Object-Oriented Design: Design Patterns

Srinivas Pinisetty

Acknowledgement: (Prof. Rajib Mall)

Use of Patterns (in OOD)

- **Pattern**:
  - Describes a **problem that occur over and over** again.
  - Describes **core of the solution** of that problem in a way that it can be reused

- **Other engineering fields use patterns**?
  - Some other engineering disciplines have handbooks describing successful solutions to known problems
  - Reuse standard designs with successful track record.

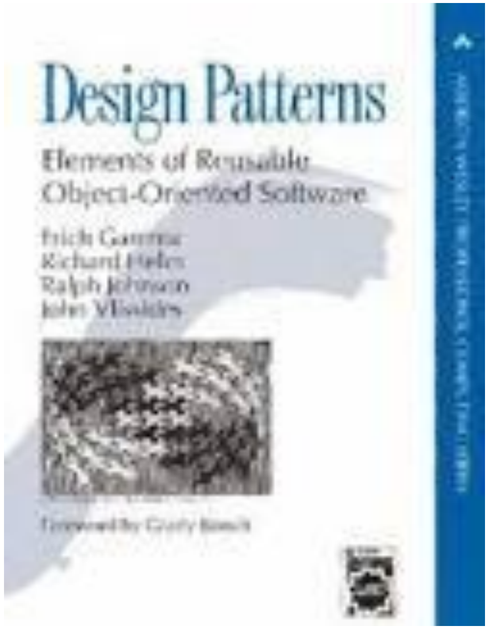
- **Should software engineers make use of patterns**?
  - Developing design/software from scratch is expensive
  - ***Patterns* support reuse of past knowledge** (in the form of software architecture and design)

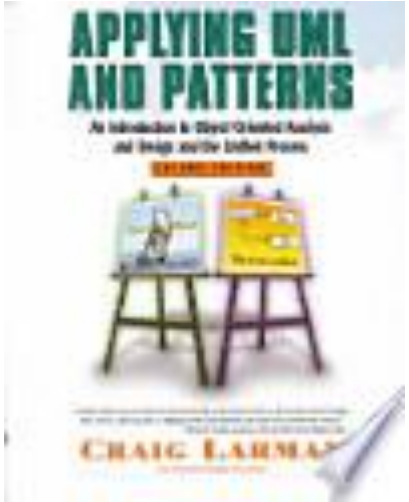
Patterns in Software Design

- Architectural Patterns: **MVC**, Layers, etc
- **GoF Design Patterns**: Singleton, Observer etc
- GUI design patterns: Window per task, Explorable interface etc
- Database patterns: decoupling patterns, cache patterns etc
- Concurrency patterns: Producer-consumer, asynchronous processing, double buffering etc
- **GRASP (General Responsibility Assignment Patterns)**: expert, creator, low coupling, high cohesion, controller, Law of Demeter, etc.,

“Gang of four” (GoF) and GRASP Patterns

- **Gamma, Helm, Johnson and Vlissides**
  - 23 different patterns




APPLYING UML AND PATTERNS
An Introduction to Object-Oriented Analysis and Design and the Unified Process
CRAIG LARMAN

- **Larman**
  - **GRASP (General Responsibility Assignment Patterns) pattern**

Elements of design patterns

Usually have four essential elements

- Pattern name: designer's vocabulary
- Problem: intent, context and when to apply
- Solution: UML model/ skeletal code
- Consequences: results and tradeoffs

Design patterns aim to?

- Give important design solution explicit name
  - Common vocabulary
- Codify good design
  - Generalize experience, aids novices and experts alike
- Save design iterations
  - Improve documentation
  - Improve understandability

Architectural patterns

- Architectural designs
  - Concern the overall structure of the software system
  - Form a basis for more detailed design
  - Cannot directly be programmed

- Architectural patterns
  - Providing high-level solutions to large problems

Design patterns

- A design pattern
  - Suggests classes in a design solution
  - Defines the interactions required among the classes..

- Design pattern solutions are described in terms of
  - Classes, their instances, roles, how they collaborate, skeletal code..etc

Patterns Vs. Algorithms

- **Algorithms & patterns..are they identical concepts?**
  - Both indeed target to provide reusable solutions to problems

- **Algorithms**
  - Mainly focus on solving problems *with reduced space/time* requirements
- **Patterns**
  - Focus on *easier development, understandability, reuse* and *maintainability*

Pros of design patterns

- Helps disseminate experts knowledge
  - Promotes reuse
- Provides common vocabulary
  - Improve communication among the developers
- Reduces number of design iterations
  - Improve design quality, productivity of designer..
- Good solution to common design problems by making use of
  - Abstraction, encapsulation, separation of concerns, coupling & cohesion, divide and conquer,…

Cons of design patterns

- Design patterns do not directly lead to reuse

- No systematic methodology exists to
  - Help select the right design pattern at the right point during the design activity

Why learn design patterns?

- Your design ideas will improve
  - Understanding well-tested/documented ideas
  - Description of patterns contain analysis of tradeoffs

- Will be able to describe complex design ideas to others
- Refactor existing design/code (improve structure of existing code)
- Better understand why some aspects of some high-level languages are designed in some specific way

Pattern problem

- **Pattern problem: describes the situation in which to use the pattern solution**
  - **Problem and its context**
  - **Situations where it works/does not work**
  - **Condition to be met to apply the pattern**

Pattern solution

- Overview of the solution
- Describes in terms of
  - Classes
  - Relationships
  - Responsibilities
  - Collaborations…

Design patterns are NOT

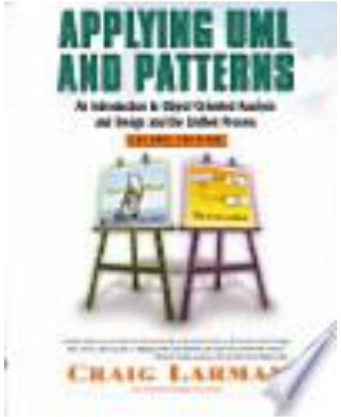
- **Not designs that can be plugged-in and reused as is**
  - Not code patterns (e.g., code for linked-lists etc)

- **Not complex "domain-specific" designs:**
  - For entire system/application

- **The essential idea:**
  - **If you can master a few important patterns, you can easily spot them in application development and use the pattern solutions.**

GRASP Patterns

Larman: GRASP (General Responsibility Assignment Patterns) pattern



- Can be said as “**best practices**”
- If guidelines used during design process, will lead to ***maintainable***, ***reusable***, ***understandable*** and easy to develop software

GRASP Patterns

- Describe **how to assign responsibilities** to classes

- **May be seen as guidelines** rather than concrete solutions

- What is responsibility?
  - Obligation/contract of a class
  - **Responsibility** related to **object creation**, **behavior**, **data storage**..etc

GRASP Patterns

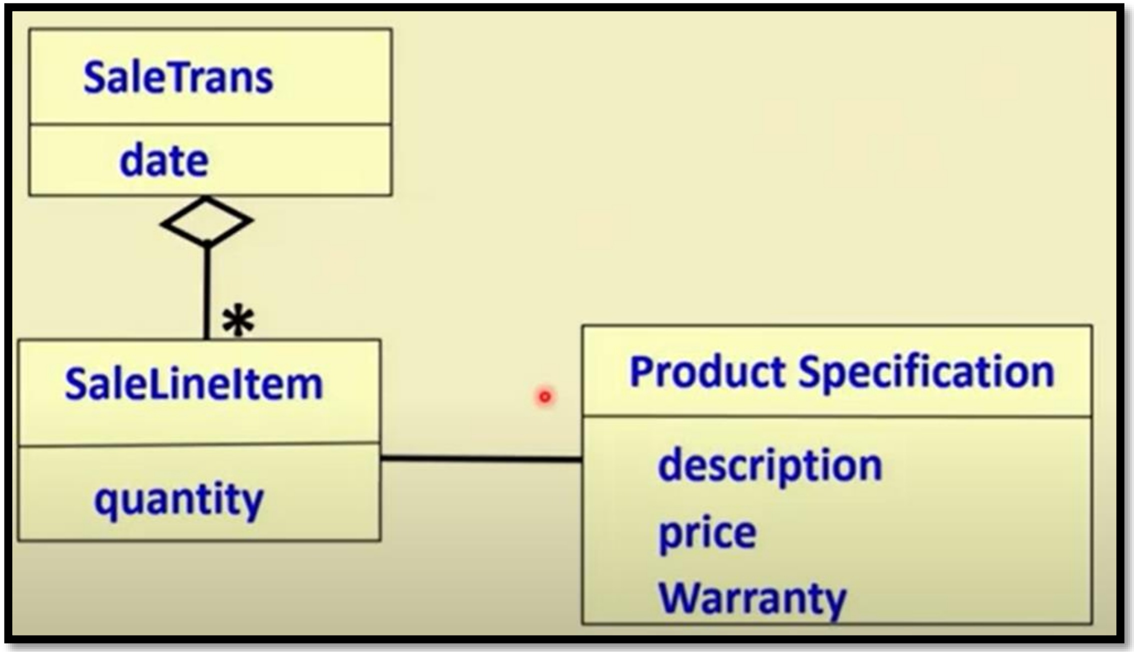
- **Creator** (who creates an object)
- Information **expert** (which class should be responsible)
- Low coupling (Support low dependency and increased reuse)
- **Controller** (who handles a system event)
- High cohesion (how to keep complexity manageable)
- Polymorphism (how to handle behaviour that varies by type)
- **Pure fabrication** (how to handle situation when we do not want to violate high-cohesion and low coupling)
- **Indirection** (how to avoid direct coupling)
- Law of demeter (Don't talk to strangers, knowing about unassociated objects)

Example Pattern: Expert

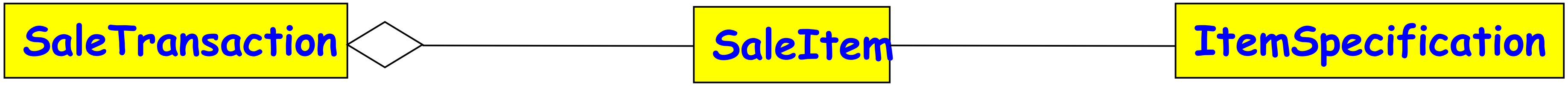
- **Problem**: Which class should be responsible for doing a certain thing?
- **Solution**: Assign responsibility to expert(the class that has all/most of the information necessary) to fulfil the required responsibility

Example: Information expert for computing total price of a sales transaction

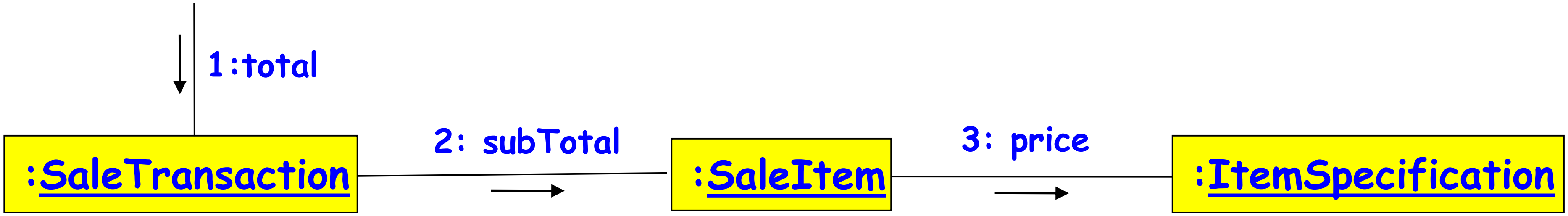
Compute_total_Price



Example Pattern: Expert Cont…

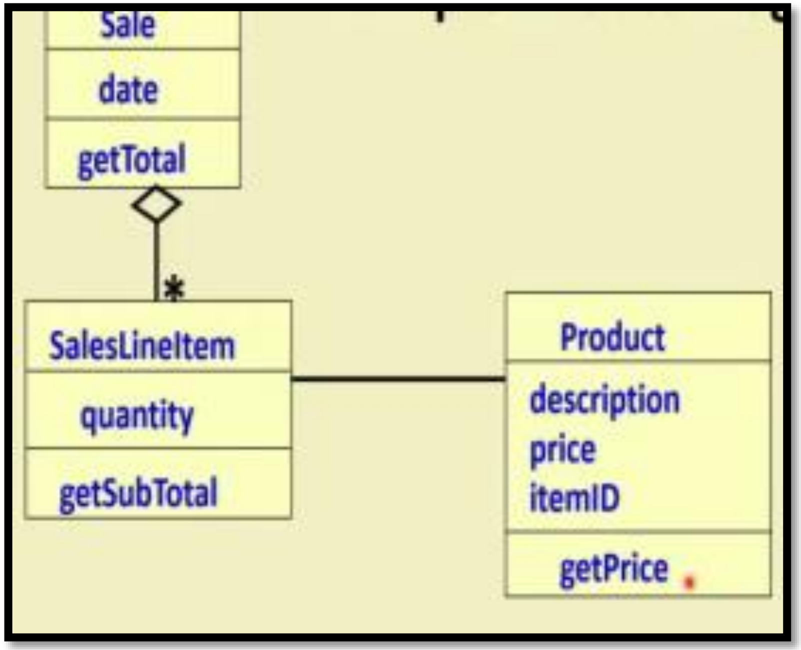


Class Diagram



Collaboration Diagram

Example Pattern: Expert Cont…



Example 2: Tic-tac toe (Initial and Refined Domain Model)

