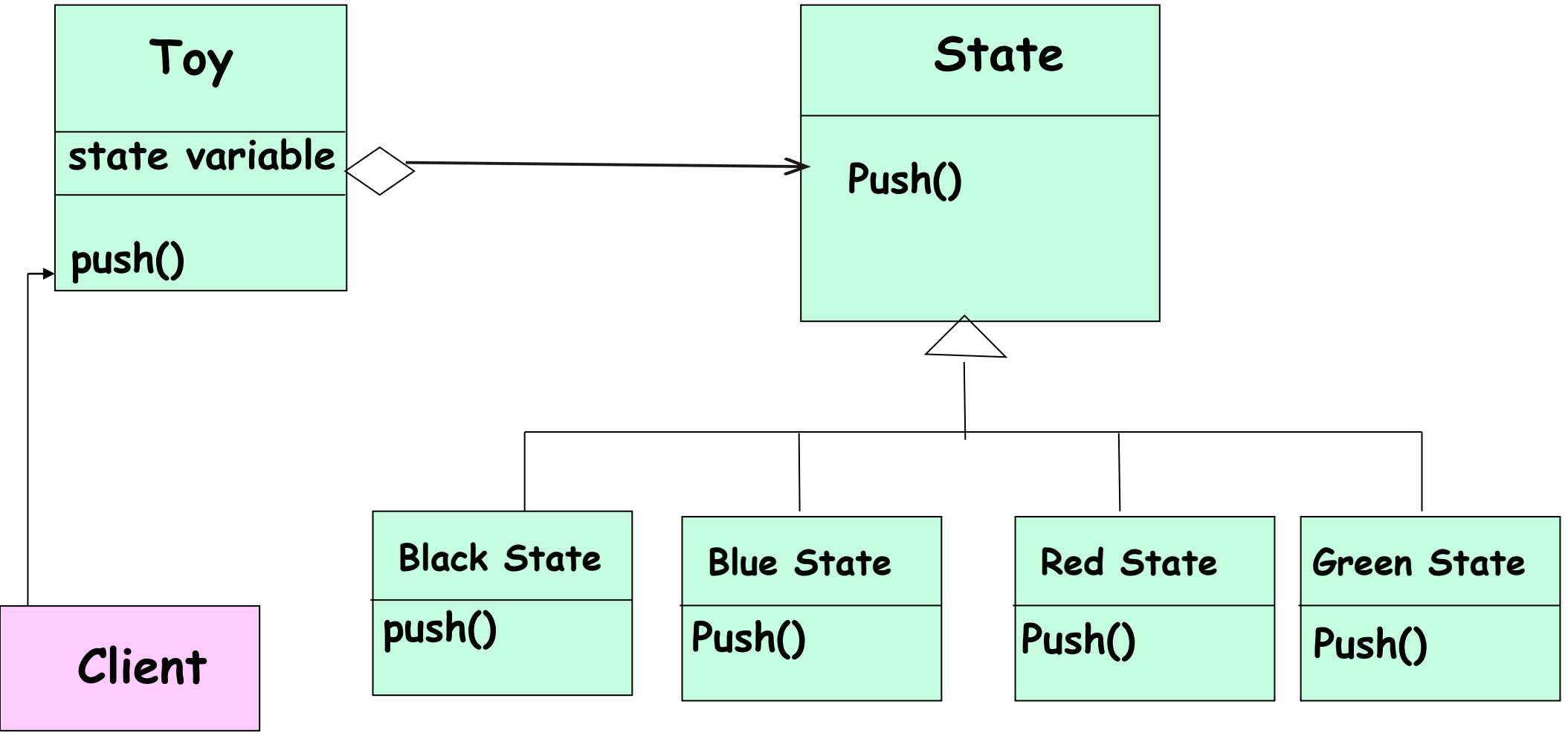


State Exercise 4: Color Change Toy

- We have 4 states:
  - blue, black, green and red.
- Two functions:
  - push() and pull() changes the current state.
- **Blue state:**
  - push -> green state, **pull -> red state**
- Black state:
  - push -> red state, pull -> black state
- Green state:
  - push -> black state, pull -> blue state
- Red state:
  - push -> blue state, pull -> green state



Exercise 4: Solution



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

Exercise 4: Solution

```
public abstract class State {
        public abstract void handlePush(Context c);
        public abstract void handlePull(Context c);
        public abstract Color getColor();
    }
public class BlackState extends State {
      public void handlePush(Context c) {
          c.setState(new RedState());}
      public void handlePull(Context c) {
           c.setState(new GreenState());}
      public Color getColor() {return (Color.black);}
    }
```

State Example Cont…

```
public class Context {
    private State state = null;  // State attribute
    public Context(State state) {this.state = state;}
    public Context() {this(new BlueState());}
    public State getState() {return state;}
    public void setState(State state) {
                                this.state = state;}
    public void push() {state.Push(this);}
}
```

Pros of State Pattern

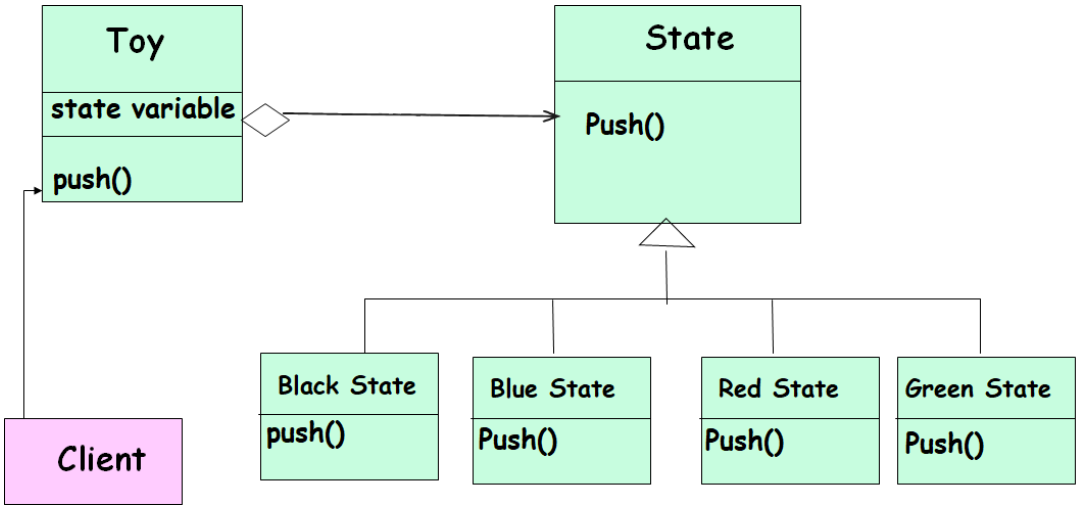
- **Localizes state-specific behavior:**
  - Partitions behavior for different states into separate objects.
- **Makes state transitions explicit**
  - Obvious when moving from one state to another: as appropriate state class instance must be swapped in

