Software Design Methodologies and Testing

(Subject Code: 410449)
(Class: BE Computer Engineering)
2012 Pattern

Objectives and outcomes

- Course Objectives
  - To understand and apply different design methods and techniques
  - To understand architectural design and modeling
  - To understand and apply testing techniques
  - To implement design and testing using current tools and techniques in distributed, concurrent and parallel
  - Environments
- Course Outcomes
  - To present a survey on design techniques for software system
  - To present a design and model using UML for a given software system
  - To present a design of test cases and implement automated testing for client server, distributed, mobile applications

Other Information

- Teaching Scheme Lectures:
  - 3 Hrs/Week
- Examination Scheme
  - In Semester Assessment: 30
  - End Semester Assessment : 70

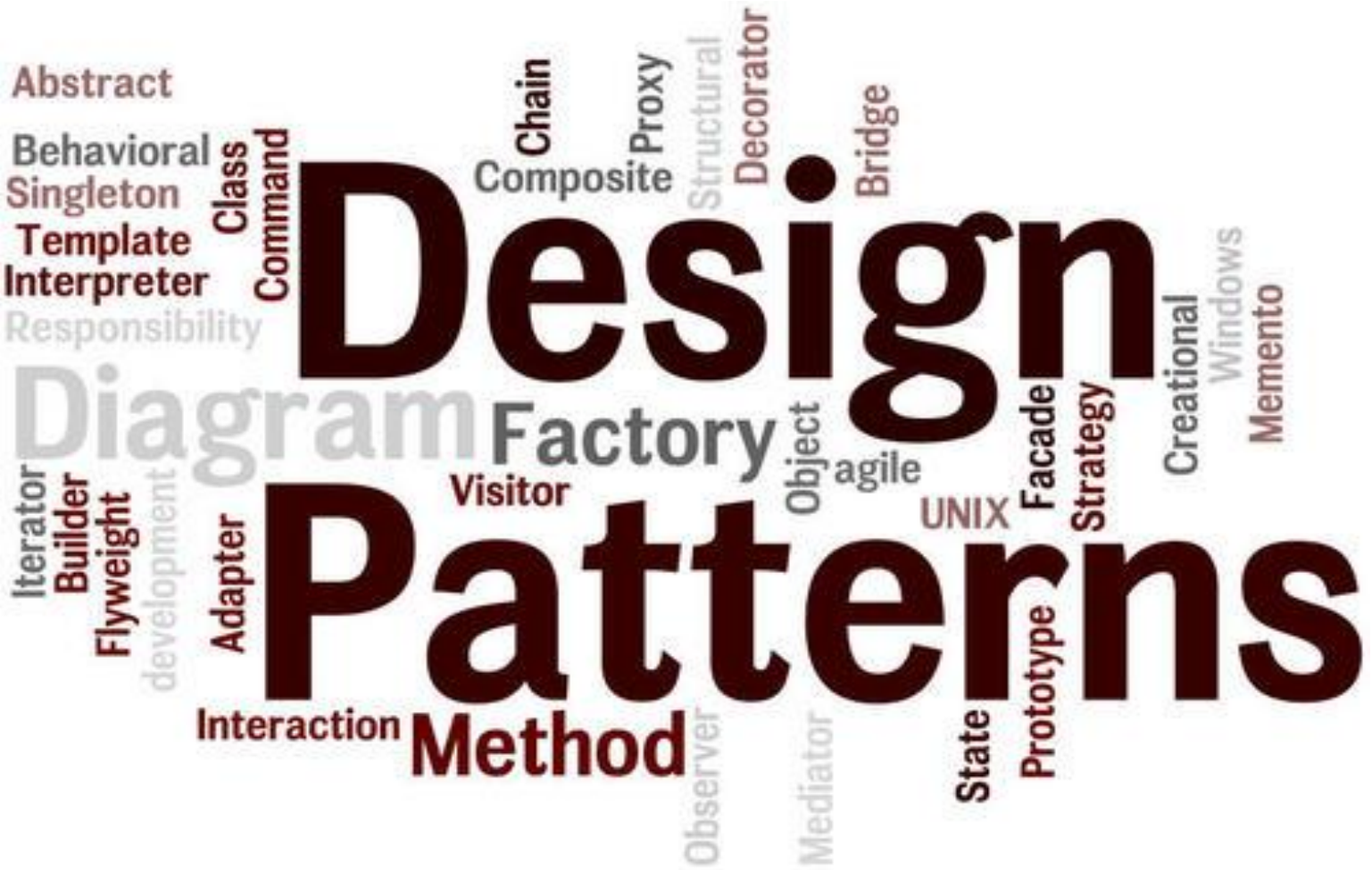
UNIT-III

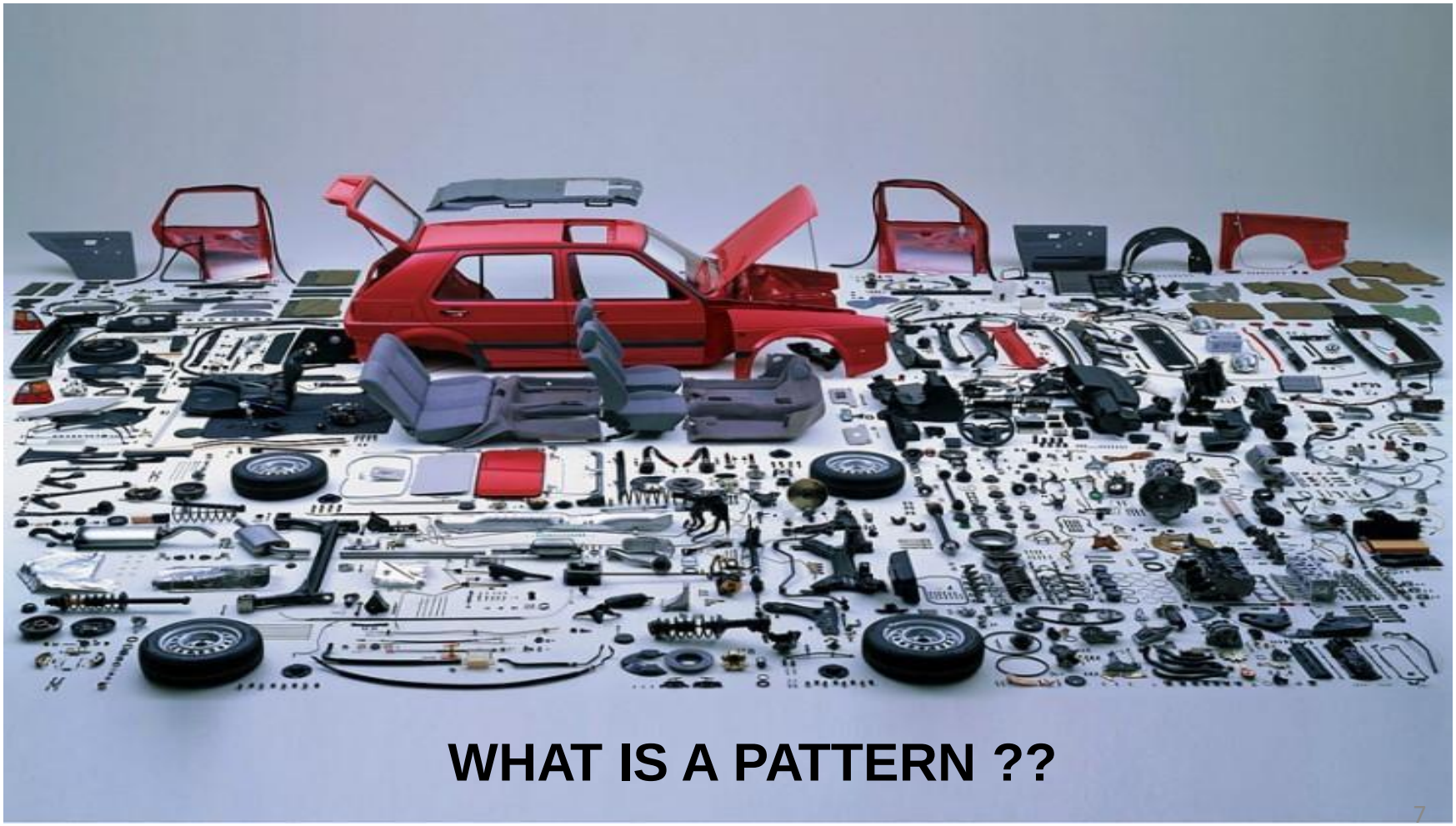
Design Pattern

DesignPatterns;Introduction,creational,Structural and behavioral patterns, singleton, proxy, adapter, factory,I terator,observer pattern with application