Board

Initial domain model

PlayMoveBoundary

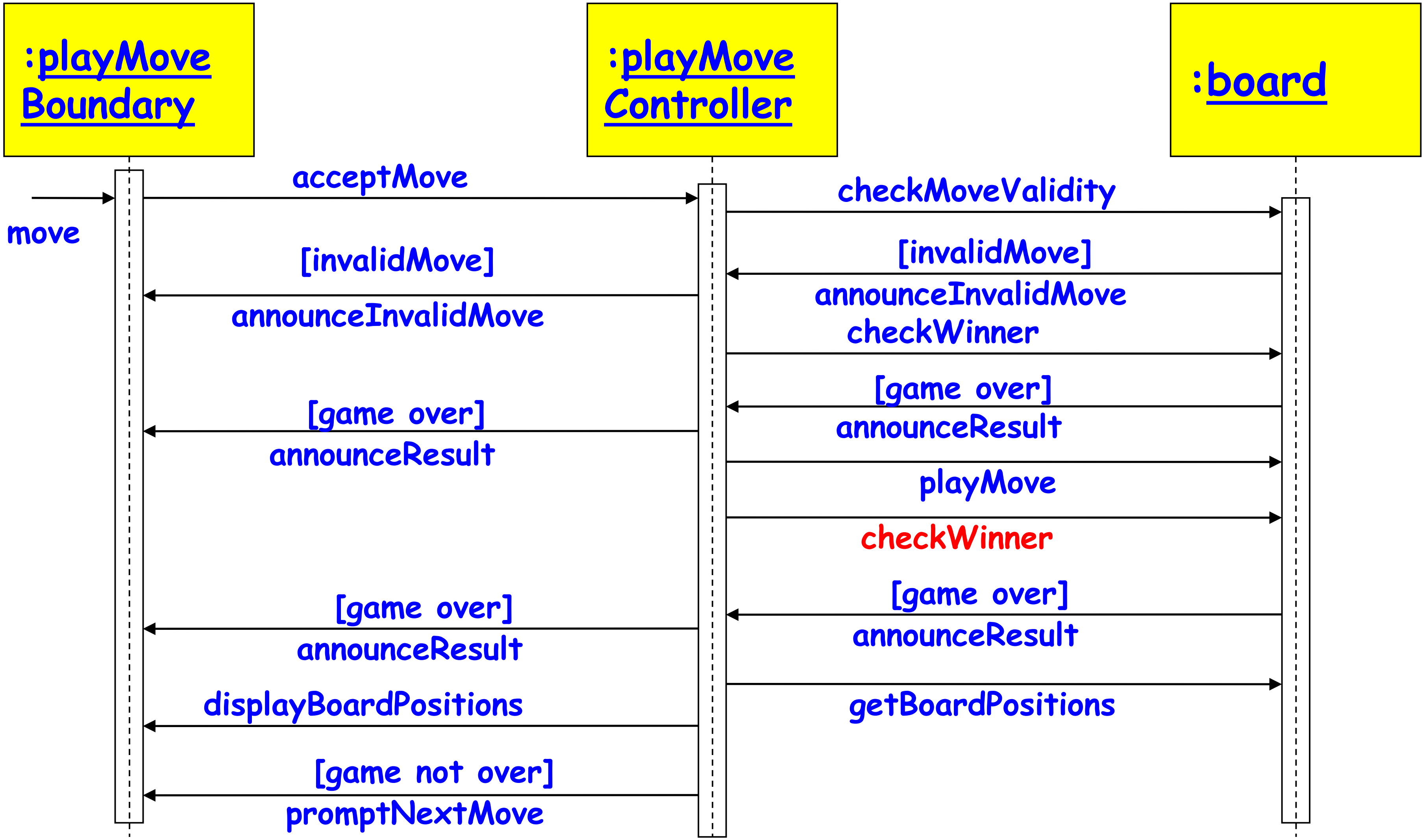
PlayMoveController

Board

Refined domain model

Which class should check game result after a move?

Example 2: Sequence Diagram



Sequence Diagram for the play move use case

Expert Pattern

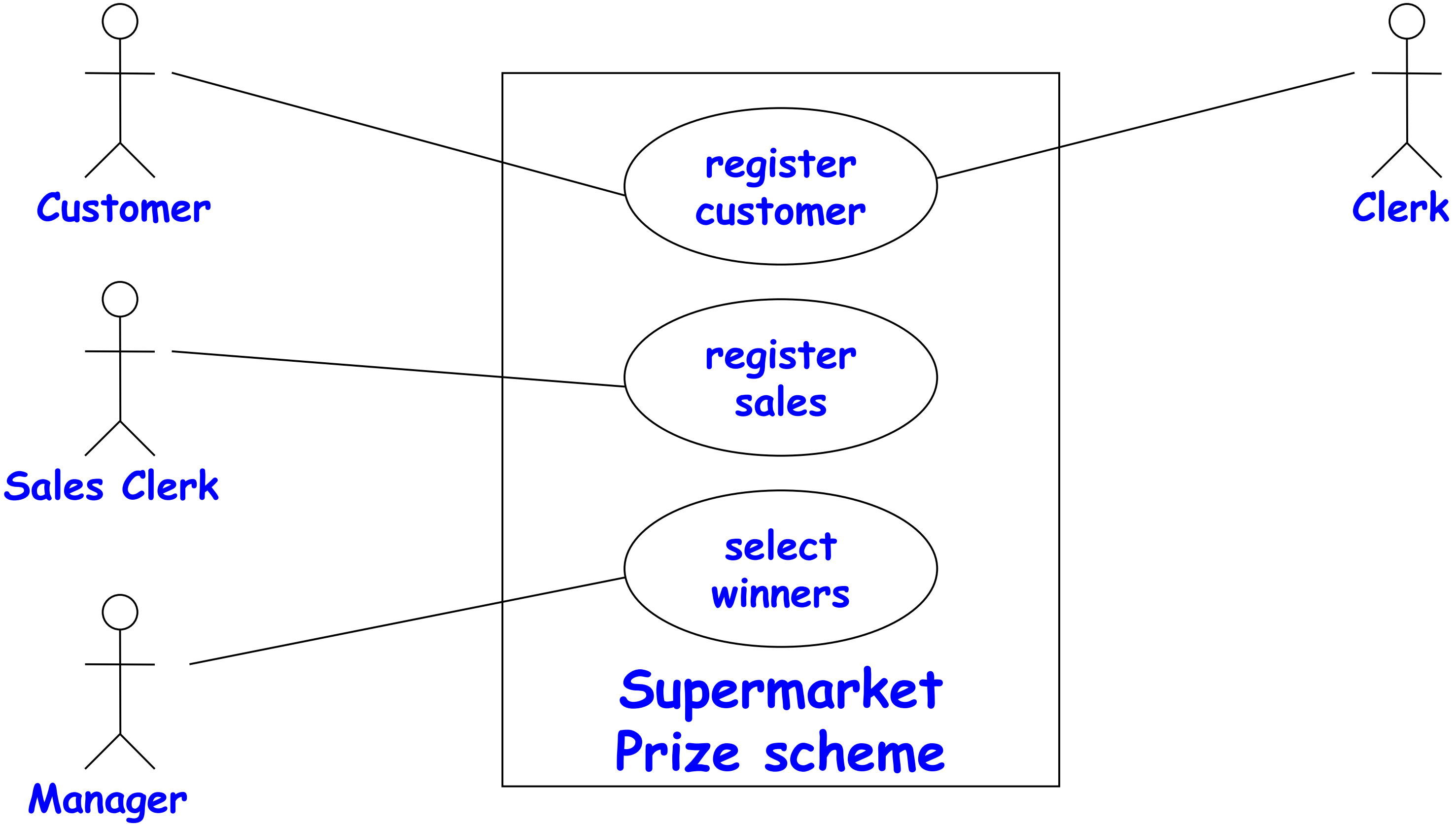
- Expert improves cohesion
- Information needed for a responsibility is in the same class

Example Pattern: Creator

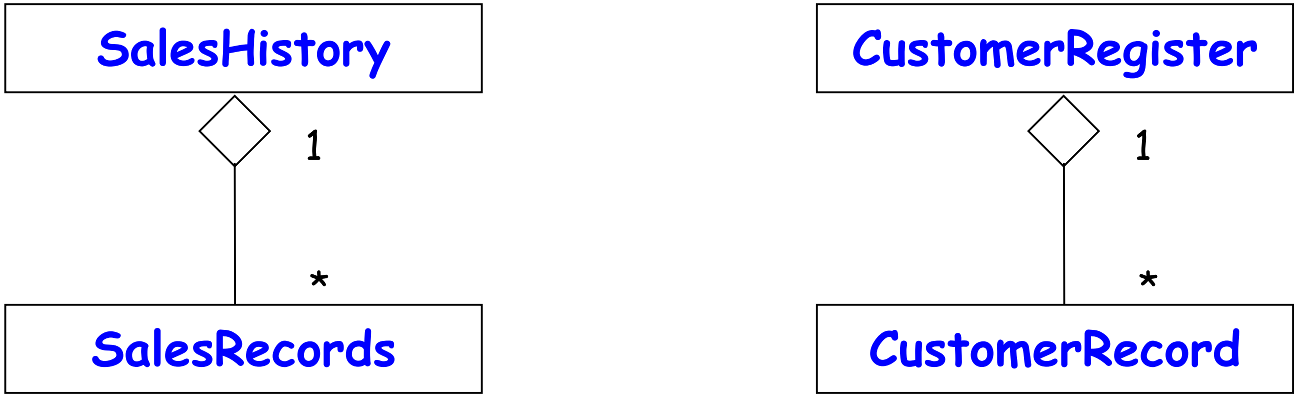
Every object must be created somewhere

- **Problem**: Which class should be responsible for creating a new instance of some class?
- **Solution**: *Assign* a class **C1** the responsibility to create object of class **C2** if
  - **C1** is an **aggregation** of objects of type **C2** (inventory of objects of type C2)
  - **C1 contains the information to initialize** the object
  - **C1 closely uses C2** (It will be primary client of the object)

Example: Supermarket prize scheme (Use Case Model)