Question

- Who creates the states that are taken on transitions?

•PreferablyIncorporated into the ConcreteState classes

•In the Context Class for simple cases



When are ConcreteState Objects Created?

- Two options:
  - **Created as and when they are needed.**
  - **Create once, used whenever needed:**
    Context class uses it when needed.

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.

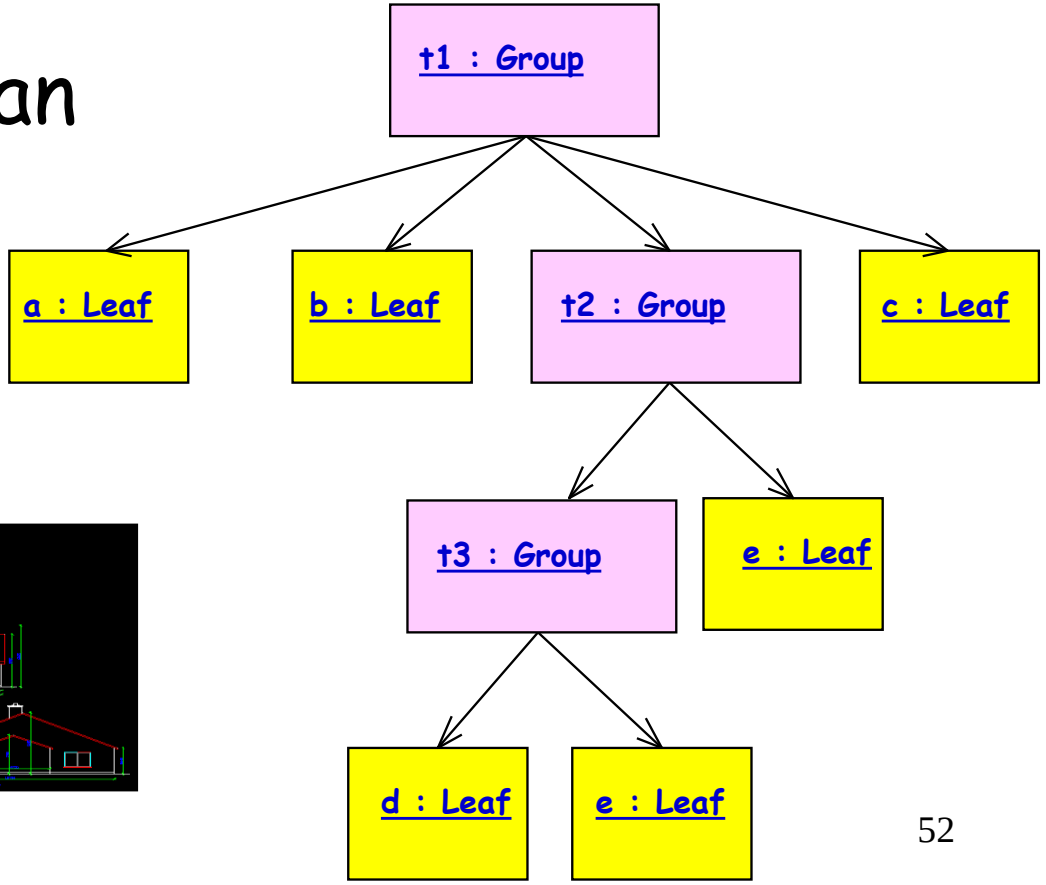
Known Uses and Related Patterns

- Several interactive drawing programs use the State pattern.
- The Flyweight pattern makes it clear:
  - When and how State objects can be shared.
- State Objects are often Singletons.

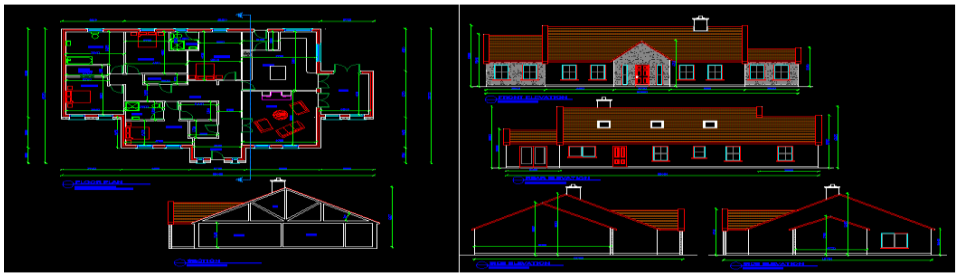
Composite Pattern

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.