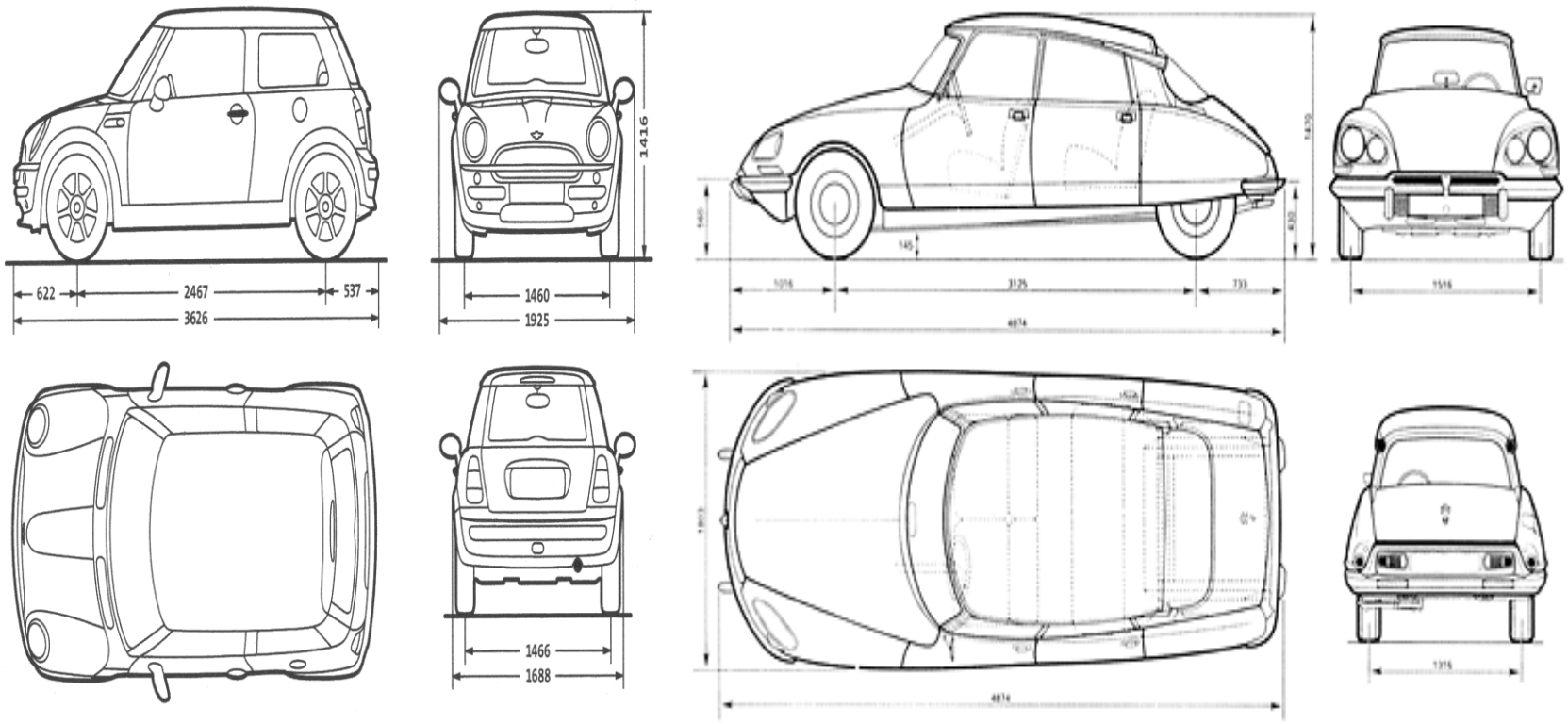
UNIT-III
Design Pattern

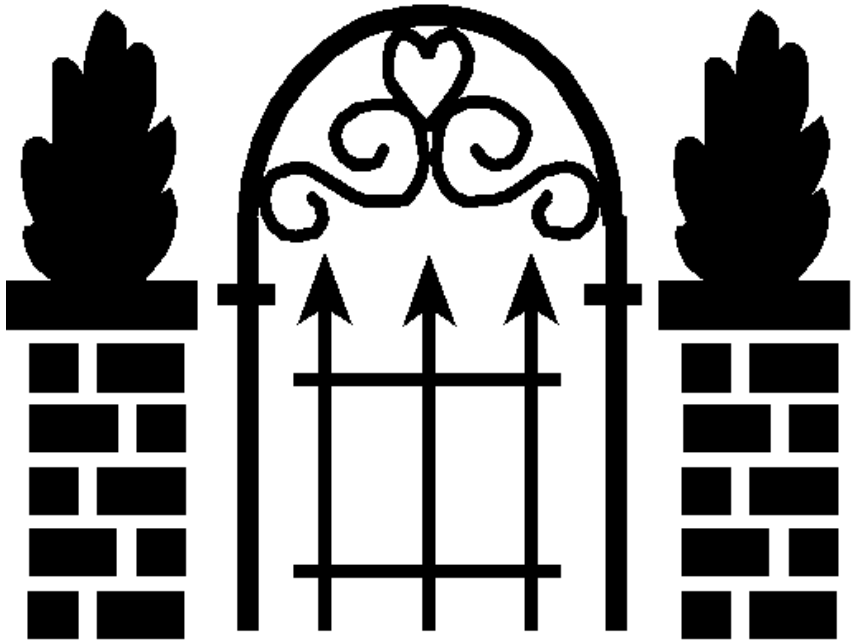
Design Patterns

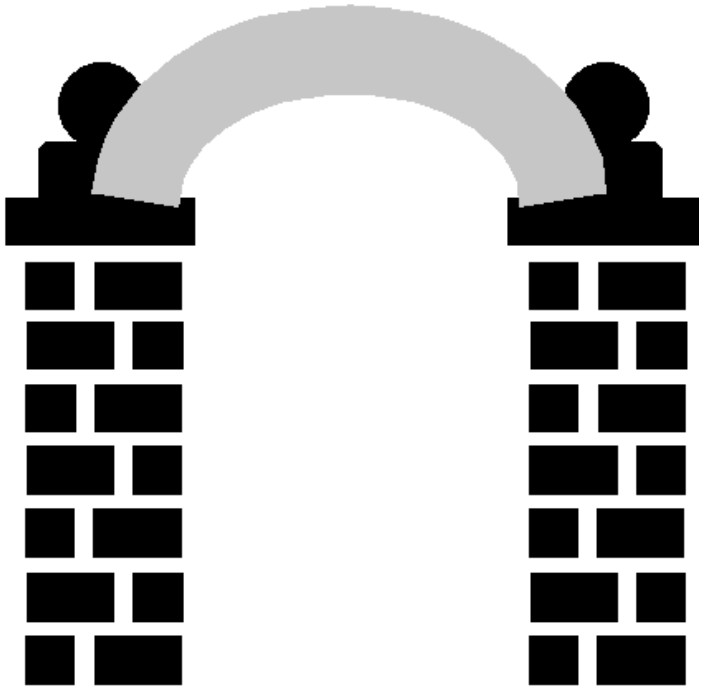




WHAT IS A PATTERN ??