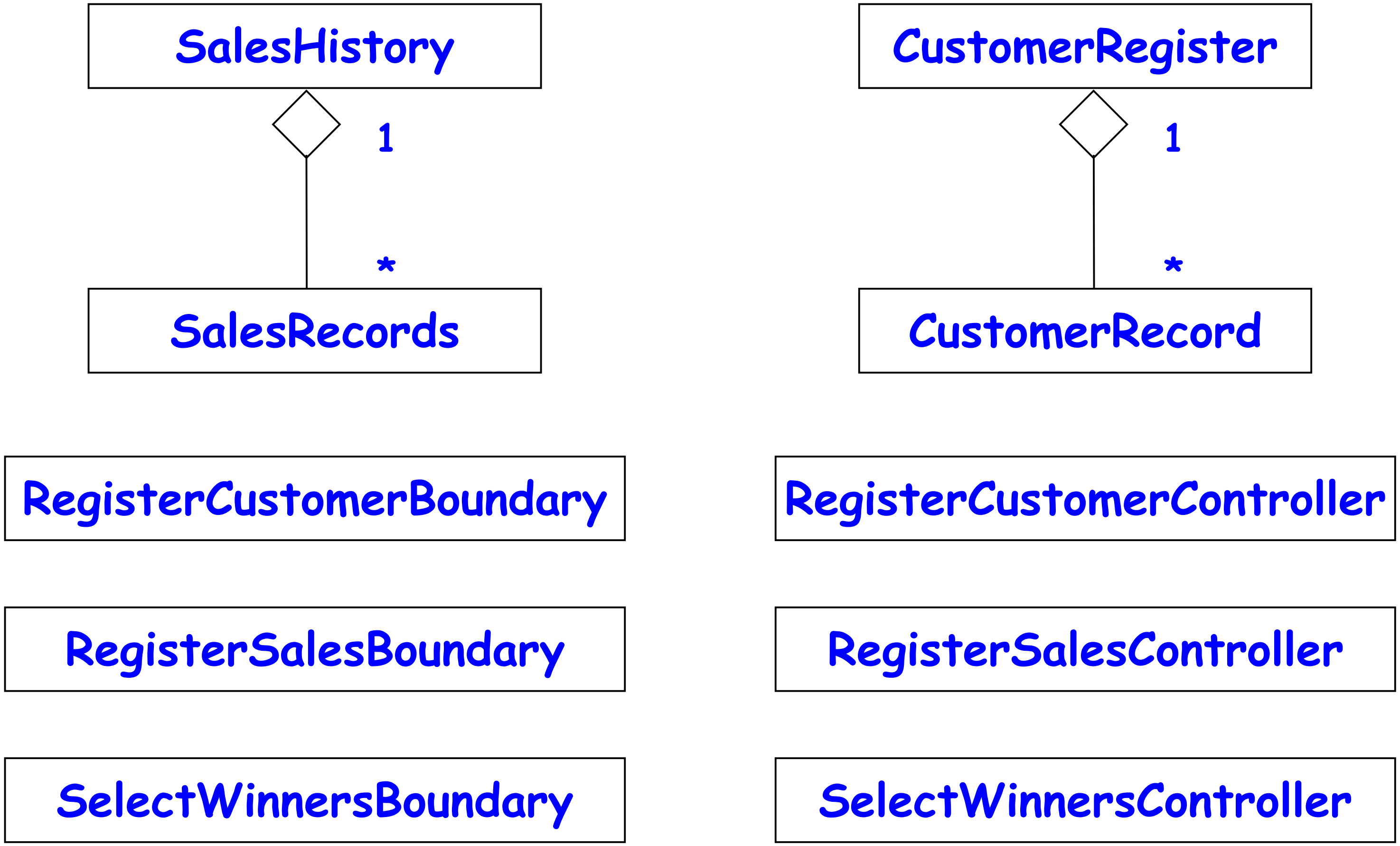


Example 2: Initial Domain Model



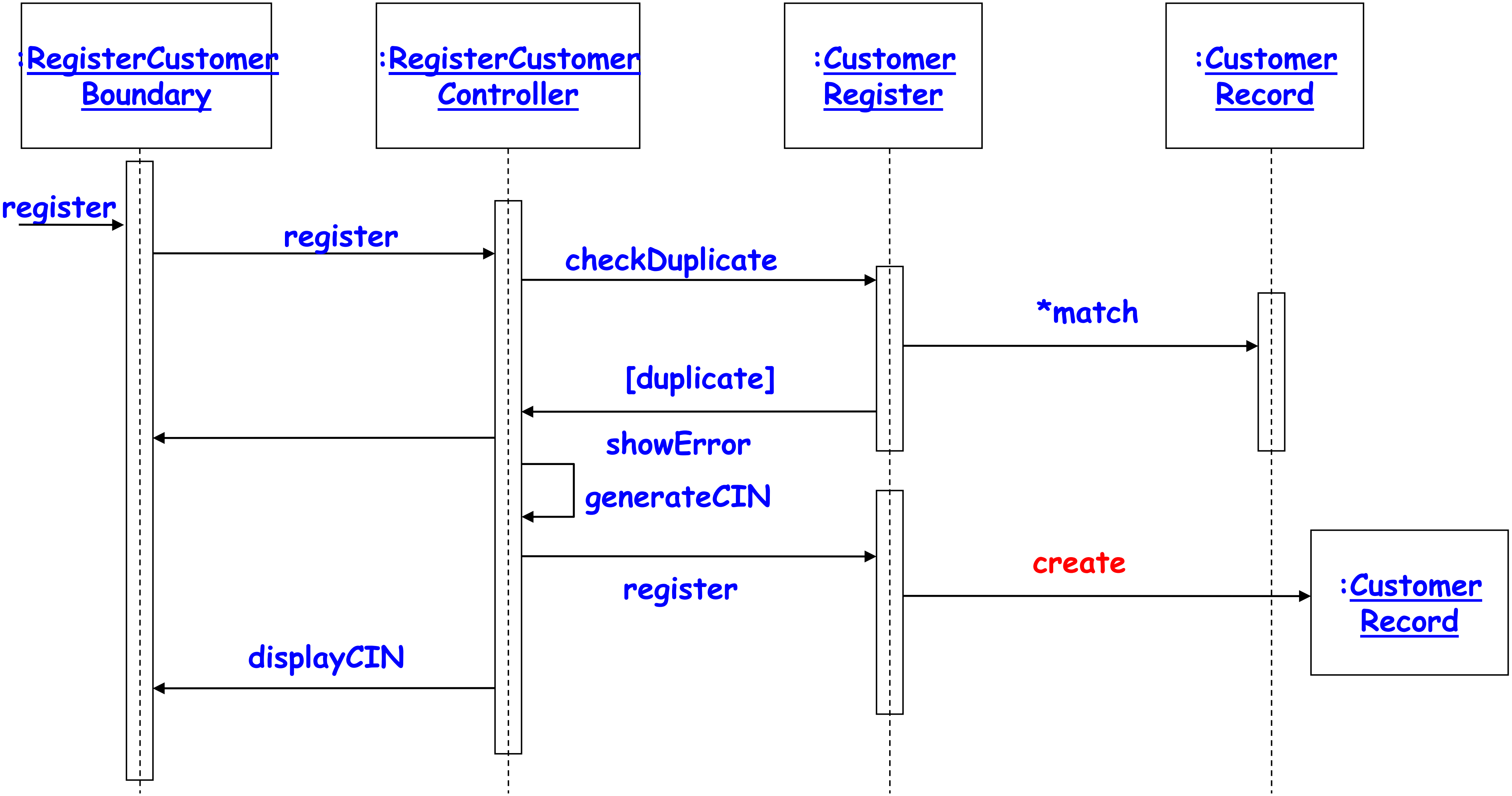
Initial domain model

Example 2: Refined Domain Model



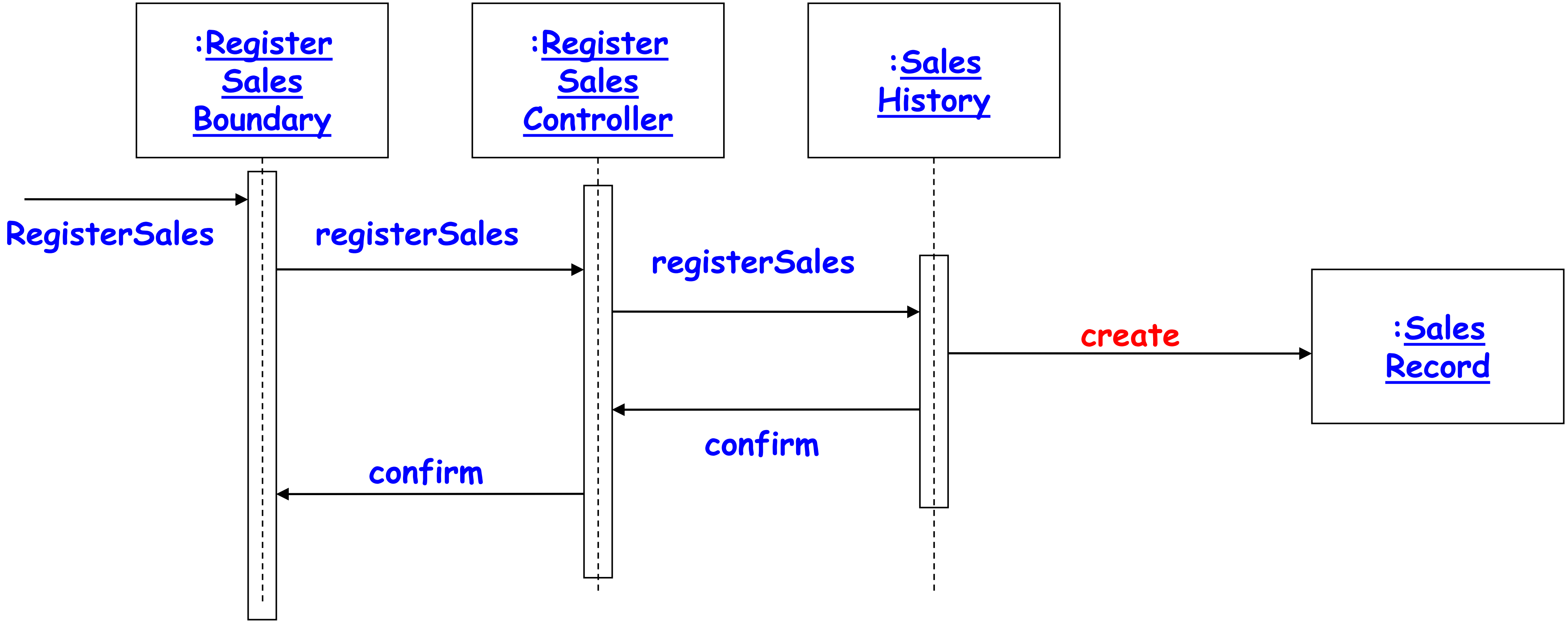
Refined domain model

Example 2: Sequence Diagram for the Register Customer Use Case



Sequence Diagram for the register customer use case

Example 2: Sequence Diagram for the Register Sales Use Case



Sequence Diagram for the register sales use case

Creator Pattern

- Creator: Ensures that coupling due to object instantiation occurs between closely related classes

- Aggregate/container of a class is already coupled with that class
  - Thus, assigning the creation responsibility does not worsen/affect the coupling in the design

Example Pattern: Controller

- **Problem**: Who should be responsible for **handling the actor requests?**

- **Solution**: **Separate controller** object for each use case.

Controller Pattern

- Controller object: the responsibility of receiving and handling an actor message

- This responsibility should not be assigned to view (or) model class

Pattern: Pure Fabrication

- Suppose that a class has some responsibilities unrelated to its main task

- It may lead to a bad design- low cohesion/ high coupling

- How to improve the design?

Pattern: Pure Fabrication

- How to improve the design when a class has high coupling/low cohesion?

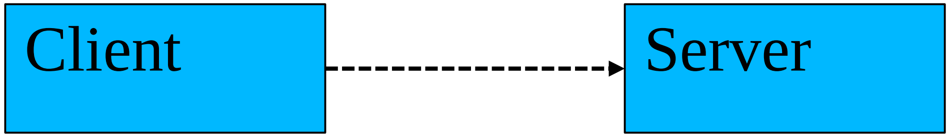
- Solution
  - Create a new artificial class
  - Separate the highly-cohesive responsibilities from the others
  - New class only to support high cohesion, low coupling and reuse….

Pattern: Indirection Pattern

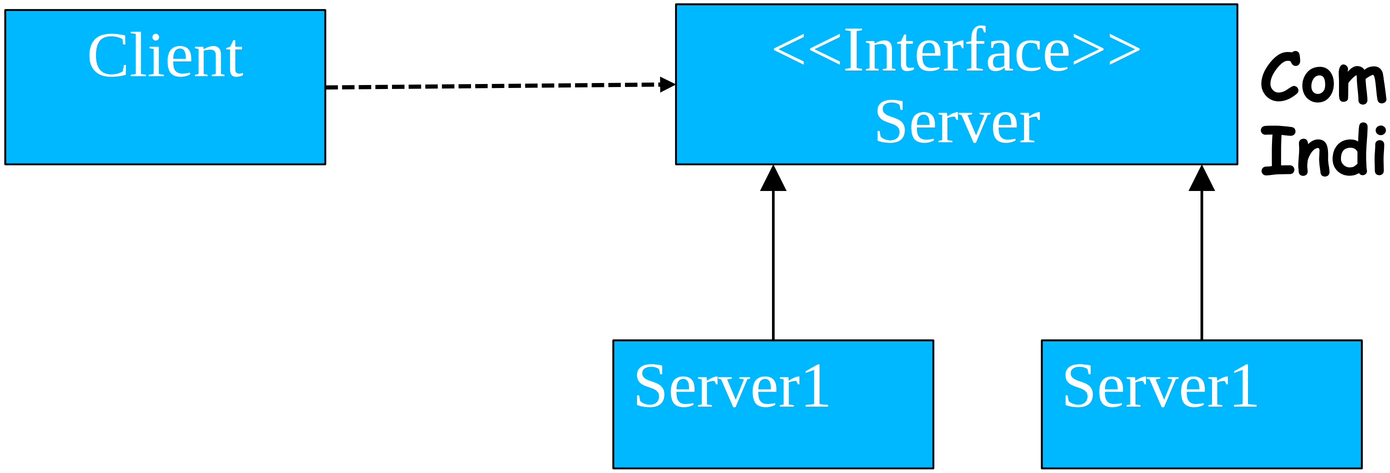
- **Problem**: How to avoid direct coupling between classes?
- How to decouple object to achieve low coupling and ease modifications/changes?

- **Solution**: Use Interface Class so that objects are not directly coupled..

Pattern: Indirection Pattern



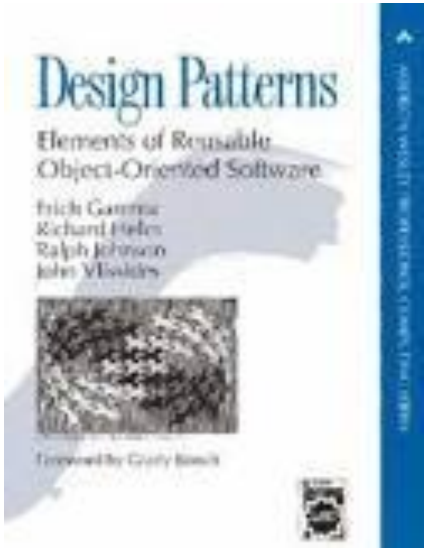
Violates indirection pattern (changes to server also require modifications to client)



Compliant with Indirection pattern !

“Gang of four” (GoF) Patterns

- **Gamma, Helm, Johnson and Vlissides**
  - 23 different patterns



“Gang of four” (GoF) Patterns

Unlike the grasp patterns, here the **problems are very specific** and also **the solutions are very specific (**in terms of class diagrams, Java code,….)

Types of GoF Patterns

- **Creational patterns:**
  - **How objects are created?**
  - Provide simple abstraction for a complex instantiation process
  - Make the system independent from the way its objects created, composed, and represented
- **Structural patterns:**
  - **How classes and objects are composed into large groups?**
  - Introduce **abstract classes** to **enable future extensions**
  - ***Adapters***, ***bridges***, ***facades***, and proxies….
- **Behavioral patterns:**
  - **How responsibility is distributed?**

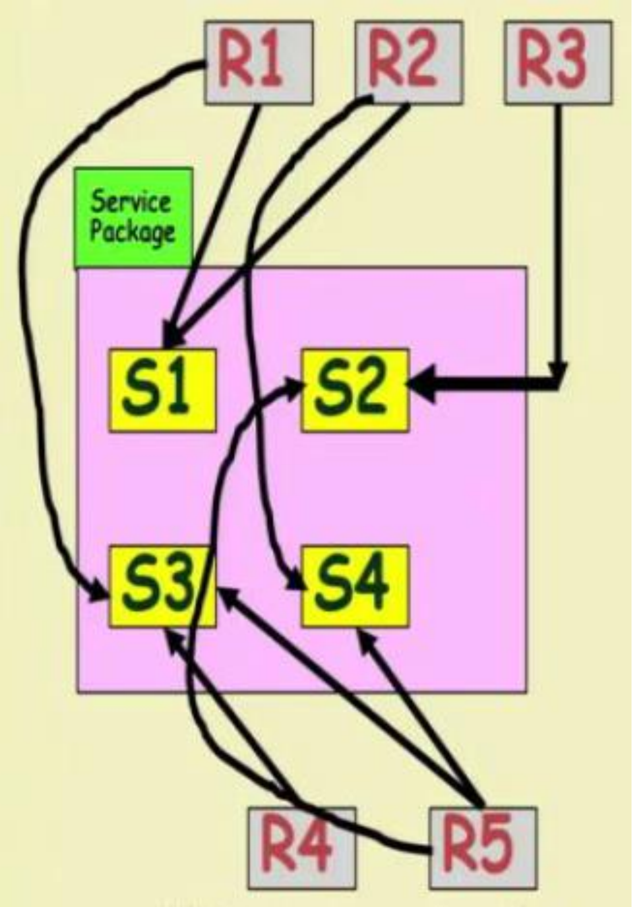
Example Pattern: Facade

- Problem: How should the services be requested from a service package?