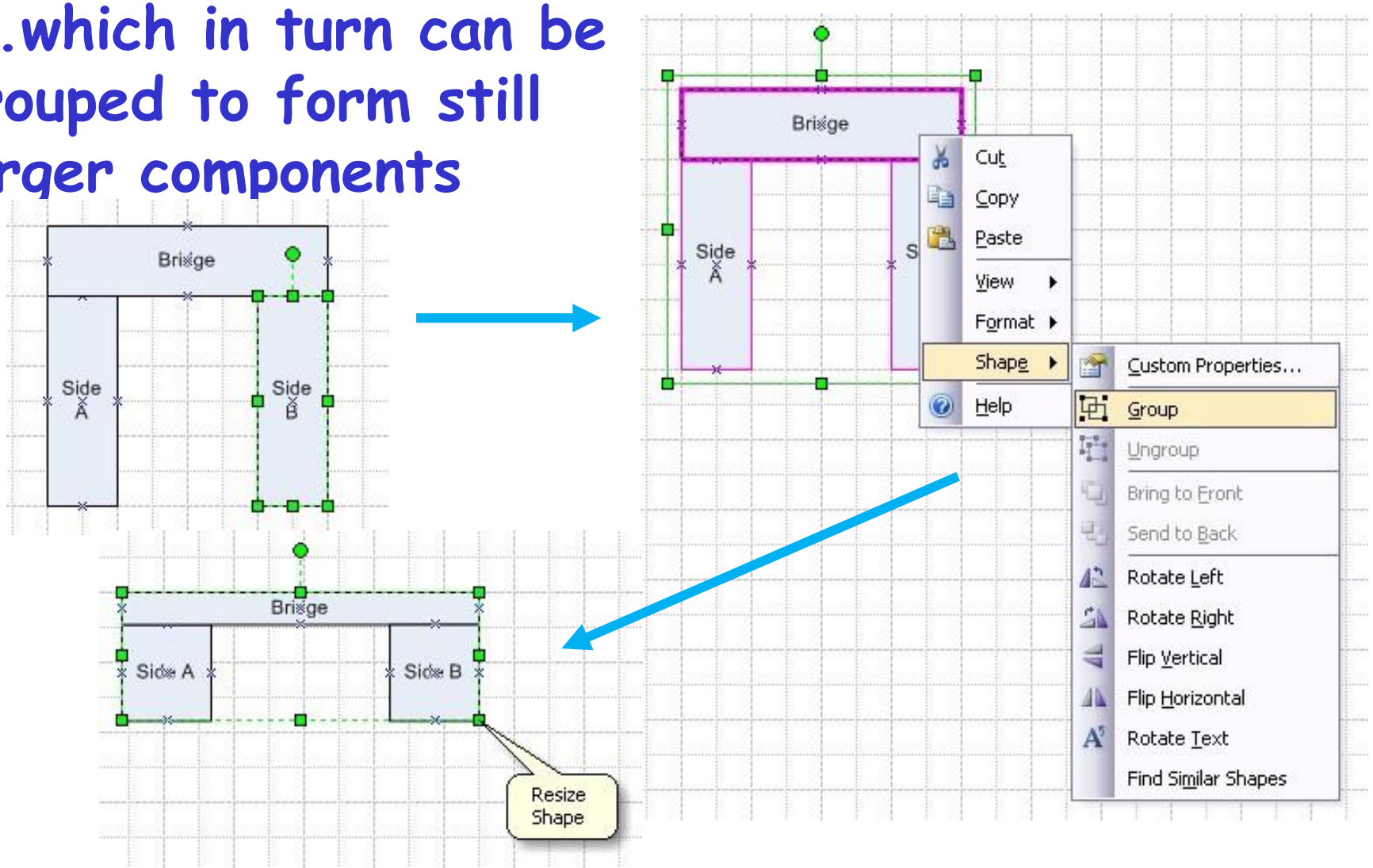


- **Example 1:A drawing---**



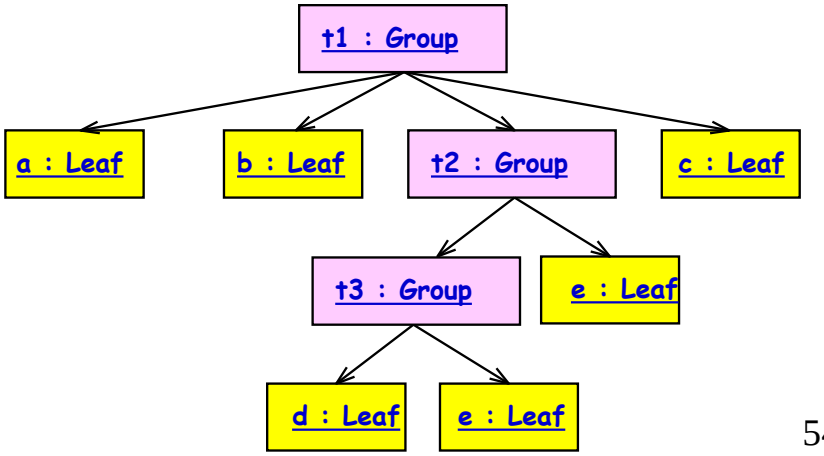
Example: Consider a Graphics Editor…

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

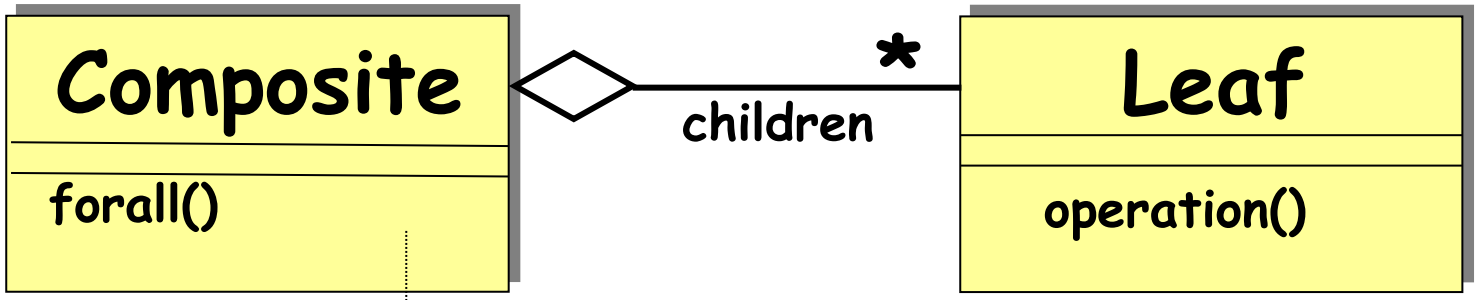


Why Composite Pattern? (Motivation)

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

Composite Solution: Attempt 1

Problem: handling nested groups of objects



For all children c:
c.operation()