622
2467
537
3626
1460
1925
1416
1466
1688
3125
733
4874
1516
1375





Structures may look different but still solve a common problem.

“Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over.”

1. **Fundamental Design Patterns**
   - A. Interface
   - B. Container
   - C. Delegation
2. **Architectural Patterns**
   - A. Model View Controller (MVC)
3. **Structural Design Patterns**
   - A. Facade
   - B. Decorator
   - C. Proxy
   - D. Adapter
4. **Creational Design Patterns**
   - A. Factory Method
   - B. Abstract Factory
   - C. Objectpool
   - D. Singleton
5. **Behavioral Design Patterns**
   - A. Iterator
   - B. Observer
   - C. Event Listener
   - D. Strategy

A.Fan, Bulb Example

A.Rinse & Repeat philosophy

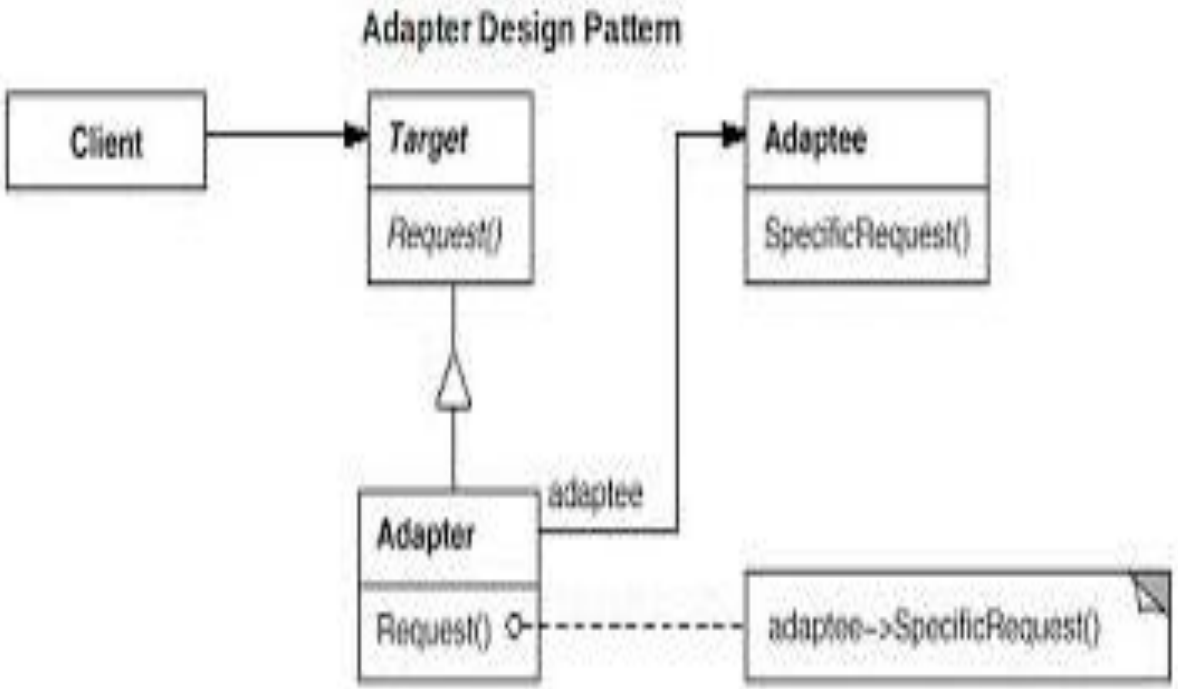
- Concerned with how classes and objects are composed to form larger objects.
- Types:
- Adapter
- Facade
- Proxy
- Decorator

.Convert the interface of a class into another interface expected by the client.

- Used to provide new interface to existing objects.
- Also known as wrapper.

Need 9v to charge

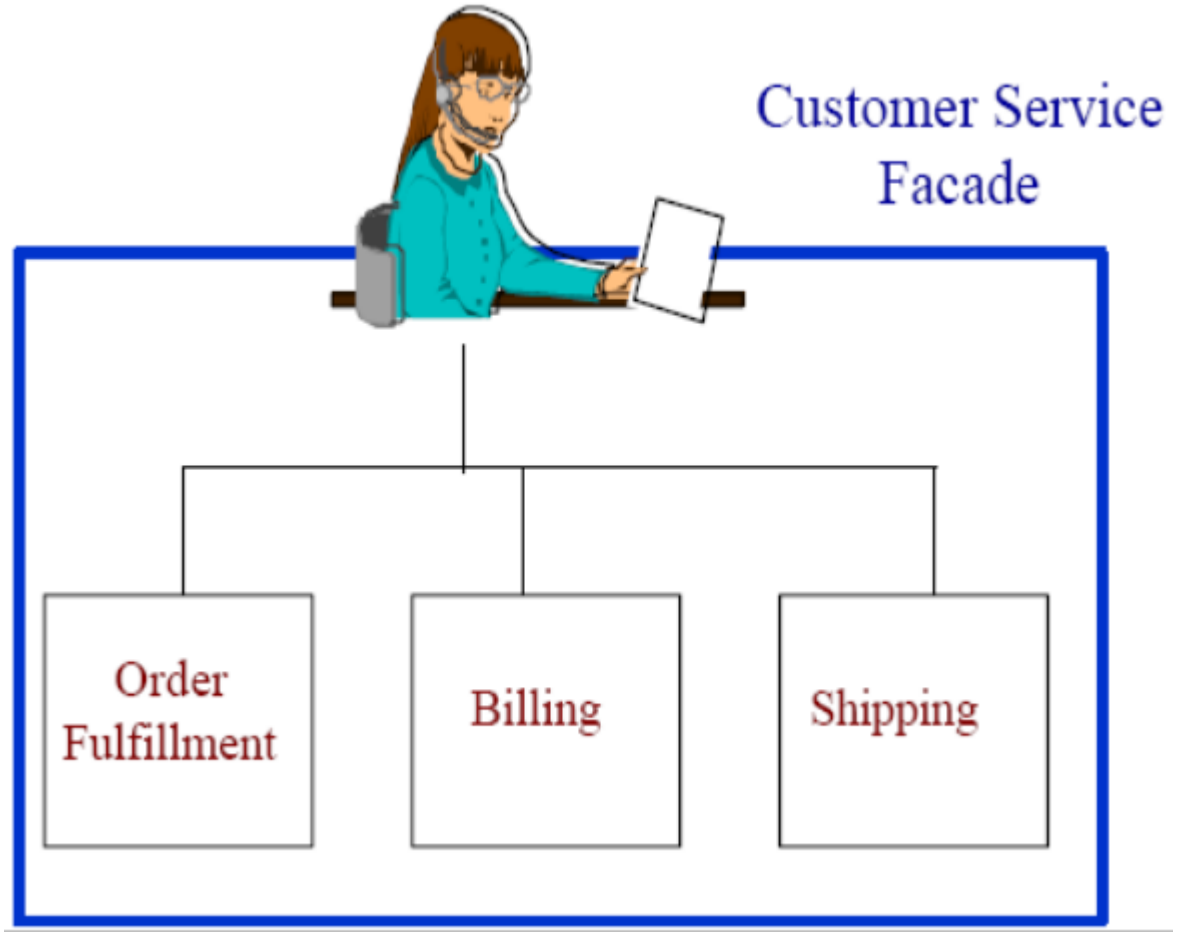
Produce 220v


Adapter Design Pattern
Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()