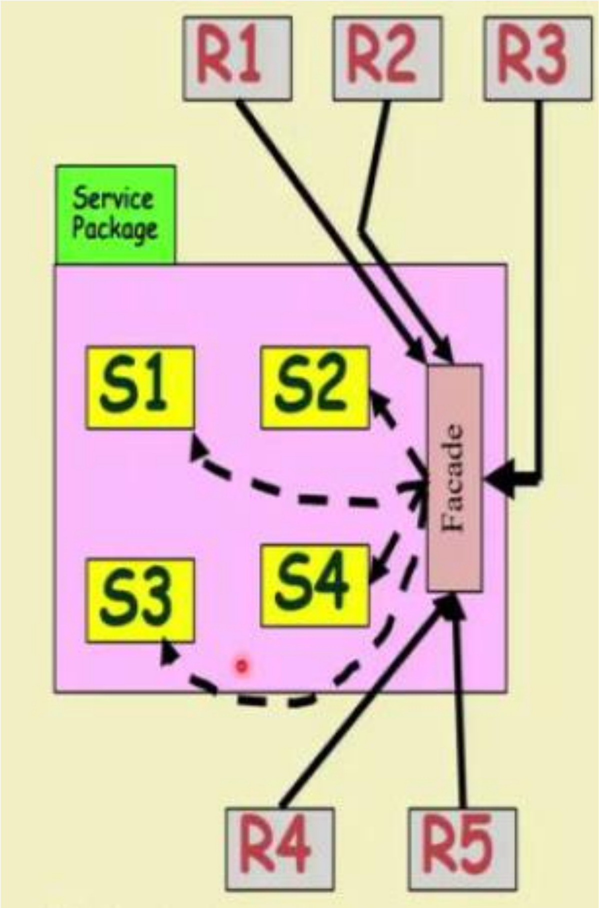
- Context (problem): A package (cohesive set of classes), example: RDBMS interface package

- Solution: A class (DBfacade) can be created which provides a common interface to the services of the package

Pattern: Facade

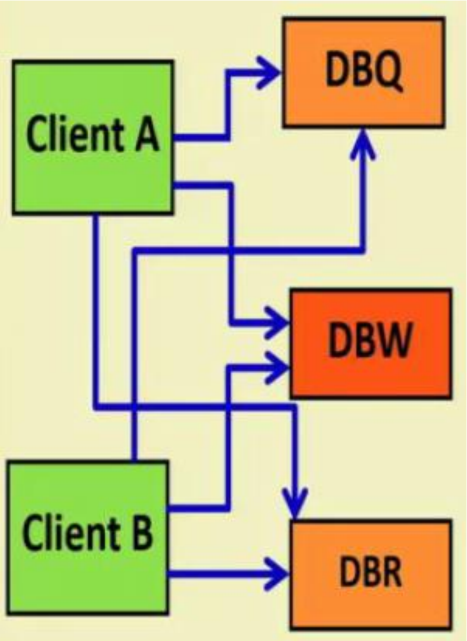


Service invocation without a facade

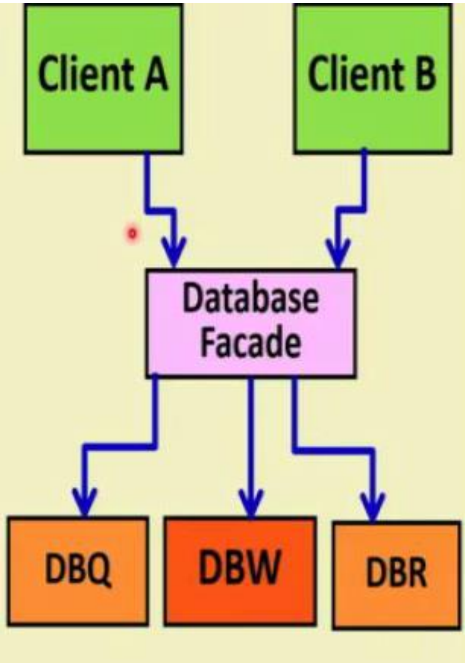


Service invocation using a facade class

Pattern: Facade



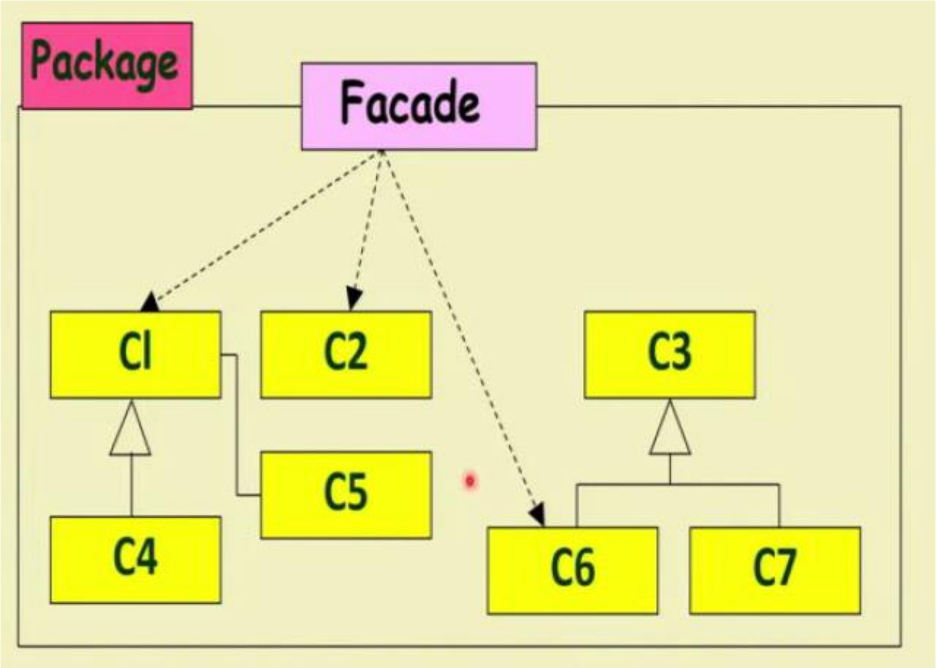
without a facade



using a facade class

- With facade, only frontal of the package visible to the clients
- Client class only associated with facade class
- Changes in server does not affect client

Pattern: Facade



Facede Structure: For every service package, **create a facade class**

Example Pattern: Observer

Problem: When a ***model* object** changes state ***asynchronously*** and is accessed by several ***view*** objects

- How to structure the interactions between the model and view objects?

Context: there could be many observers (can vary dynamically)

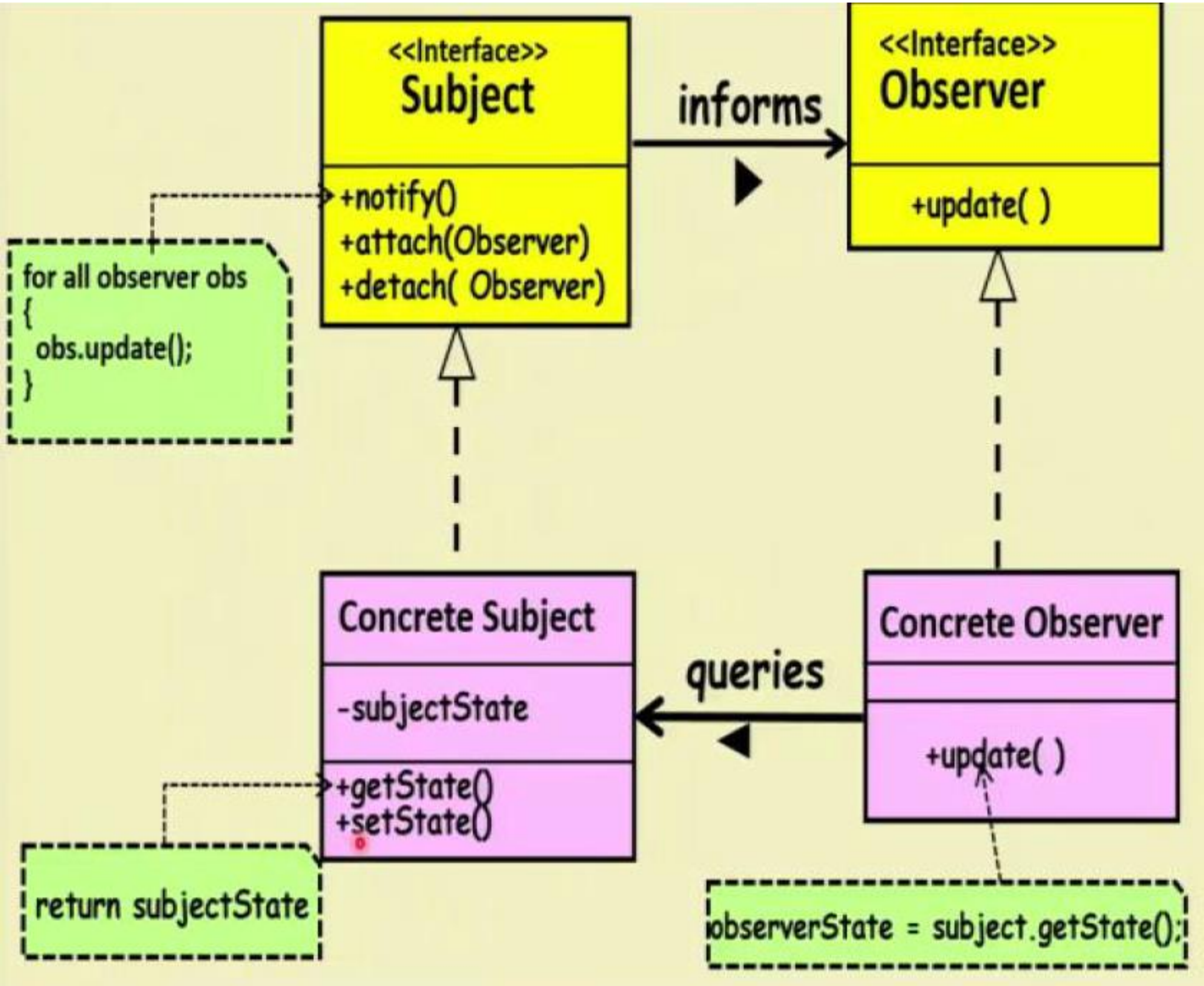
- Each observer may react differently to an update/notification

Example Pattern: Observer

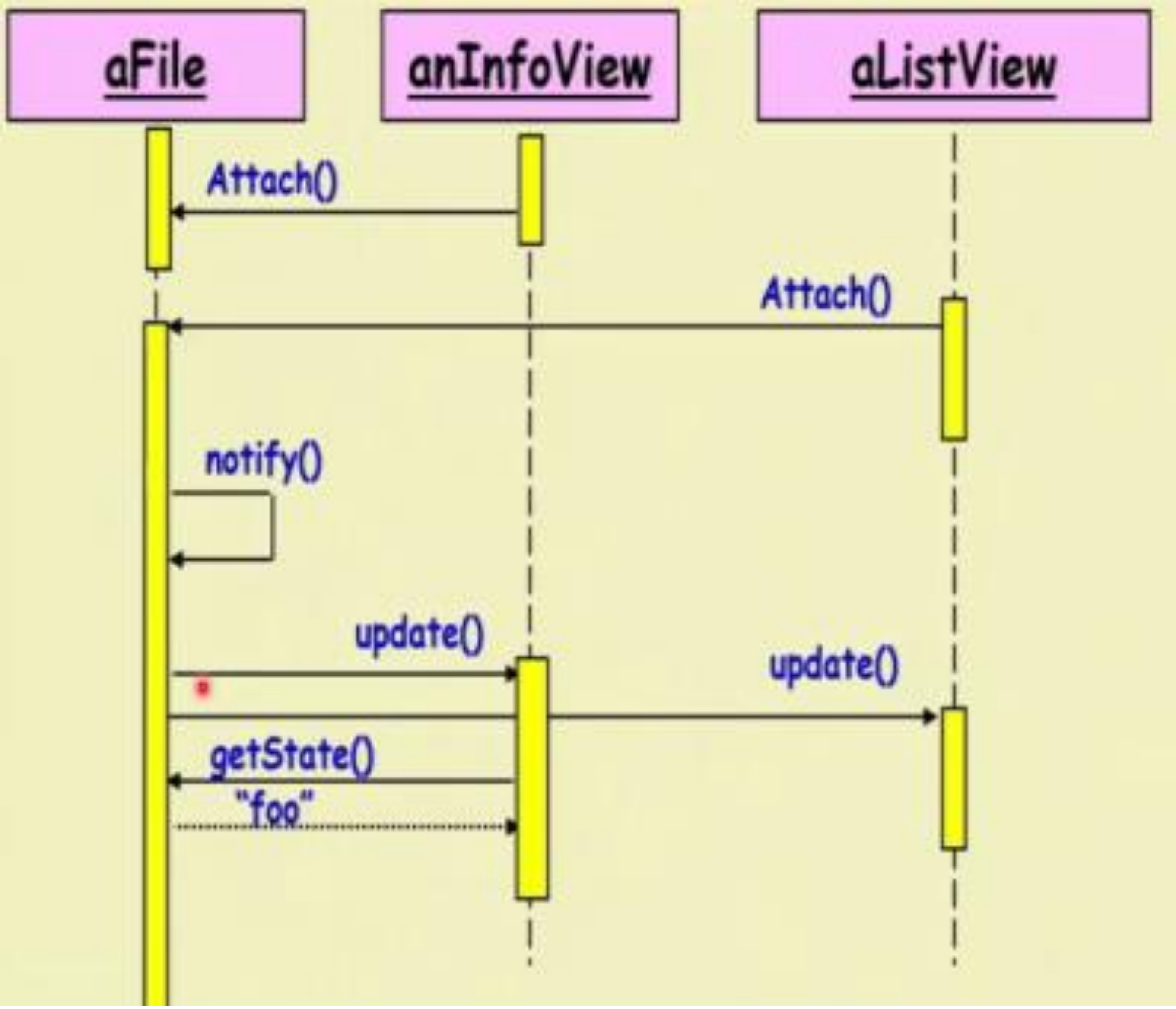
Solution: define a **one-to-many dependency** so that when the model changes state, all the dependents are notified and updated automatically.