- Lets see a designer

Mark=1/100

solution from scratch…

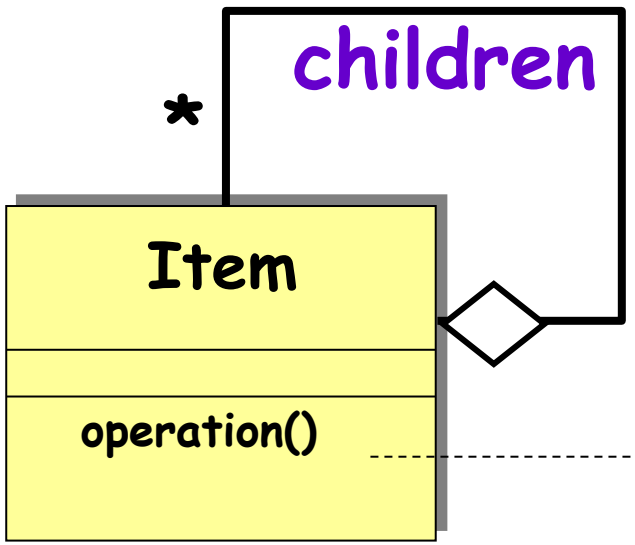
```
…
if X is
Composite
then
    X.forall()
else
X.operation();
…
```

Analysis of Solution: Naïve solution…

What are the problems?

- Only single nesting level (depth =1)
- Composite and leafs always treated differently in code
- Difficult to extend with new kinds of leafs or composites.

Composite: Attempt 2




```
if there are children then
    for all children c: c.operation()
else doSomething();
```

Client:

```
item.operation()
```

Mark= 40/100

Surely there are improvements…

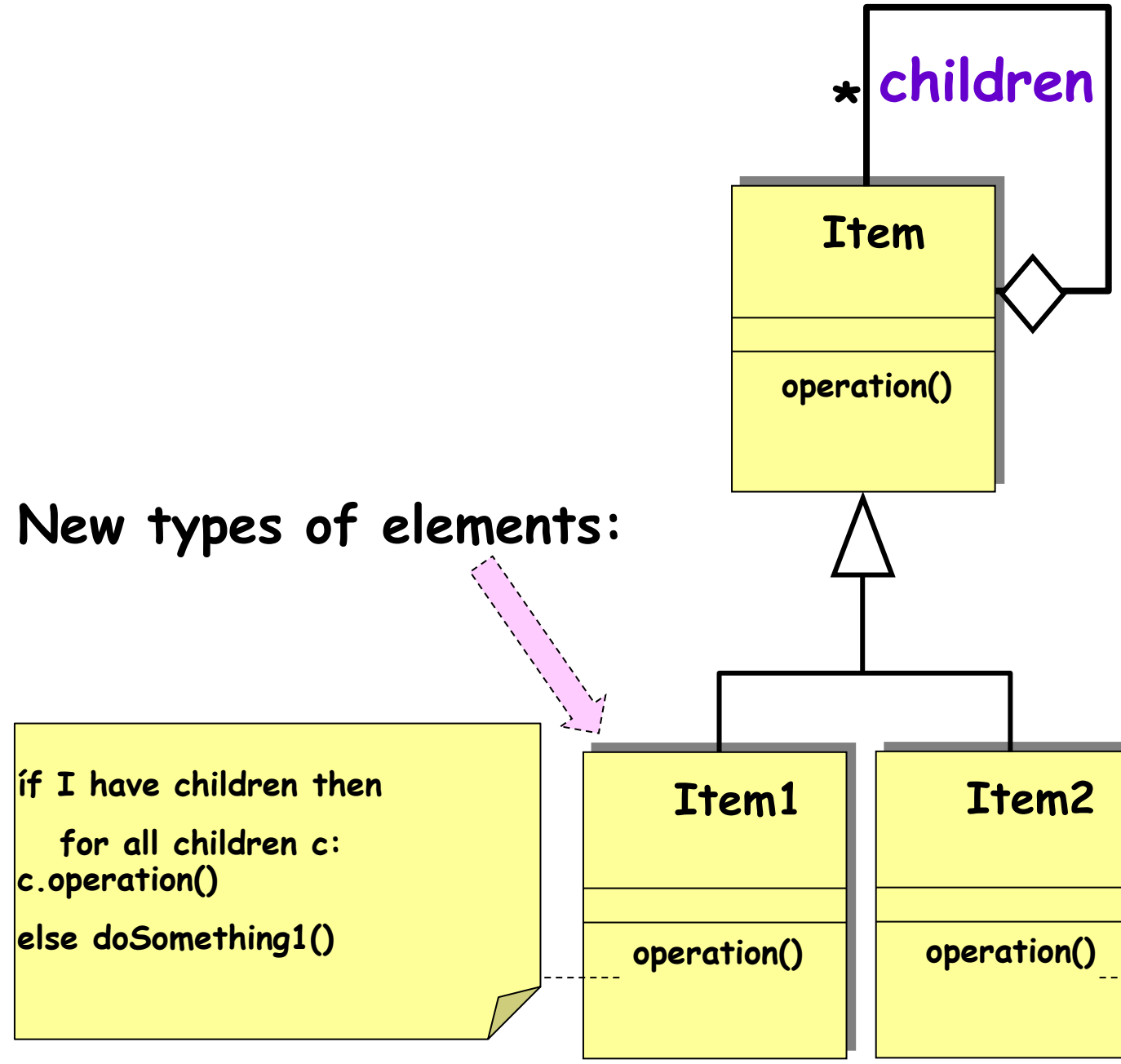
Unified treatment in client **and** unrestricted depth of parts…

What are the problems?

-Does not handle different item types

-Difficult to extend with new kinds of leafs or composites.