Java module
Java Native Interface (JNI)
C++ module

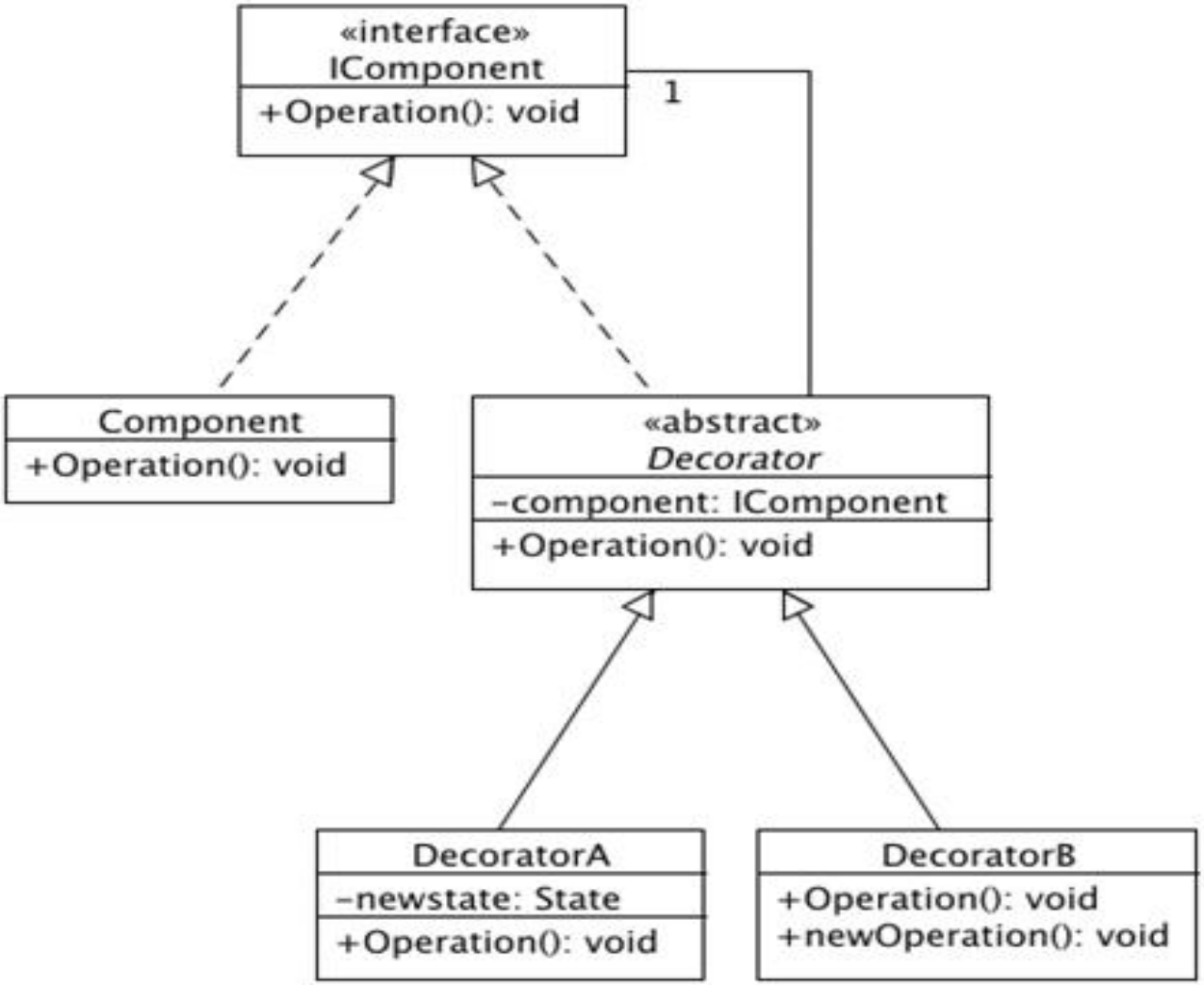
- Provides different interface to existing objects.
- Adapter makes two existing interfaces work together as opposed to defining entirely new one.

Facade shows how to make a single object represent an entire subsystem. It carries out its responsibility by forwarding messages to the objects it represents.



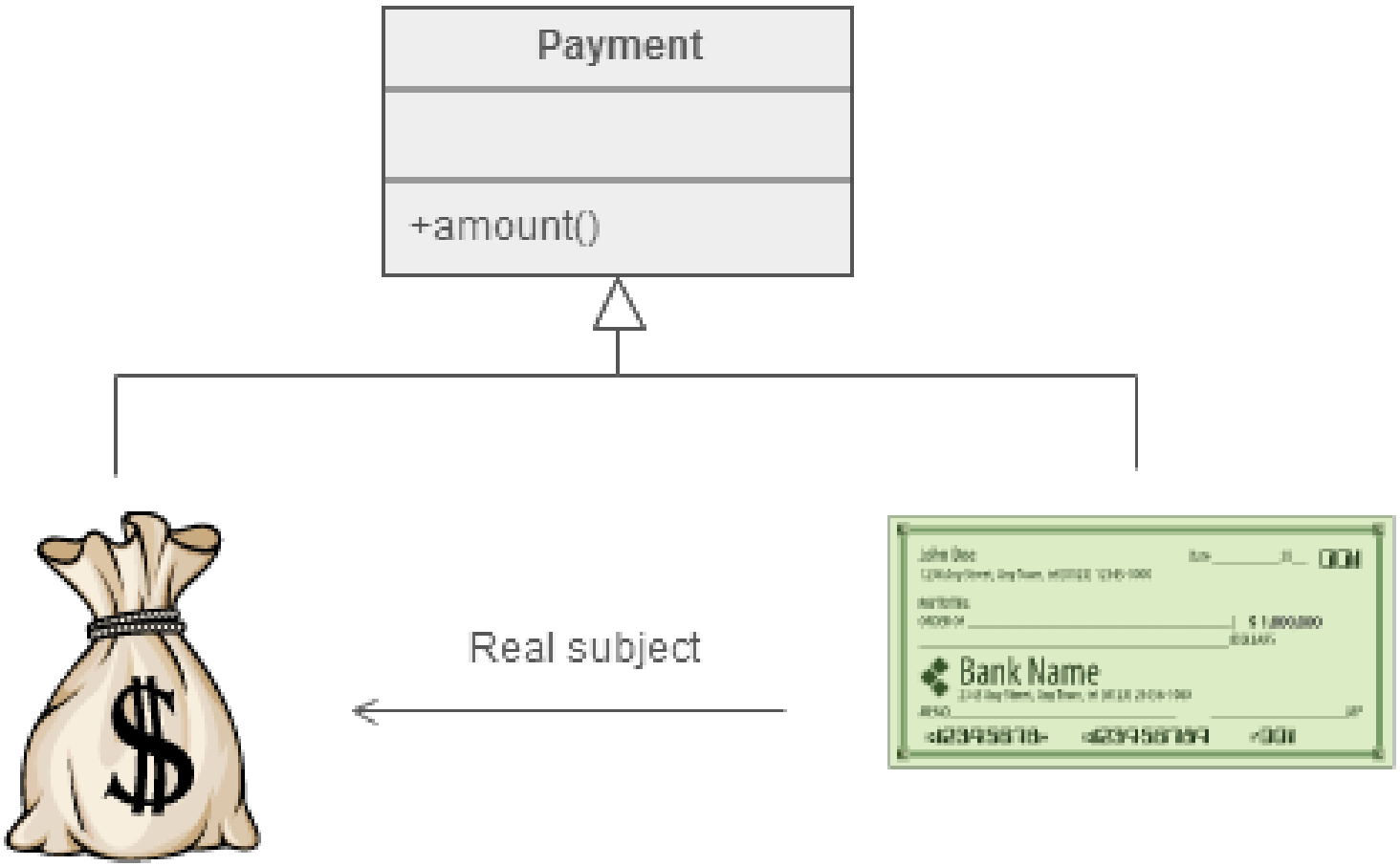
Provide a unified interface to a set of interfaces in a subsystem.