- Observers first report to the model (model stores the ID of the observers)
- Many observers can ***attach*** themselves to the model,
- whenever there is a change the model communicates to attached observers
- When observer wants to leave, they communicate to the model

Structure: Observer Pattern



Structure: Observer Pattern



Composite Pattern

Composite Pattern

- **Problem:**
  - To represent part-whole relationship of objects
  - Clients to **ignore the differences** between **parts** and **whole**
  - **Parts should be created dynamically**
    - E.g., building a complex system from primitive components and previously defined subsystems
    - Reusing subsystems defined earlier

Composite Pattern

- A composite is a **group of objects in which some objects contains others**
  - An object may represent a group;
  - Or may represent an individual item.
- **Example: CAD Design**

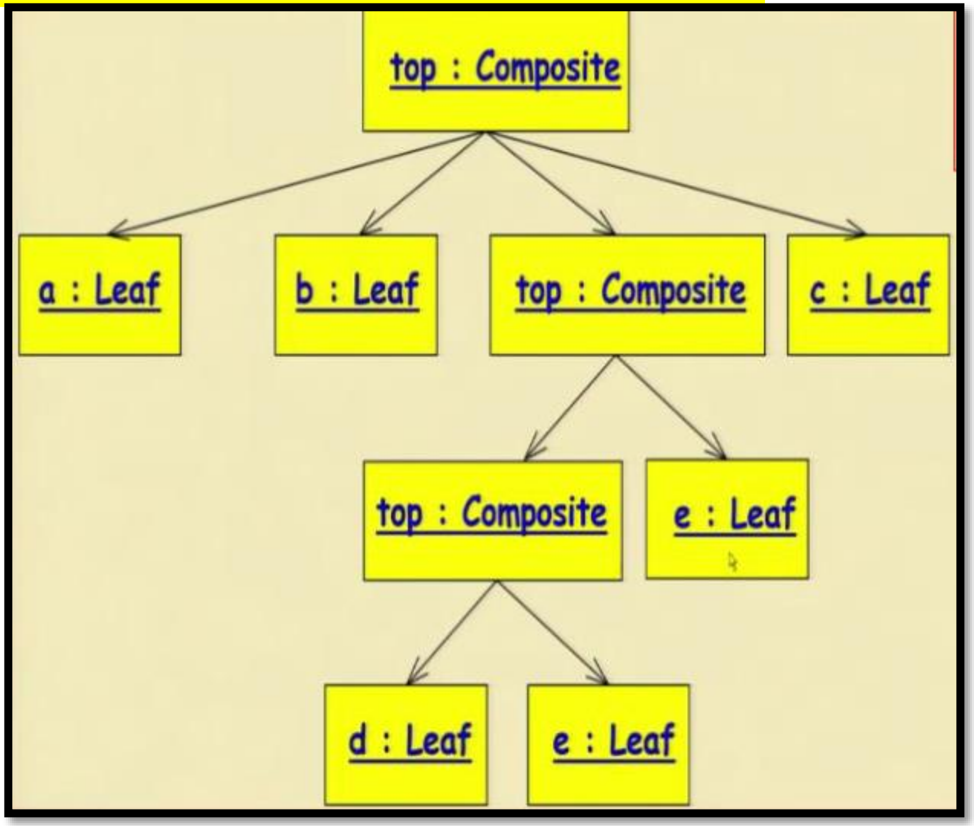
CAD Editor

- Build complex diagrams using simple components
  - Group components to form larger components
  - Which can be further grouped to form still larger components

Composite pattern- **Intent**

- To compose nested group 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**

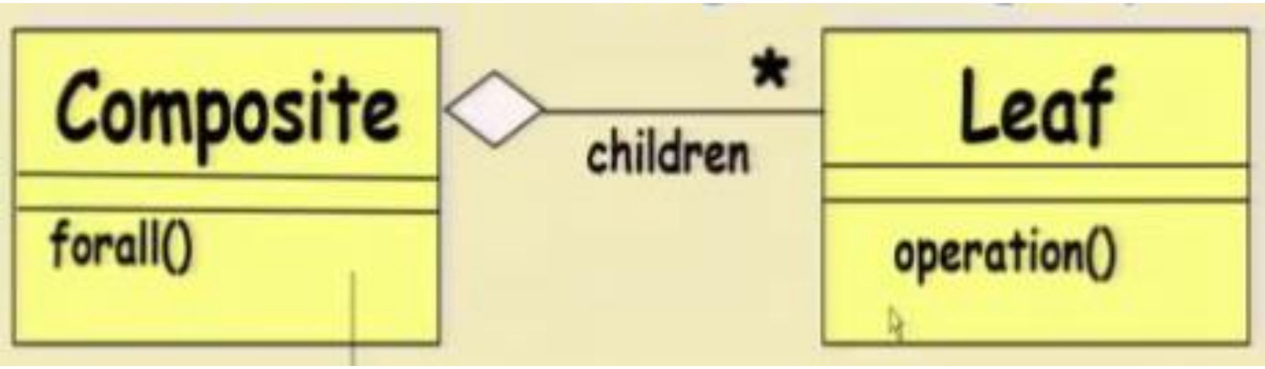


Why Composite pattern?

- If Composite pattern not used
  - Client code should treat primitive and container classes differently..
  - Makes the application more complex
  - Additions of new types of components is challenging

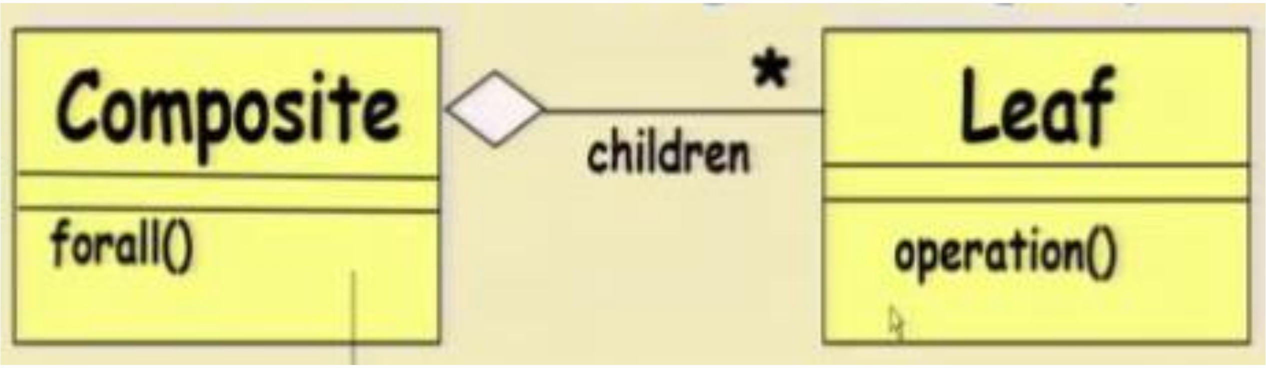
Composite- Solution (Attempt 1)

- Is this a good solution? Any issues?



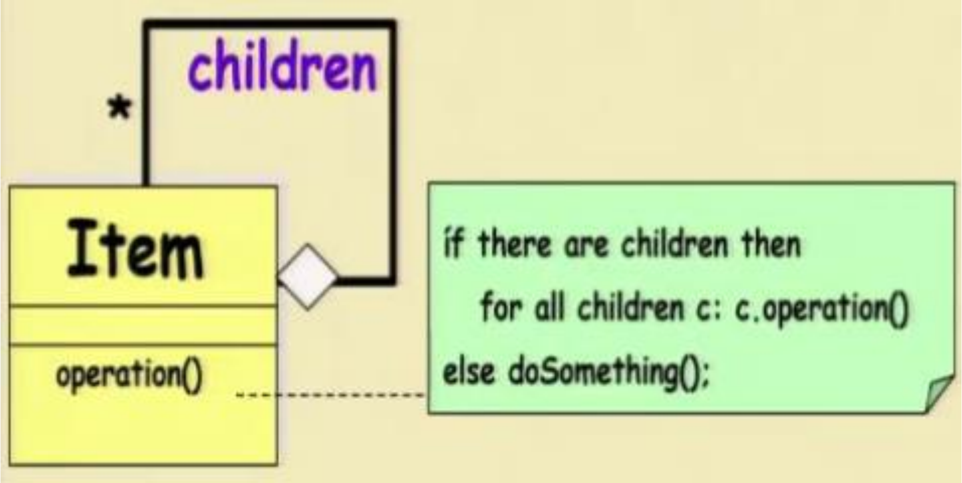
Composite- Solution (Attempt 1)

- Is this a good solution? Any issues?



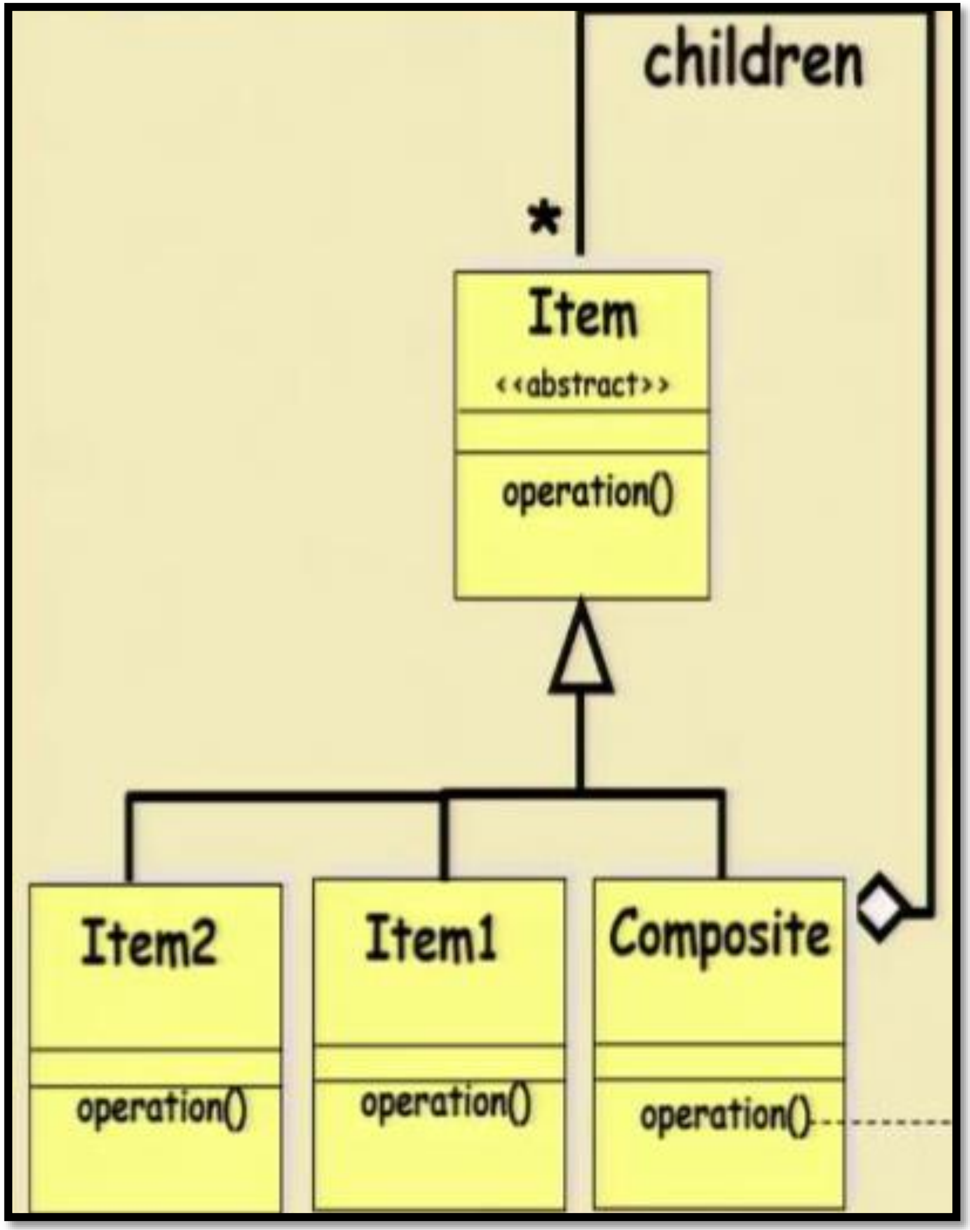
- Analysis (solution 1)
  - Restricted to only one level of nesting
  - Composite and leaves should be treated differently by the client, difficult to extend