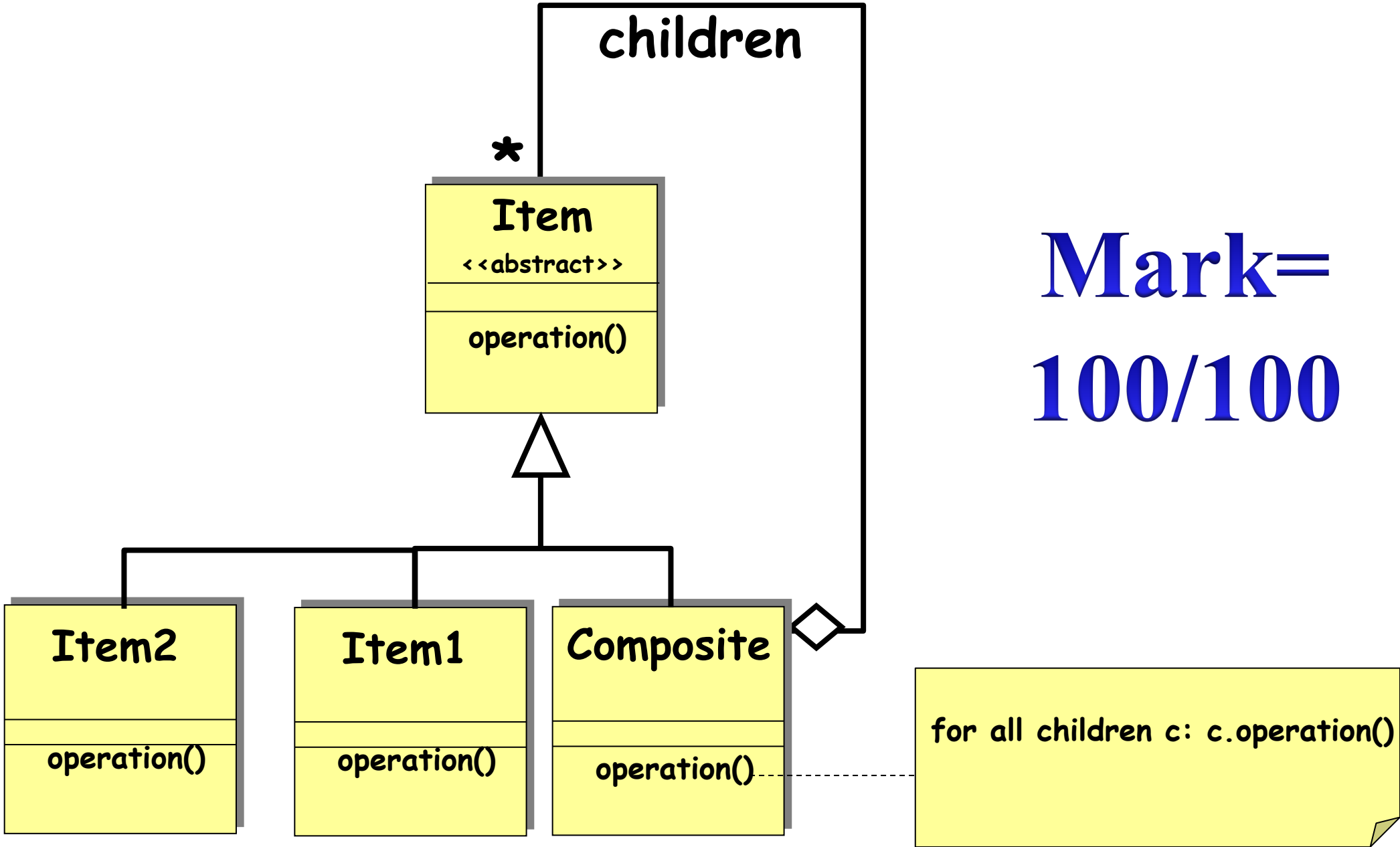
Attempt 3: Handling different Items



Mark= 50/100

Finally: Composite Pattern

Separating the composite class:

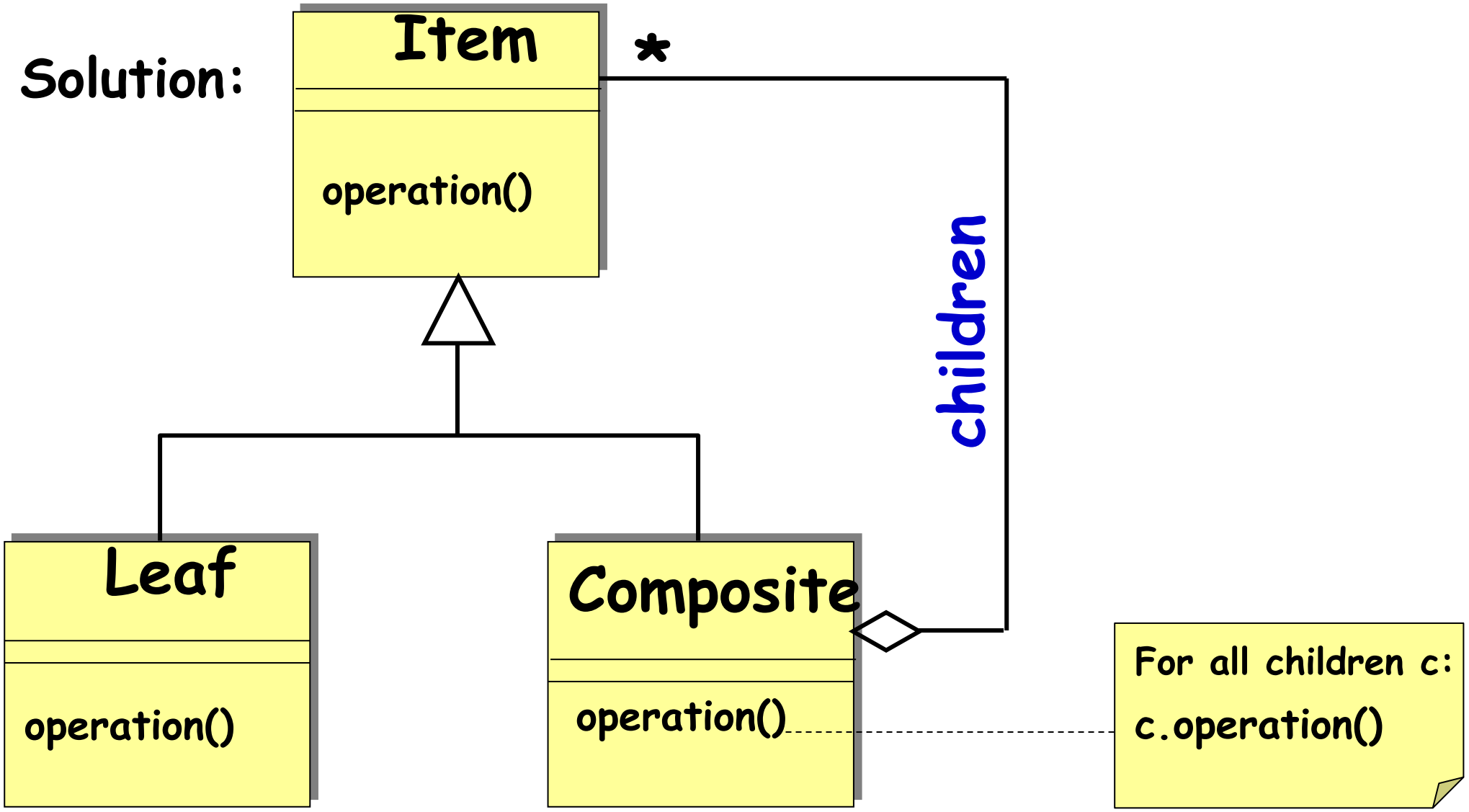


Mark= 100/100

Composite Design Pattern

Problem: How to organize hierarchical object structures so that the clients are not aware of the hierarchy?

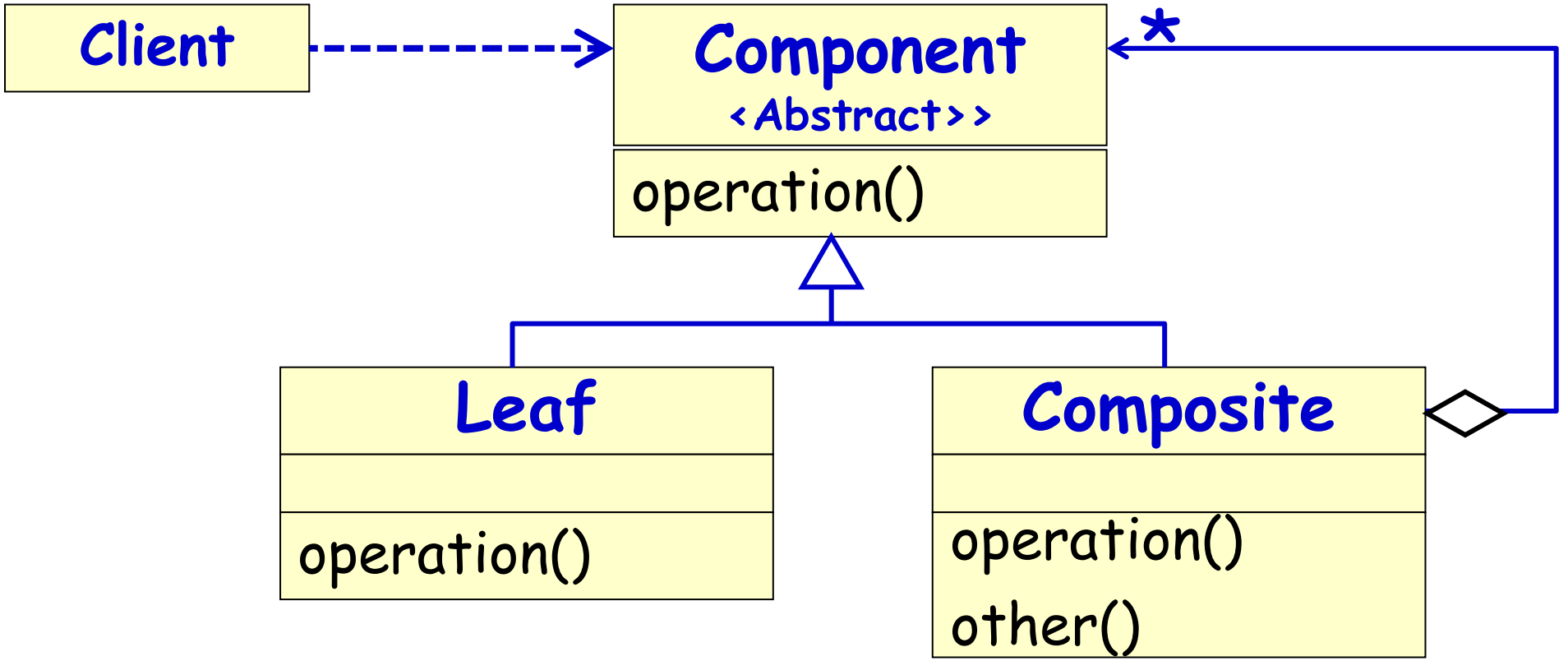
Solution:



Composite Pattern: Issues

- What is the class diagram?
- Who is the client?
- What operations are defined for:
  - The component, the composite, and the leaf?
  - How are they carried out?
- How are associations implemented in the code?