- What is Decorator?
- Decorator allows to modify an object dynamically.
- Rather than rewrite old code you can extend with new code.


«interface»
IComponent
+Operation(): void
1
Component
+Operation(): void
«abstract»
Decorator
-component: IComponent
+Operation(): void
DecoratorA
-newstate: State
+Operation(): void
DecoratorB
+Operation(): void
+newOperation(): void

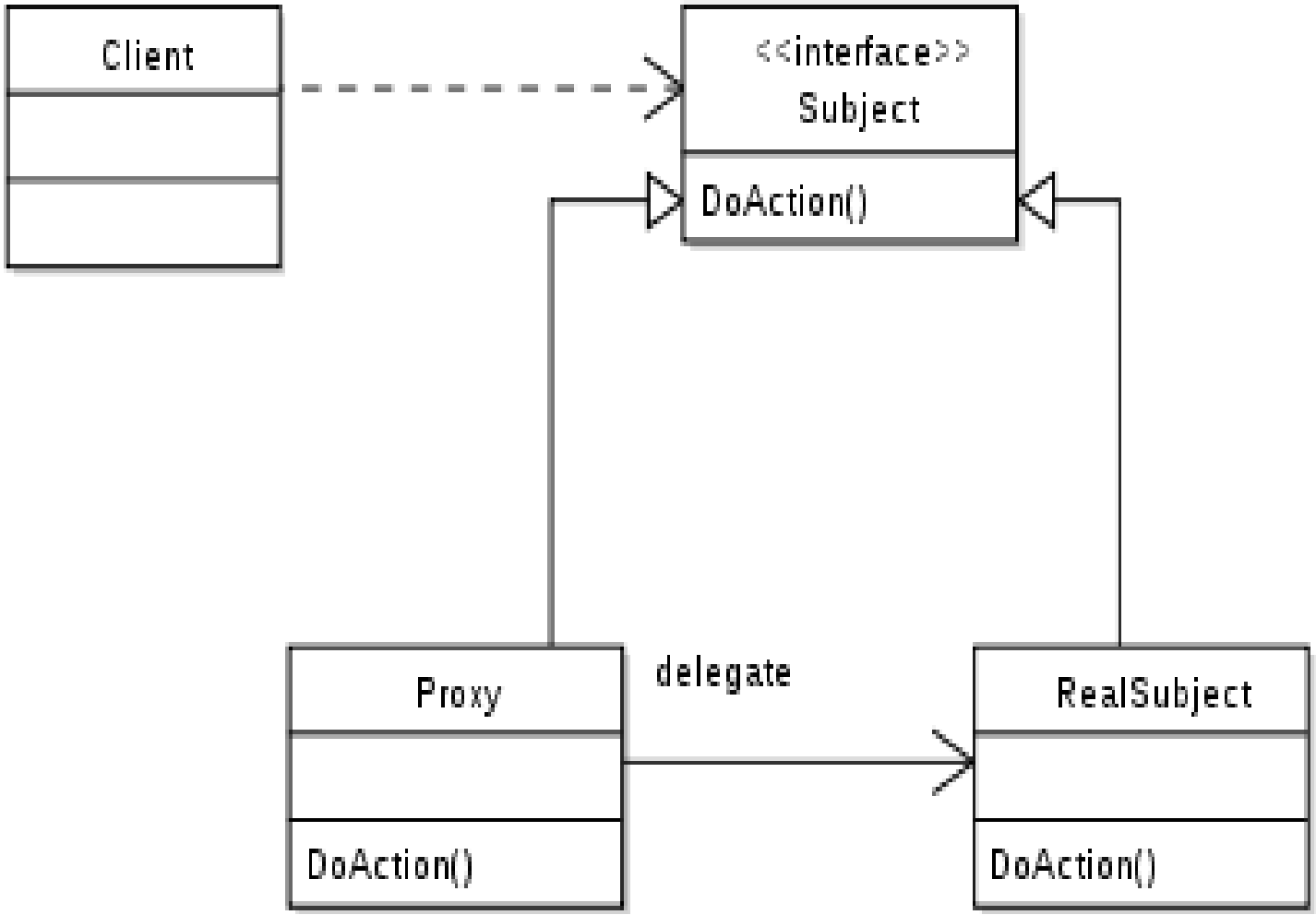
- add responsibilities to individual objects transparently.
- remove these responsibilities again without affecting other objects.

Proxy acts as a placeholder for another object. A level of indirection is introduced hence a Proxy can be used in different ways - it can restrict, enhance or alter properties of the object it represents.