Composite- Solution (Attempt 2)



- **Better than the previous solution…**
  - Unrestricted depth
  - Unified treatment in client
- **Any problems?**
  - Different item types (primitive Vs composite) cannot be handled
  - Difficult to extend with new leafs etc

Composite- Solution

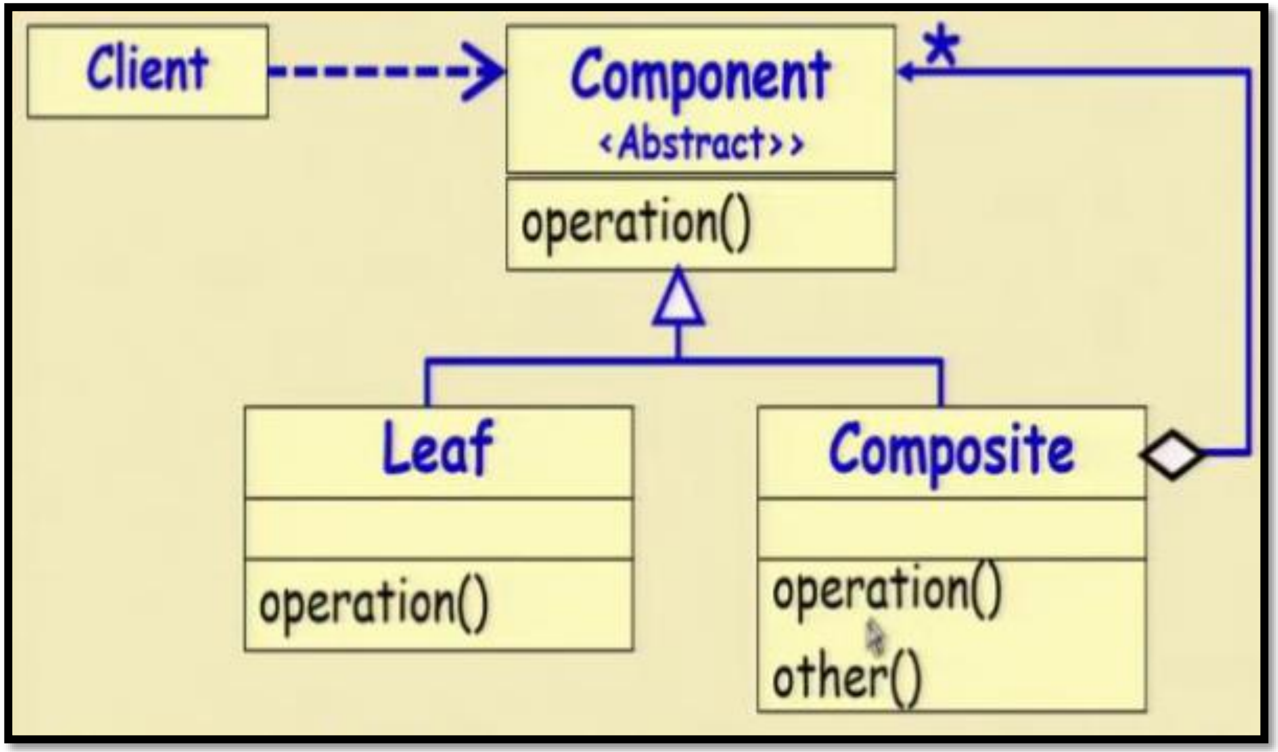


Organize a hierarchical object structure so that clients are not required to be aware of the hierarchy…

Composite pattern

- What is the class structure
- How does the client interact
- What operations to define for
  - The component/composite and the leaf.
  - How are the operations carried out

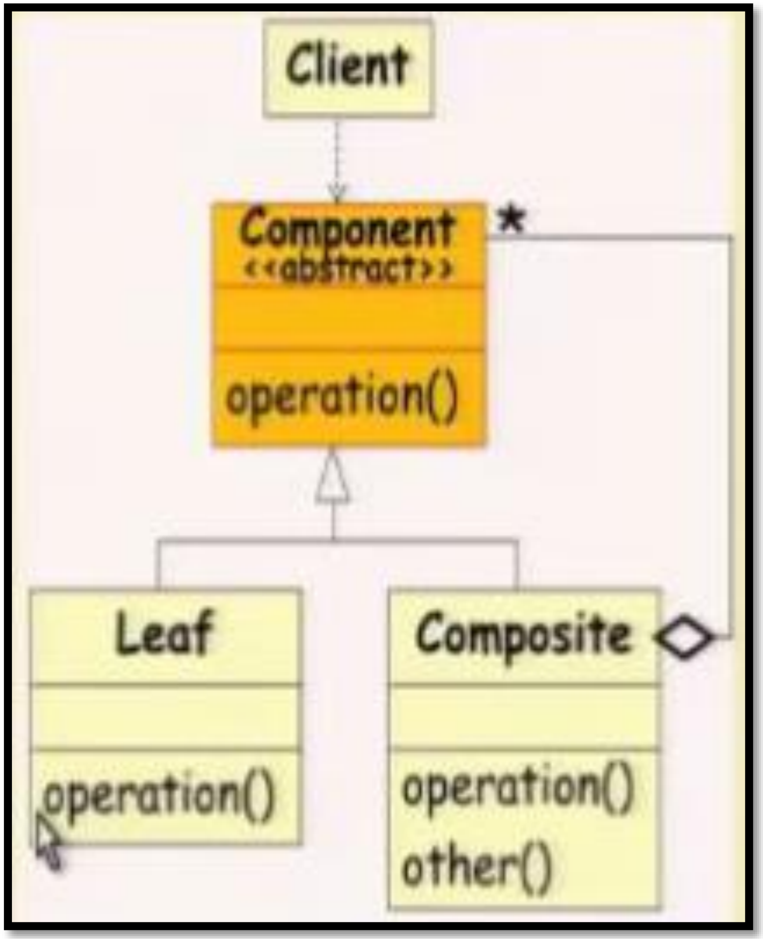
Composite pattern



The client calls operation of the component and the structure responds “appropriately”.

Composite pattern- Component

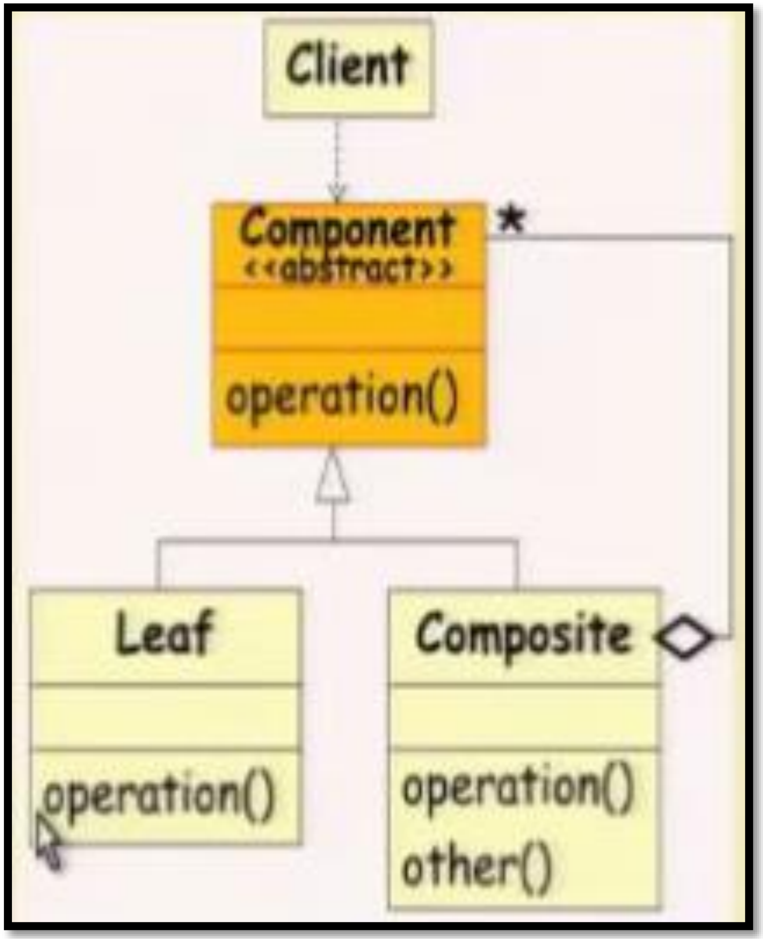
- Component- Abstract class
  - Interface for accessing/managing its child components
  - Defines an interface for default behavior



Composite pattern- Leaf & composite

- **Leaf**
  - Defines behavior of primitive objects
  - Has no children

- **Composite**
  - Includes behavior for components having children
  - **Stores child components**

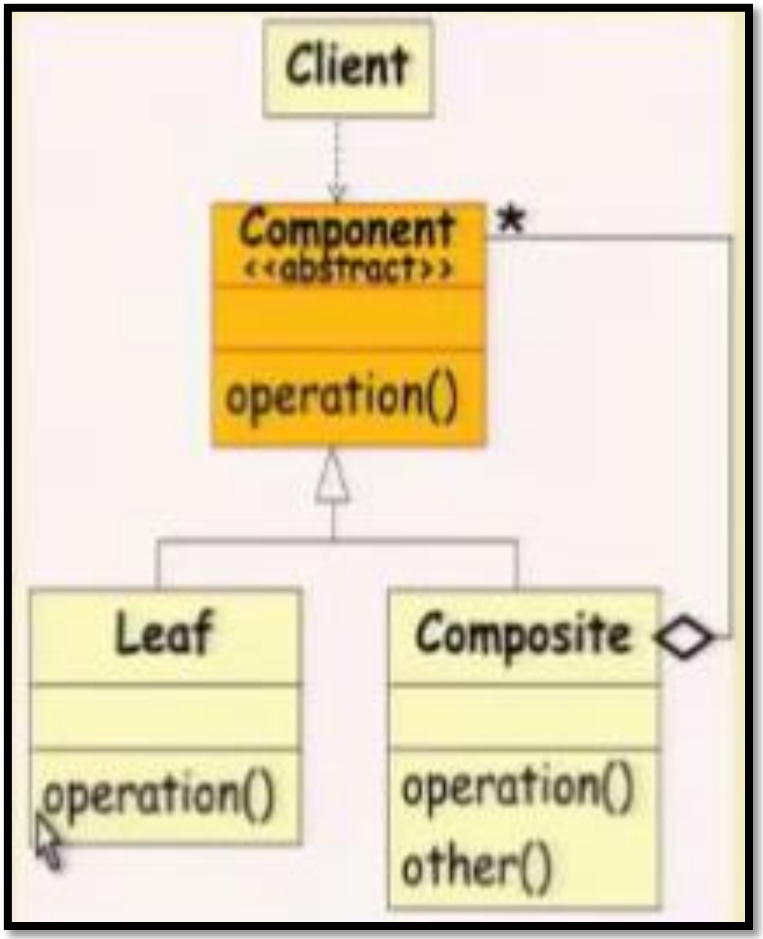


Composite pattern

- ***Client* uses** the **Component** class **interface**, that in turn interacts with the objects

- If ***leaf***, handles the request directly

- If ***composite***, forwards request to its child components



Composite pattern- Consequences

- **Allows to define recursive composition of primitive and composite objects**

- **Invocation by clients is made simpler** (client does not need to know whether dealing with composite or leaves).

- **Easier to extend** with new kind of components

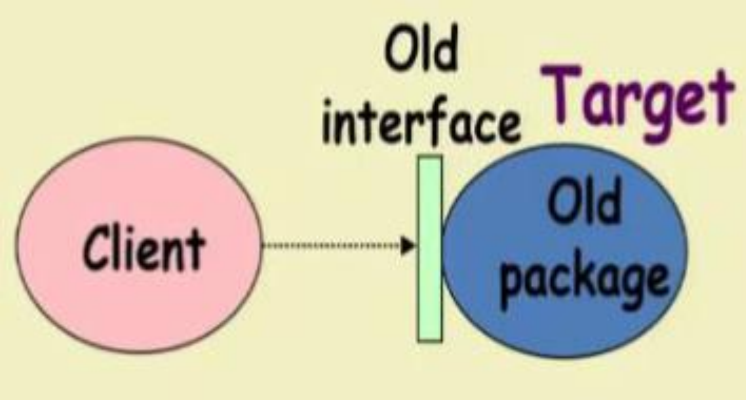
Adapter Pattern

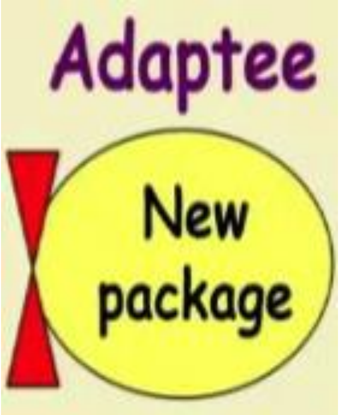
Adapter Pattern

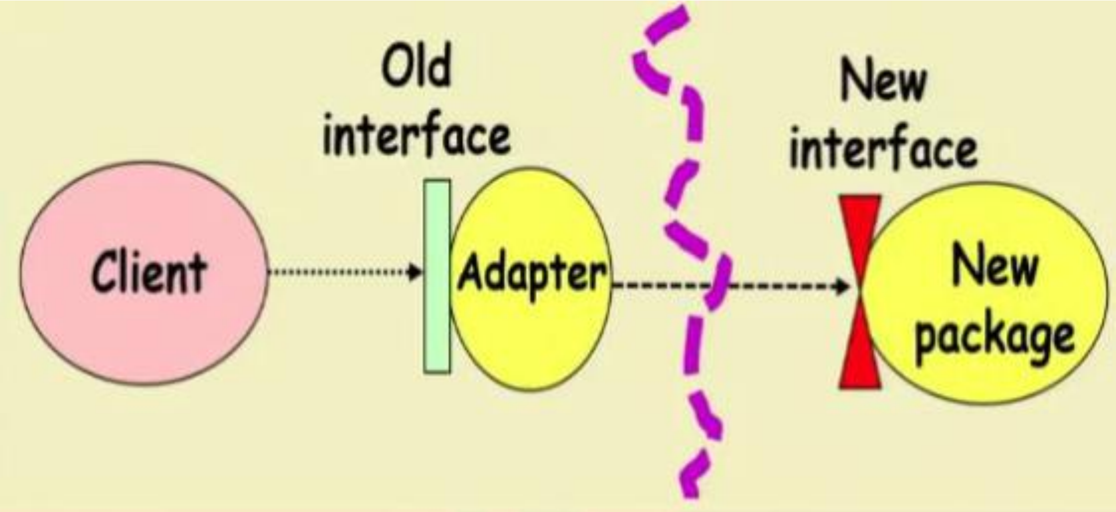
- Problem/intent: **Convert the interface of a class to the interface expected by the user of a class.**
  - Allow the classes to work together even when they have in-compatible interfaces

- Example: When we travel abroad (US/Europe), have an Indian electronic device, how to charge/use it in the US?
  - Use Adapters !!