Composite Pattern: Class Diagram

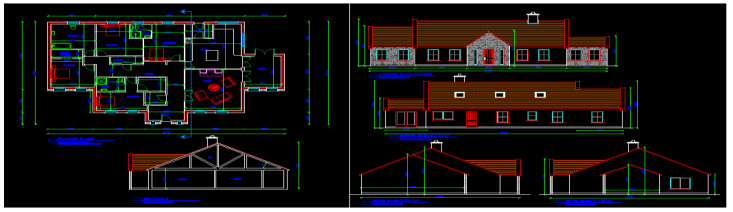


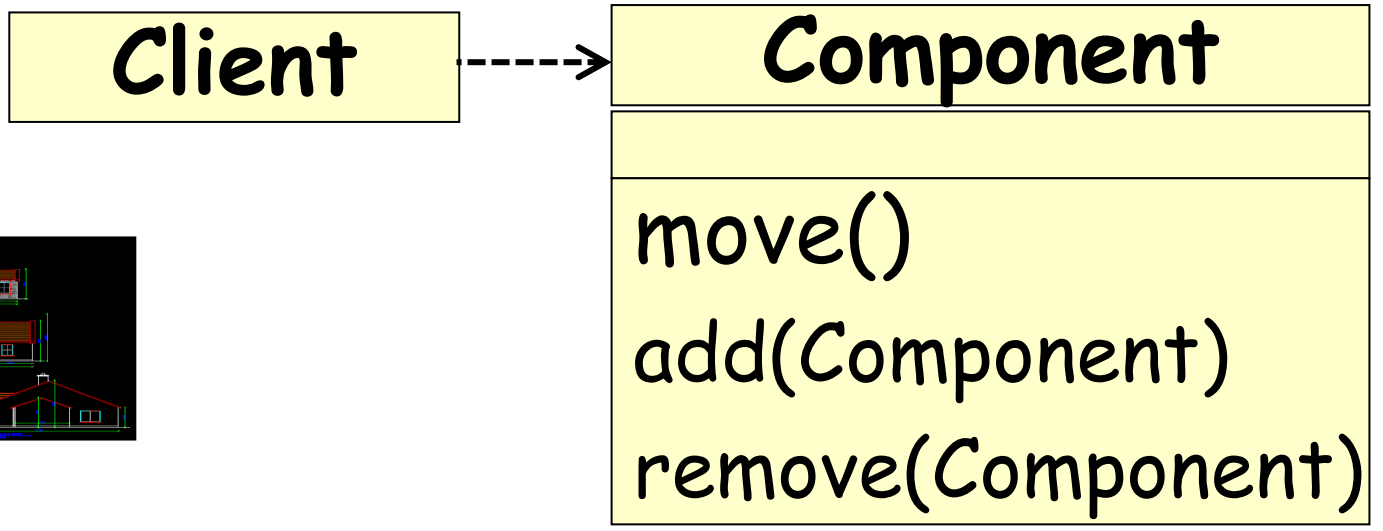
·Each node of the Component structure can respond to some common operation(s).

·The client can call operation of the Component and the structure responds "appropriately".

Client

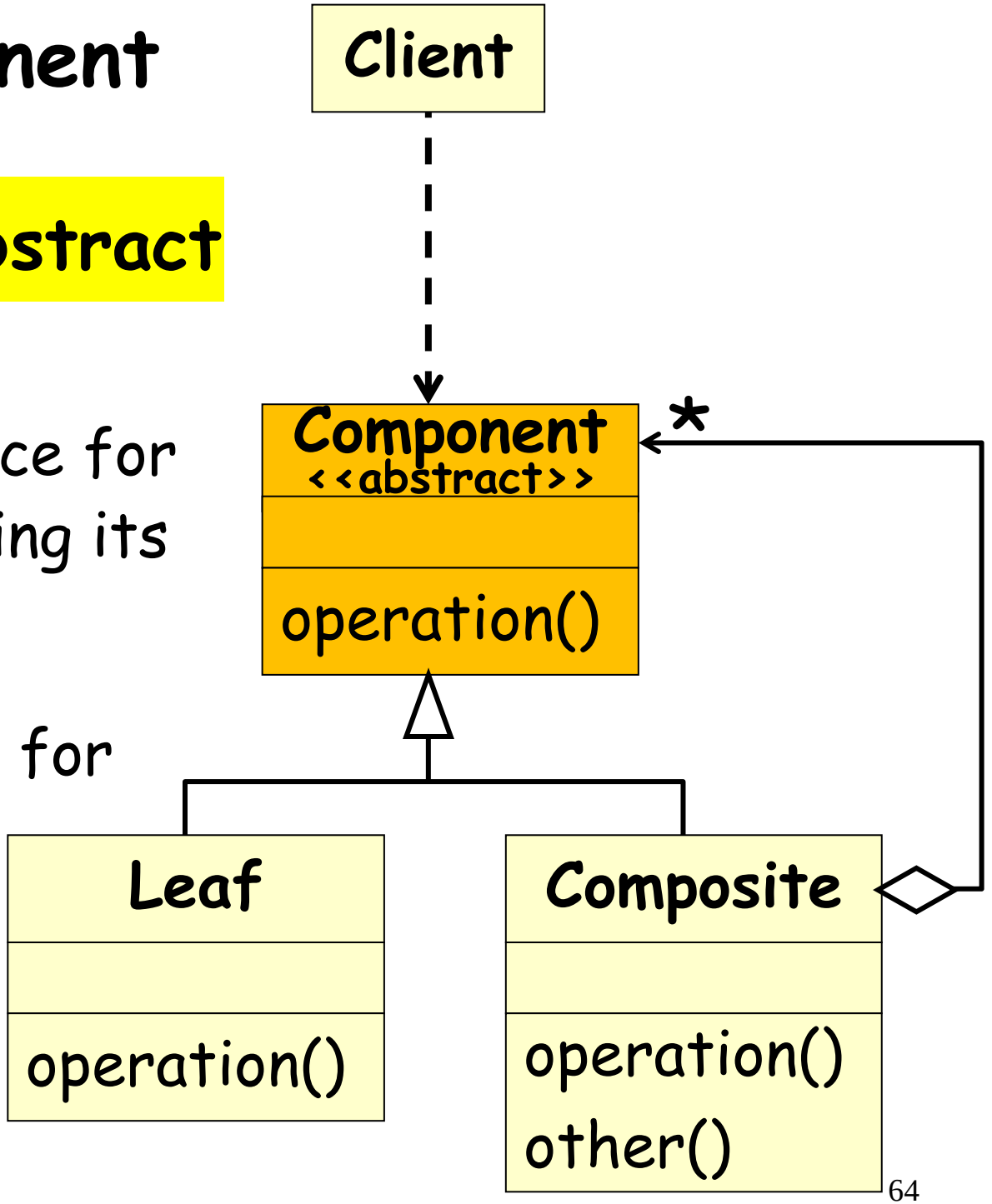
- The client is any class that operates on the composite:
  - It manipulates objects in the composition through the Component interface
- **Ex. VLSI CAD software**