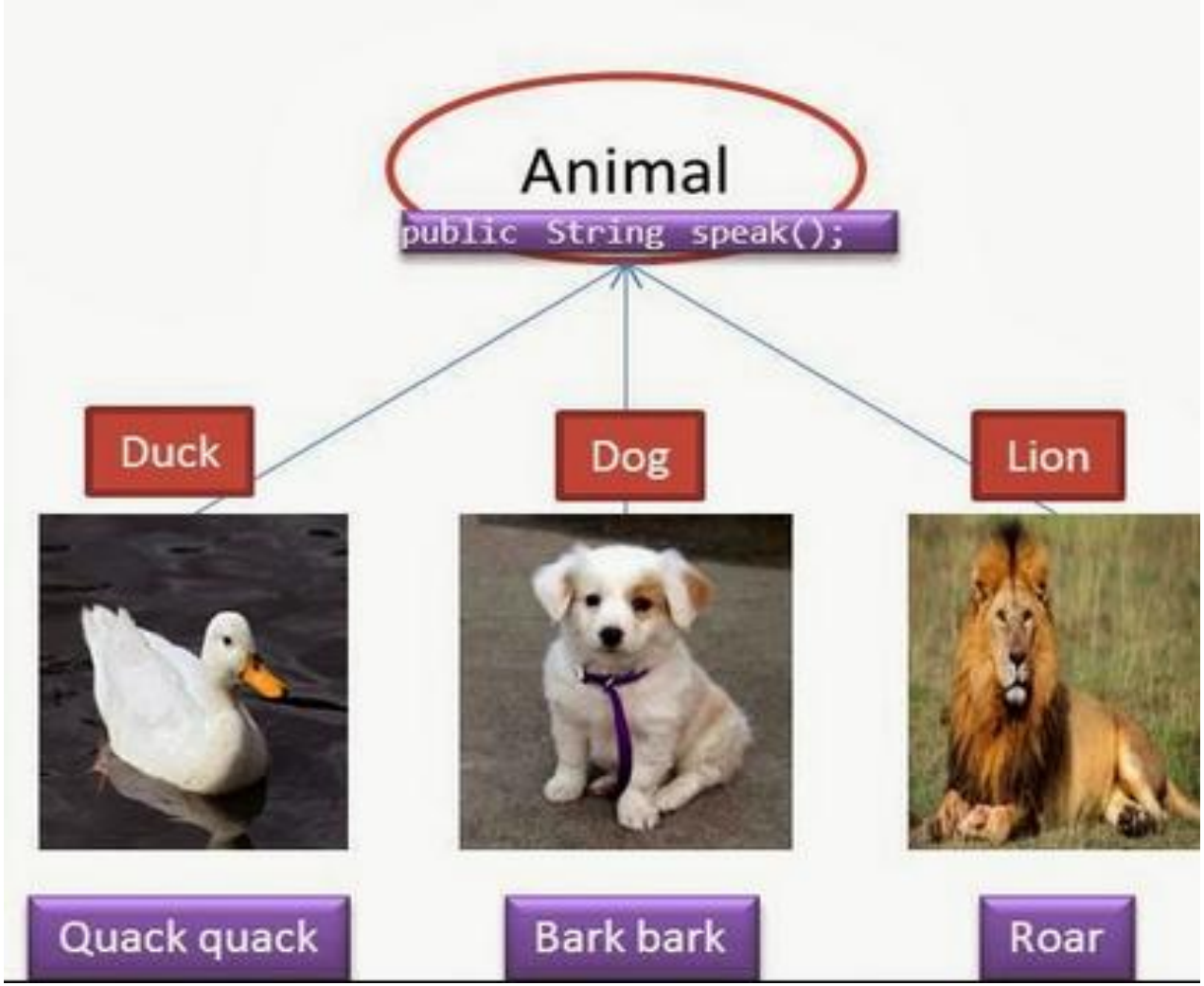


FundsPaidFromAccount

CheckProxy

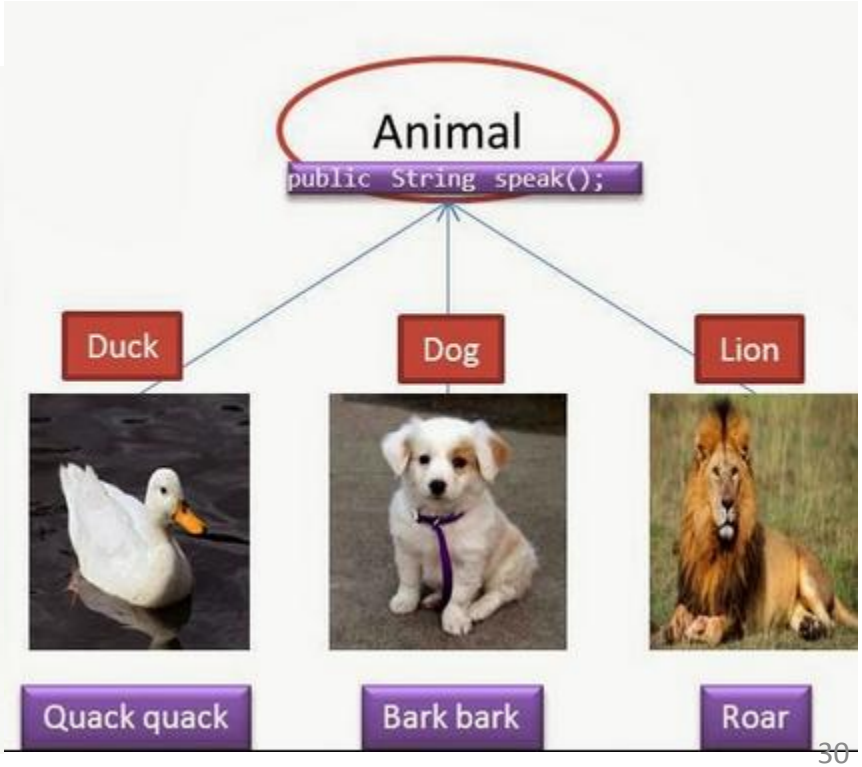

Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

- Factory Method
- Abstract Factory
- Object pool
- Singleton


Animal
public String speak();
Duck
Dog
Lion
Quack quack
Bark bark
Roar

Animal factory

```
public Animal getAnimal(String animalType)
{
   Animal animal =null;
   if ("dog".equals(animalType))
   {
     animal = new Dog();
   }
  else if("duck".equals(animalType))
  {
    animal = new Duck();
   }
  else if("lion".equals(animalType))
  {
    animal = new Lion();
   }
   return animal
}
```

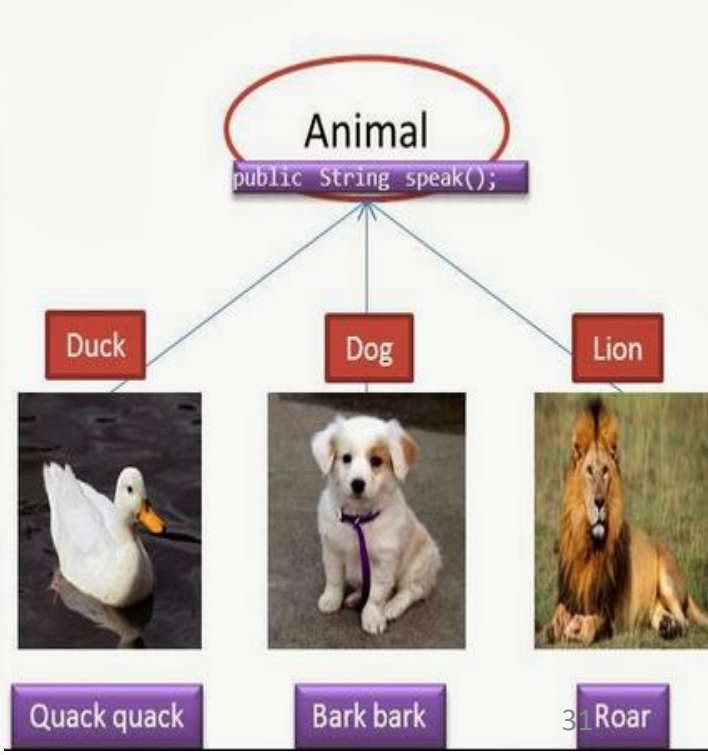


Animal Factory

Client

```
AnimalFactory animalFactory=new AnimalFactory ();
Animal animal = animalFactory.getAnimal("dog")
animal.speak();
```

```
public Animal getAnimal(String animalType)
{
   Animal animal =null;
   if ("dog".equals(animalType))
   {
     animal = new Dog();
   }
  else if("duck".equals(animalType))
  {
    animal = new Duck();
   }
  else if("lion".equals(animalType))
  {
    animal = new Lion();
   }
   return animal
}
```