Idea-Adapter pattern







Adapter Pattern

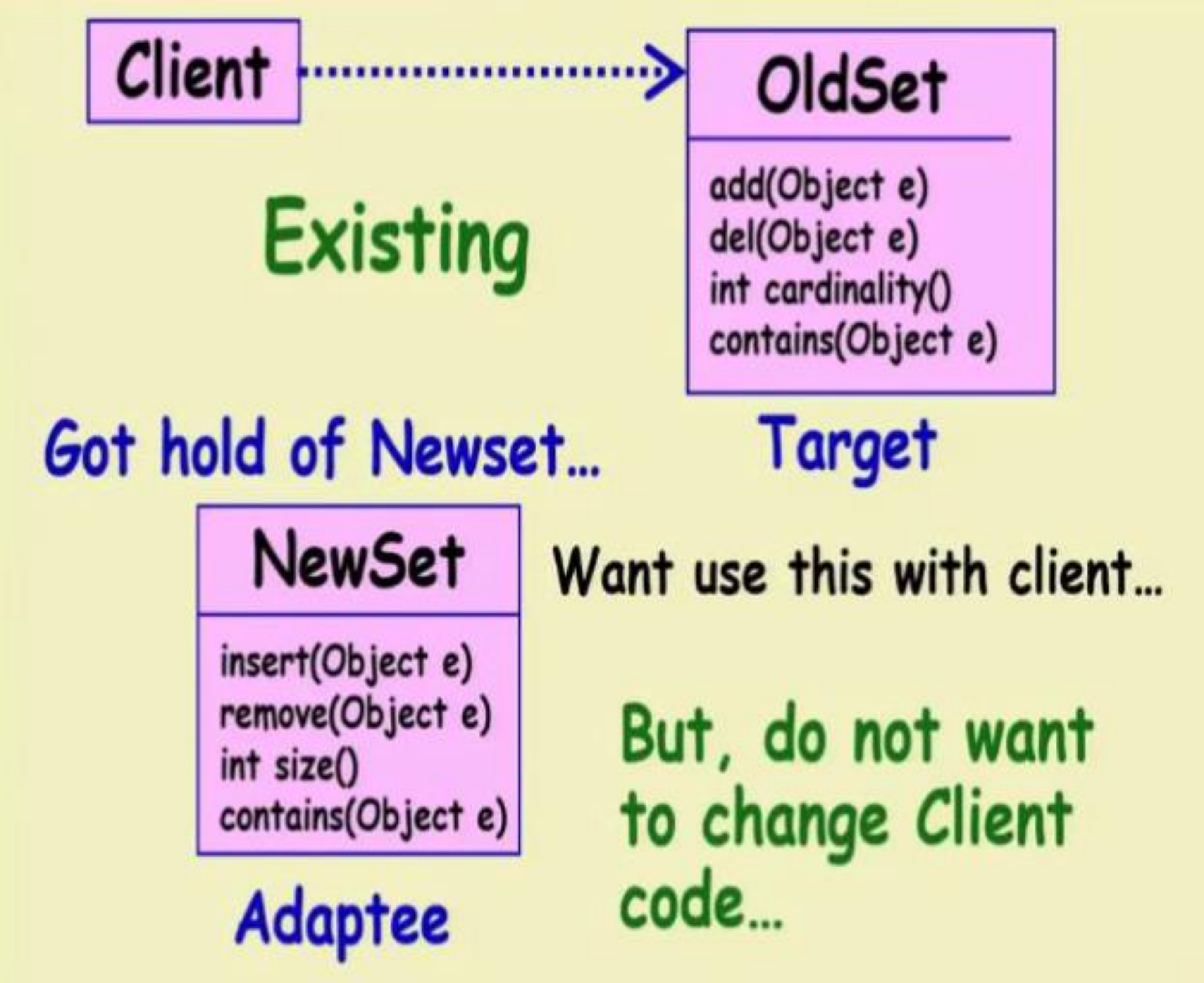
- **To allow two incompatible types to communicate**
  - E.g. when a client expects an interface that is not supported by the server class
  - **Adapter to act as a translator between the two types**

- Classes involved:
  - Target- interface that client uses
  - Adapter- to wrap the operations of the adaptee in interfaces familiar to the client
  - Adaptee- class with operations that client class desires to use

Adapter Pattern- Example

- Example- **Set implementation**
  - Consider an existing set implementation used by a client class (having poor performance)
  - There is another new efficient set class
    - **However**: has different interface
    - Do not want to change client code
  - Solution: **Design a set Adapter class**
    - With same interface as existing set
      - Which **translates to the new set's interface**

Adapter Pattern- Example



Client

OldSet

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

Existing

Target

Got hold of Newset...

NewSet

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

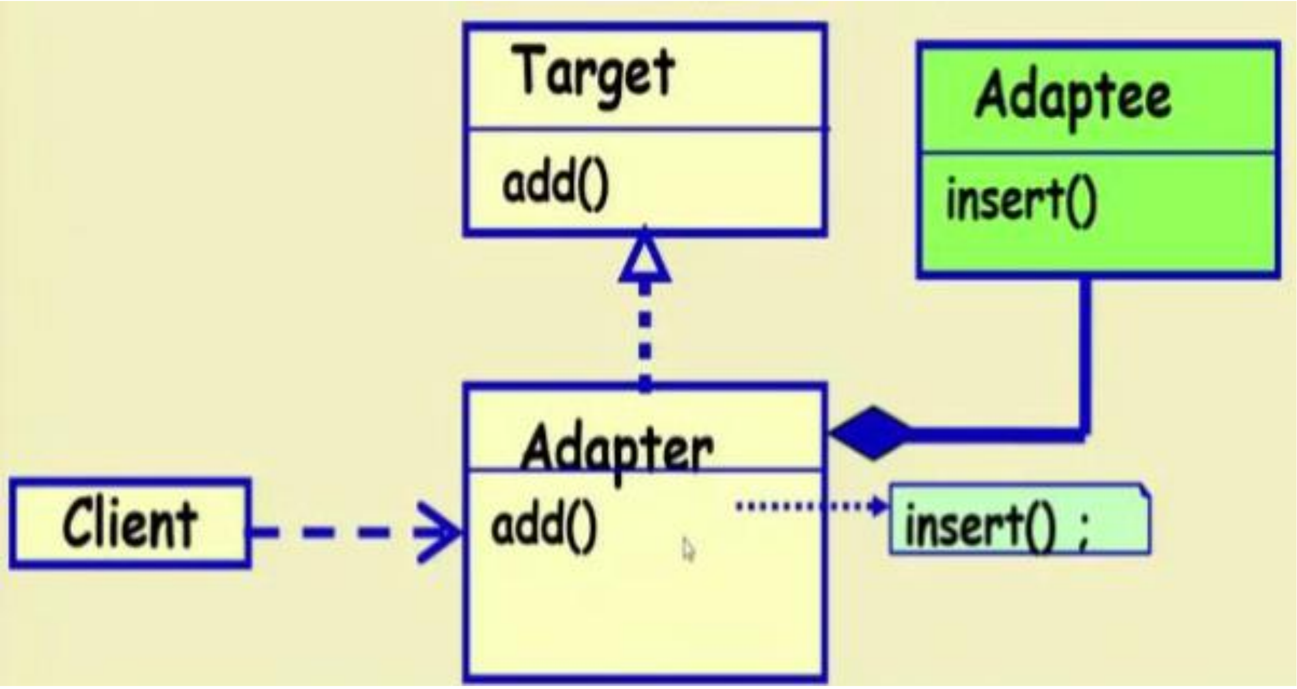
Adaptee

Want use this with client...

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

Adapter Pattern- Example

- Example- Idea.. Delegation !!
  - Adapter- hold an instance of the adaptee internally
  - Call adaptee operations from with operations supported by the target !



Adapter Pattern- Example

Client Code:

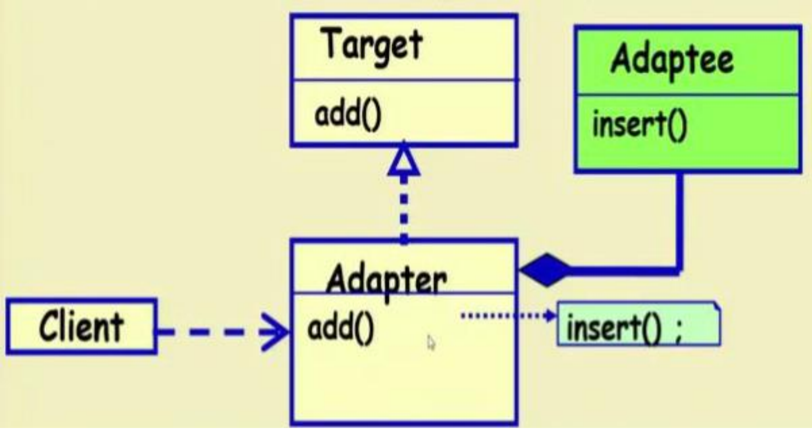
```
Adaptee a = new Adaptee( );
Target t = new Adapter (a);
Public void test1() {t.add( ); }
```