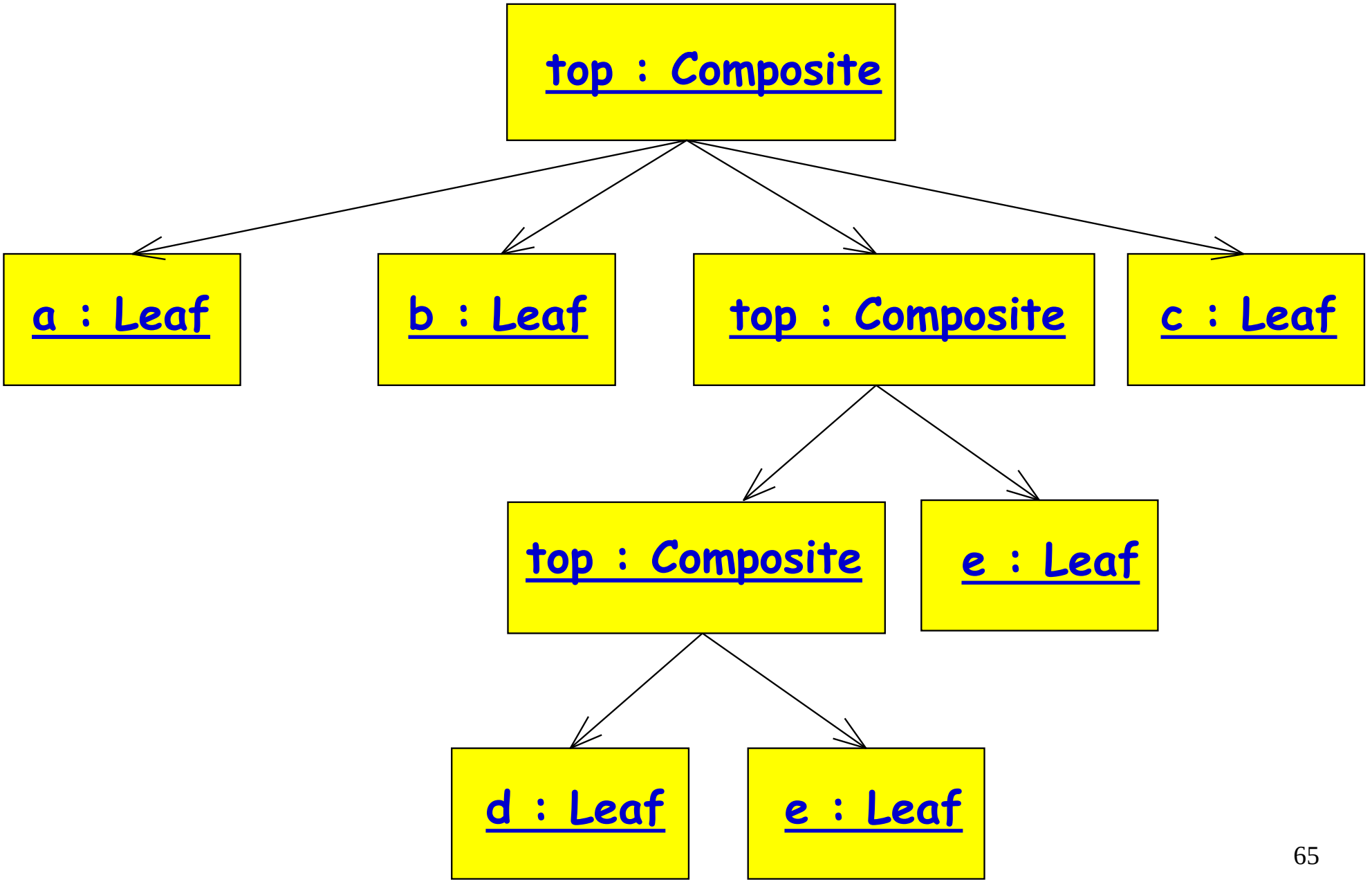


Component

- **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

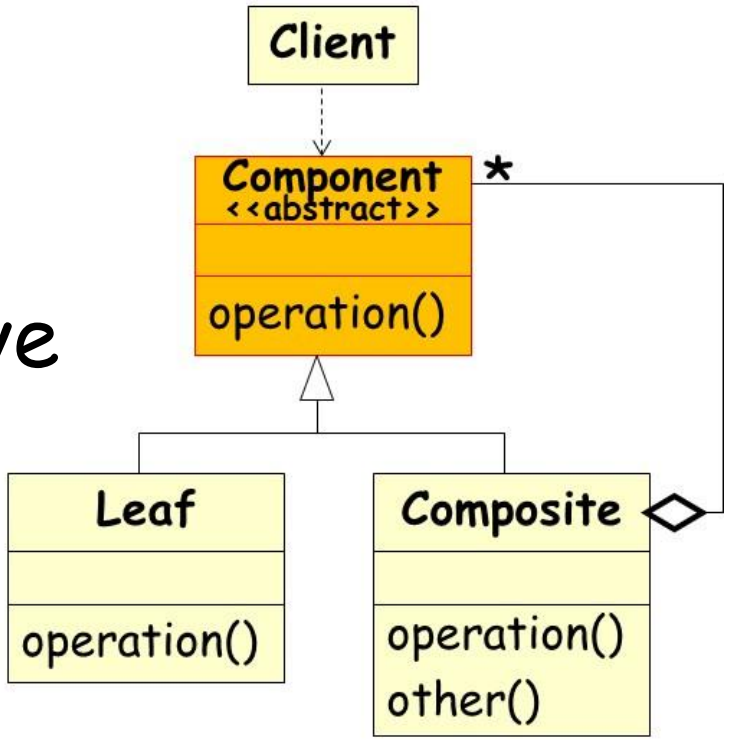


Composite: Object Diagram



Other Participants

- **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 management methods.

Interaction with Clients

- Clients use the Component class interface to interact with objects.
- If the recipient is a Leaf:
  - **Then request is handled directly**
- If the recipient is a Composite:
  - Forwards the request to child components