100 java files

```
Duck duck = new Duck();
duck.speak();
```

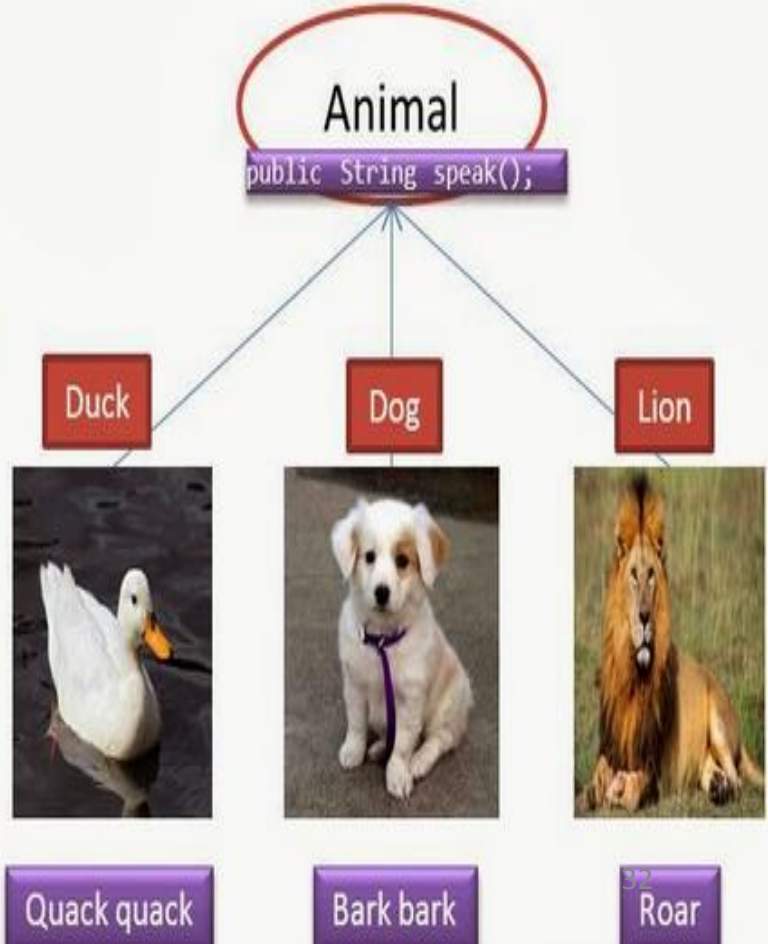
```
Duck duck = new Duck(4);
duck.speak();
```

Client

```
AnimalFactory animalFactory=new AnimalFactory ();
Animal animal = animalFactory.getAnimal("dog")
animal.speak();
```

AnimalFactory

```
public Animal getAnimal(String animalType)
{
   Animal animal =null;
   if ("dog".equals(animalType))
   {
     animal = new Dog();
   }
   else if("duck".equals(animalType))
   {
     animal = new Duck();
   }
   else if("lion".equals(animalType))
   {
     animal = new Lion();
   }
   return animal
}
```



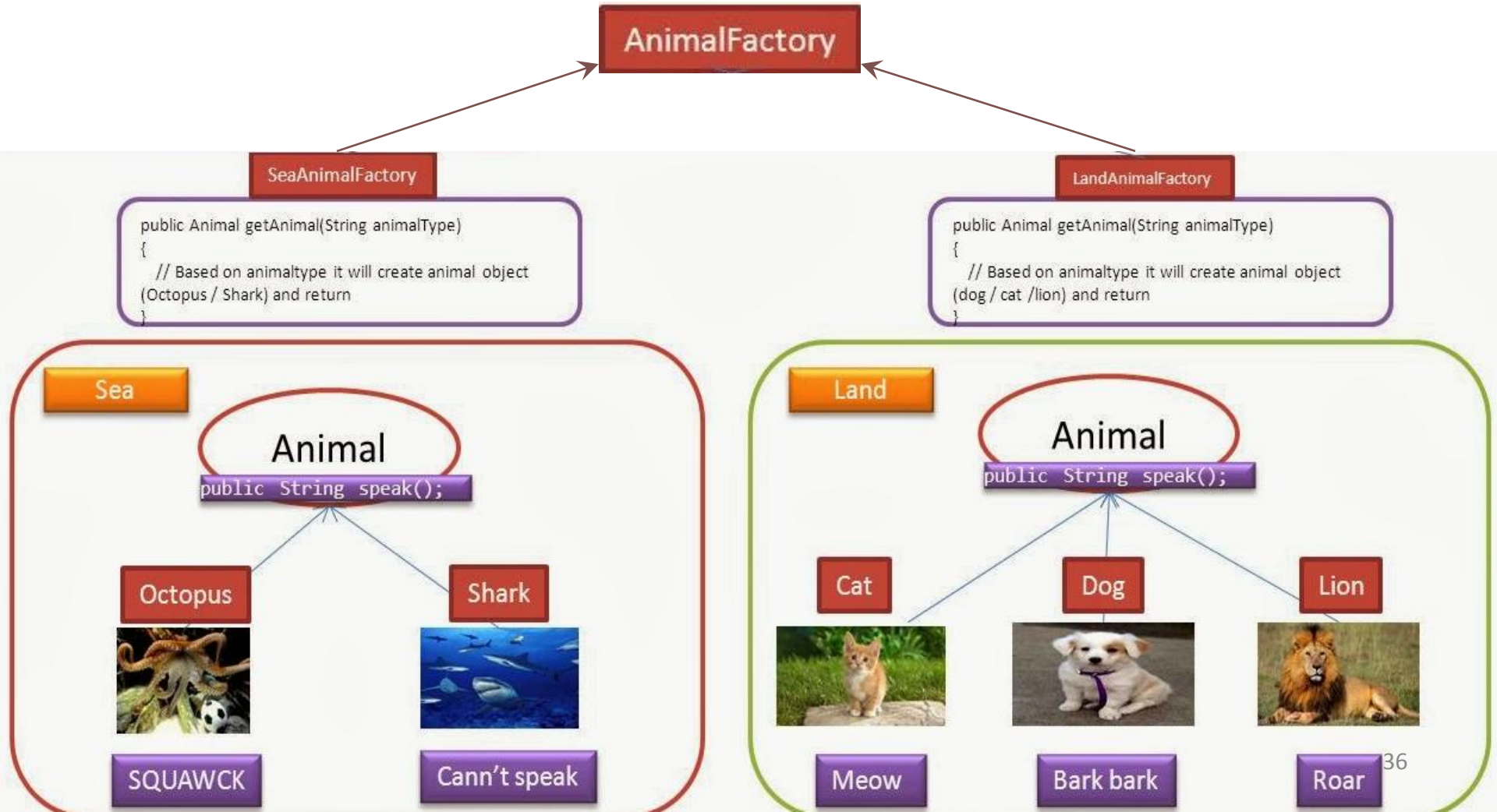
- In Factory pattern, we create object without exposing the creation logic and refer to newly created object using a common interface.

- In simple words, if we have a super class and n sub-classes, and based on data provided, we have to return the object of one of the sub-classes, we use a factory pattern.

- The basic principle behind this pattern is that, at run time, we get an object of similar type based on the parameter we pass.

- If object creation code is spread in whole application, and if you need to change the process of object creation then you need to go in each and every place to make necessary changes.

When to use Factory Pattern?