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();}
………………….
}
```



Singleton Pattern

Example Pattern: Singleton

Problem: Ensure that a class has **only one instance**

- Provide a global point of access to it

- Example: **An object to maintain application's configuration**
  - Many objects read and update config
  - A singleton class to ensure that all the objects of the application get the same copy of configuration

Example Pattern: Singleton

- Another example: A counter that gives unique numbers (e.g., token numbers in a hospital)
  - Counter needs to be unique
  - Singleton to generate numbers and synchronize

Solution: Singleton

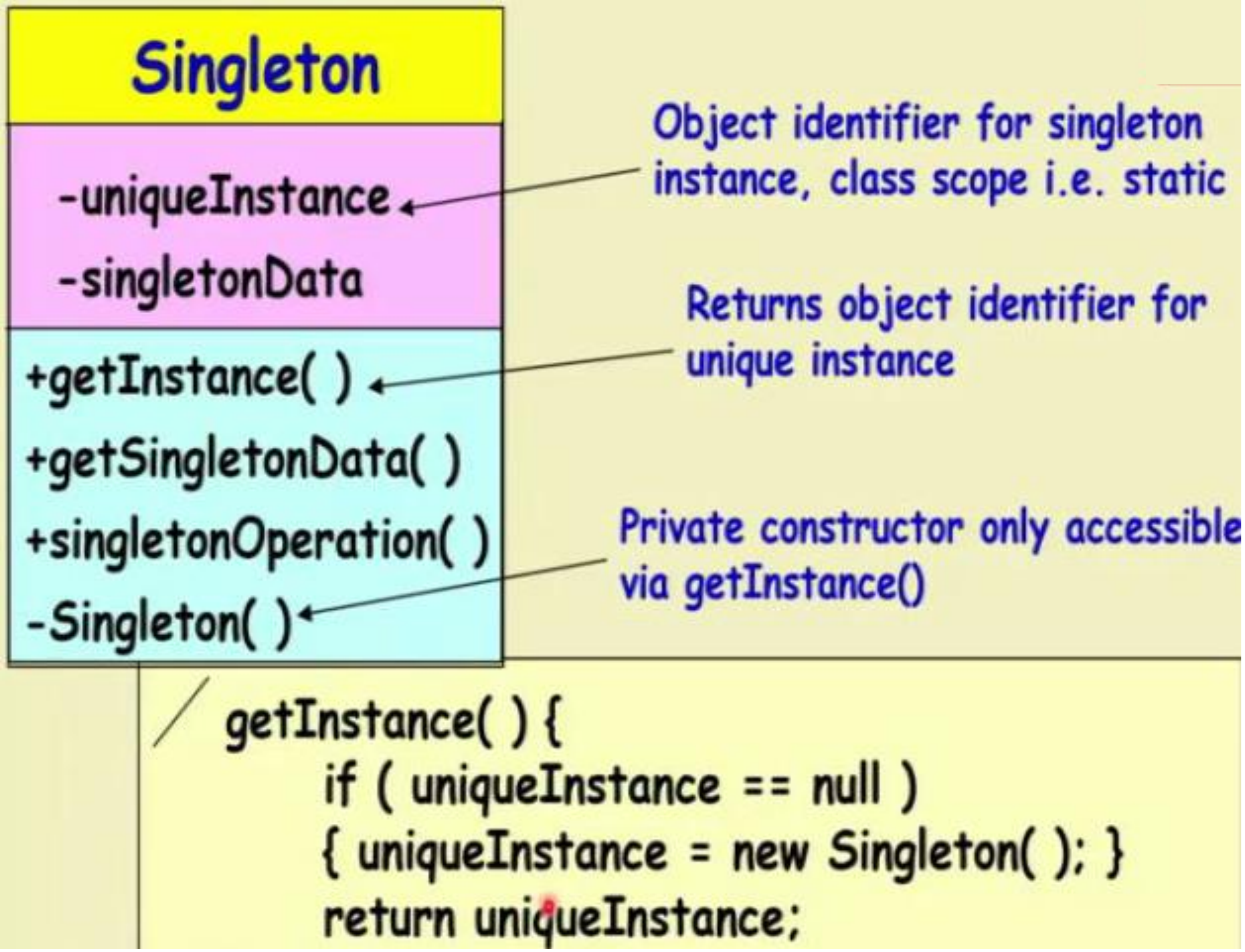
- Create an object with operation:
  - getInstance( )
- On first call,
  - Relevant object instance is created and object ID is returned
- On subsequent calls,
  - ID of existing object is returned (no new object created)

Solution: Singleton

- Singletons maintain a static reference to the singleton instance
  - Return a reference to that instance from the static instance method

- Singleton class itself responsible for keeping track of its sole instance

Solution: Singleton



Singleton

-uniqueInstance

-singletonData

+getInstance( )

+getSingletonData( )

+singletonOperation( )

-Singleton( )

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

Returns object identifier for unique instance

Private constructor only accessible via getInstance()

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

State Pattern

State Pattern

- **Behavioural pattern**
- **Problem: Object behaving differently to the same message.** Why this may happen?
  - Object perhaps has moved to a ***different state***

Example:

- Consider the response to "***renew***" request for the same book:
  - Renewed successfully
  - Already renewed 3 times, cannot be renewed
  - Reserved, cannot be renewed

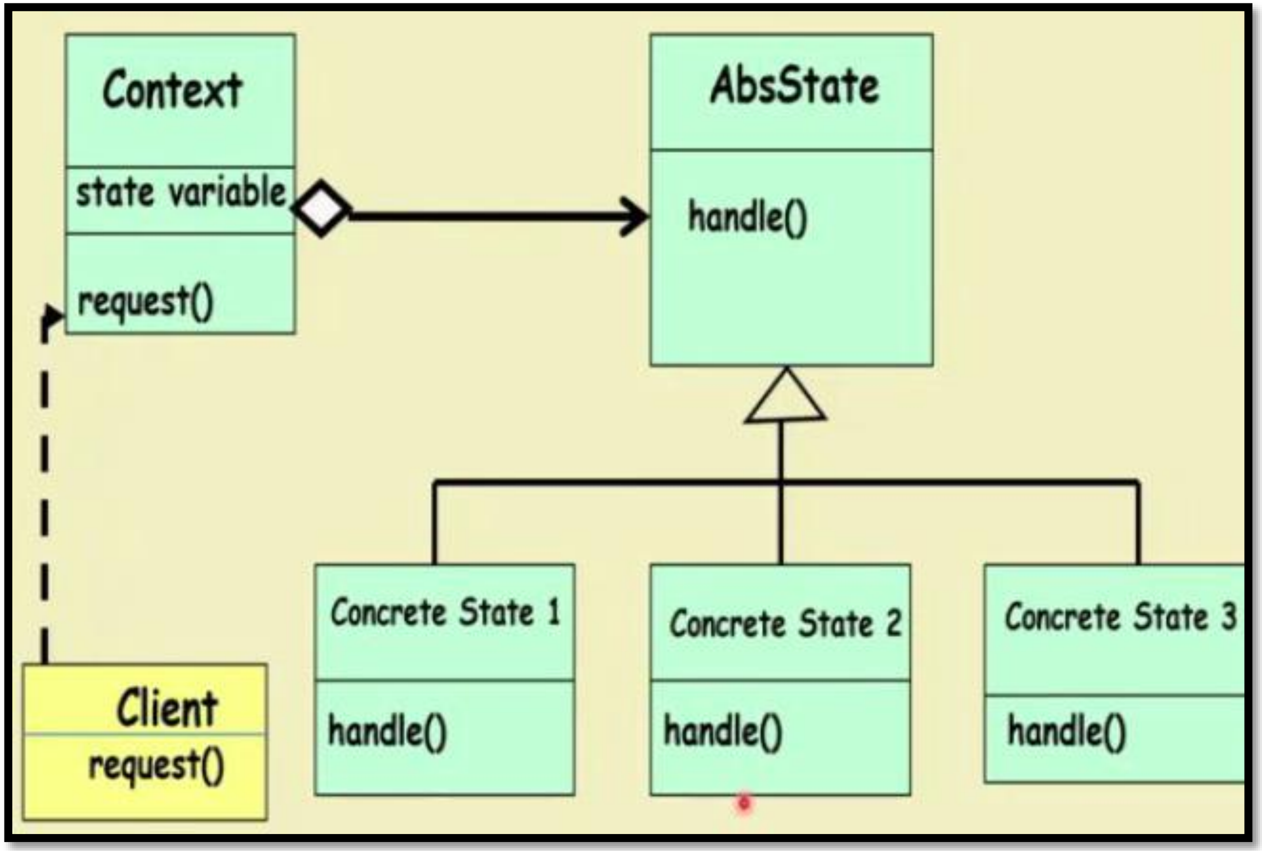
State of an Object

- One or more attributes of a class act as **state variable**.

- Depending on the values of the state variable
  - Some methods exhibit different behavior

- **Example**: Renew book related method

State pattern structure



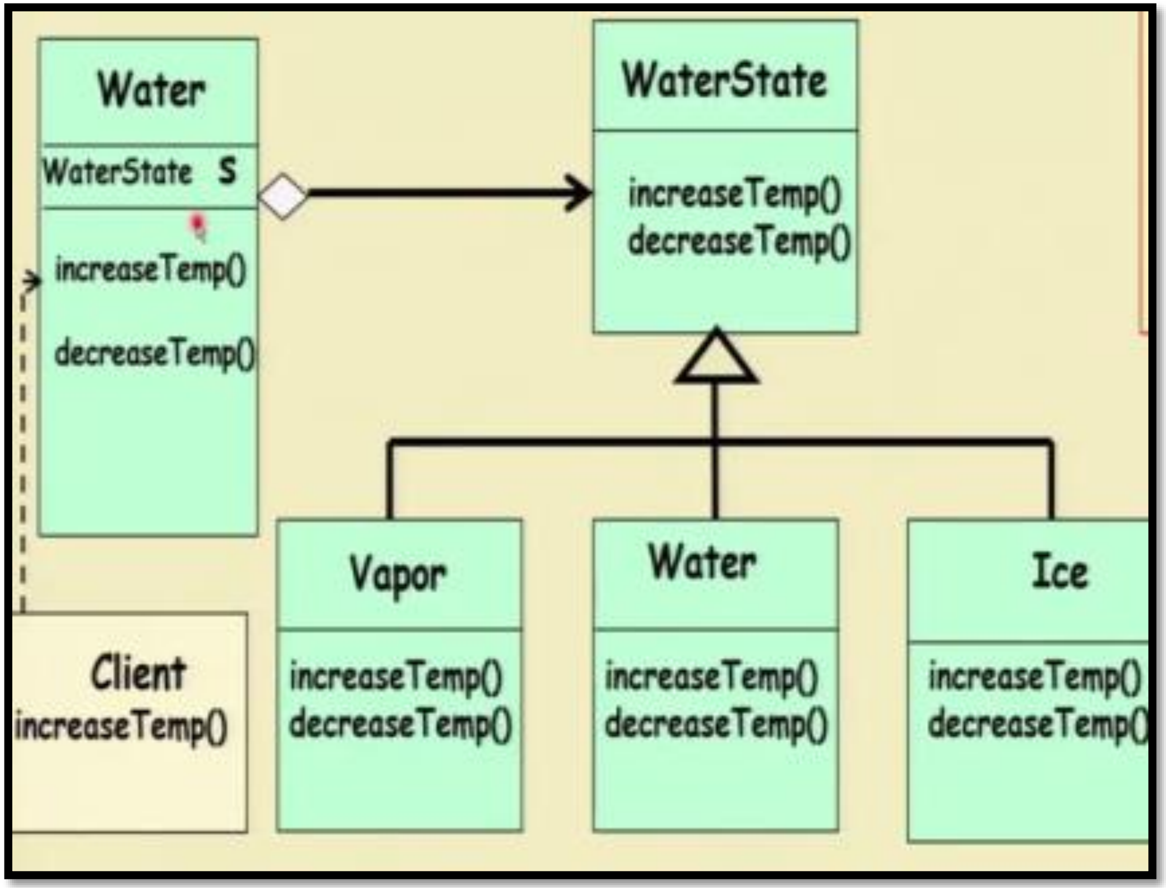
Allows object to alter its behavior (binding to a different methods), when its internal state changes.

*An **object-oriented state machine***

State pattern structure

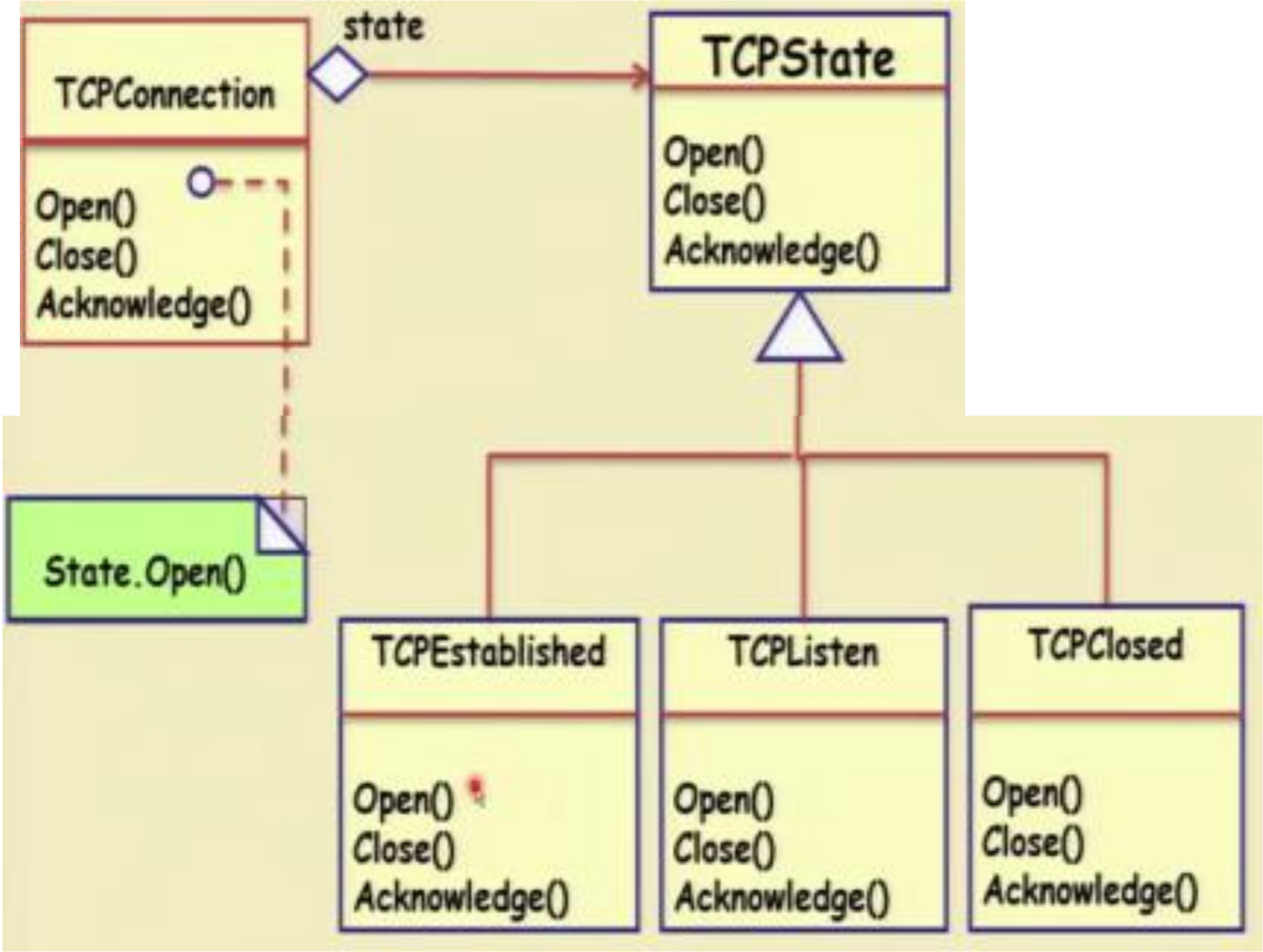
- State pattern: solution to the problem of how to make behavior depend on state.
- "***context***" class: present a single interface to the outside world.
- Define a State abstract base class.
- **different "states"** of the state machine as **derived classes** of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

State pattern- Example



- Behavior of water- depending on its state

State pattern- Example (TCP connection)



- **TCP connection- Responds differently to requests at different states**

State Pattern- Pros/Cons

- Advantages:
  - Behavior of a state encapsulated into an object
  - Avoids inconsistent state- state change occurs using one object.
  - Code becomes modular

- Disadvantage:
  - increased number of objects

Multiple other patterns….

- Iterator pattern
- Proxy pattern
- Decorator pattern
- Bridge pattern
- ……