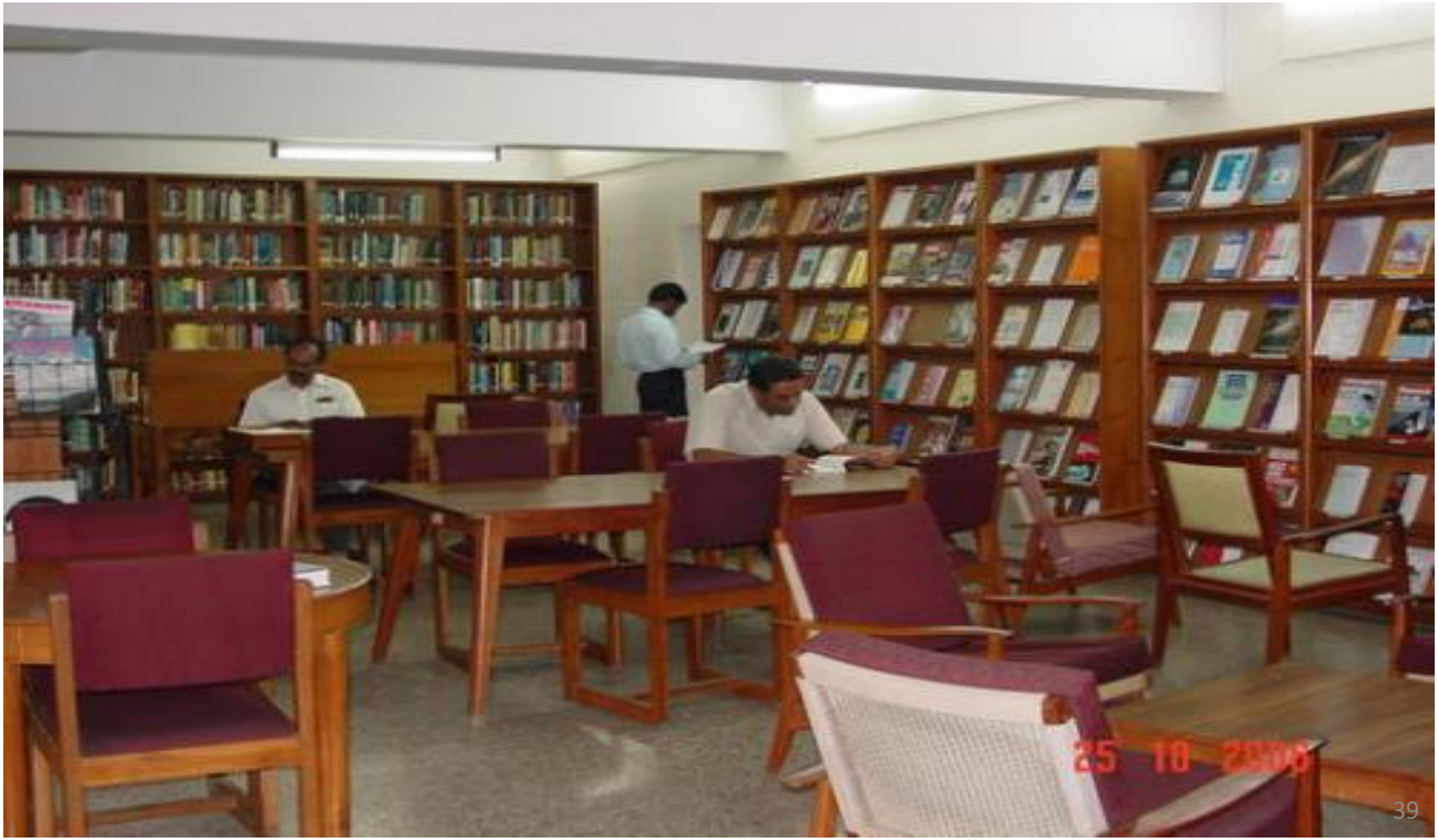

AnimalFactory
SeaAnimalFactory
public Animal getAnimal(String animalType)
{
// Based on animaltype it will create animal object
(Octopus / Shark) and return
}
LandAnimalFactory
public Animal getAnimal(String animalType)
{
// Based on animaltype it will create animal object
(dog / cat /lion) and return
}
Sea
Animal
public String speak();
Octopus
Shark
SQUAWCK
Cann't speak
Land
Animal
public String speak();
Cat
Dog
Lion
Meow
Bark bark
Roar

Abstract Factory pattern is a super-factory which creates other factories. This factory is also called as Factory of factories.

- When the objects are no longer needed by the processes , they are released to the pool.

- Object pool lets us to reuse the objects that are released into object pool.

- We can instantiate the new objects instead of waiting for the release of objects.



1. Several parts of the application requires the same object at different parts of the program.

2. Program periodically needs objects which are very expensive to create.

- Most of the times we need single object of the class to maintain the originality of the class.
- Class itself is given the responsibility of taking care of single object creation.

- The Abstract Factory and Builder patterns can use Singletons in their implementation.

1. There must be exactly one instance of a class, and it must be accessible for many clients via a known access point.

2. The sole instance should be extensible by sub-classing , hence clients should be able to use an extended instance without changes in their code.

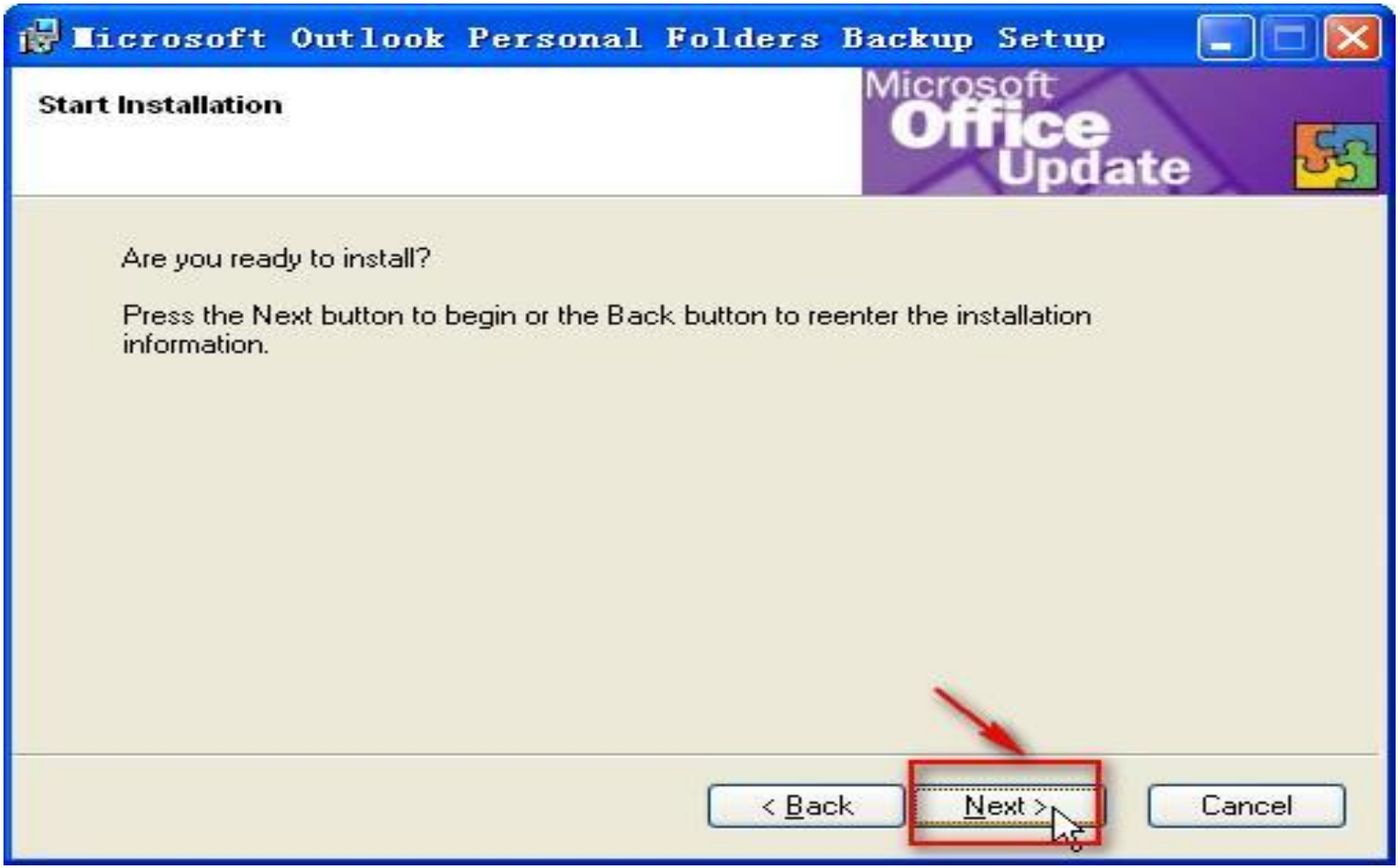
- Behavioral Design Patterns are design patterns that identify common communication patterns between objects and realize these patterns. - WIKIPEDIA

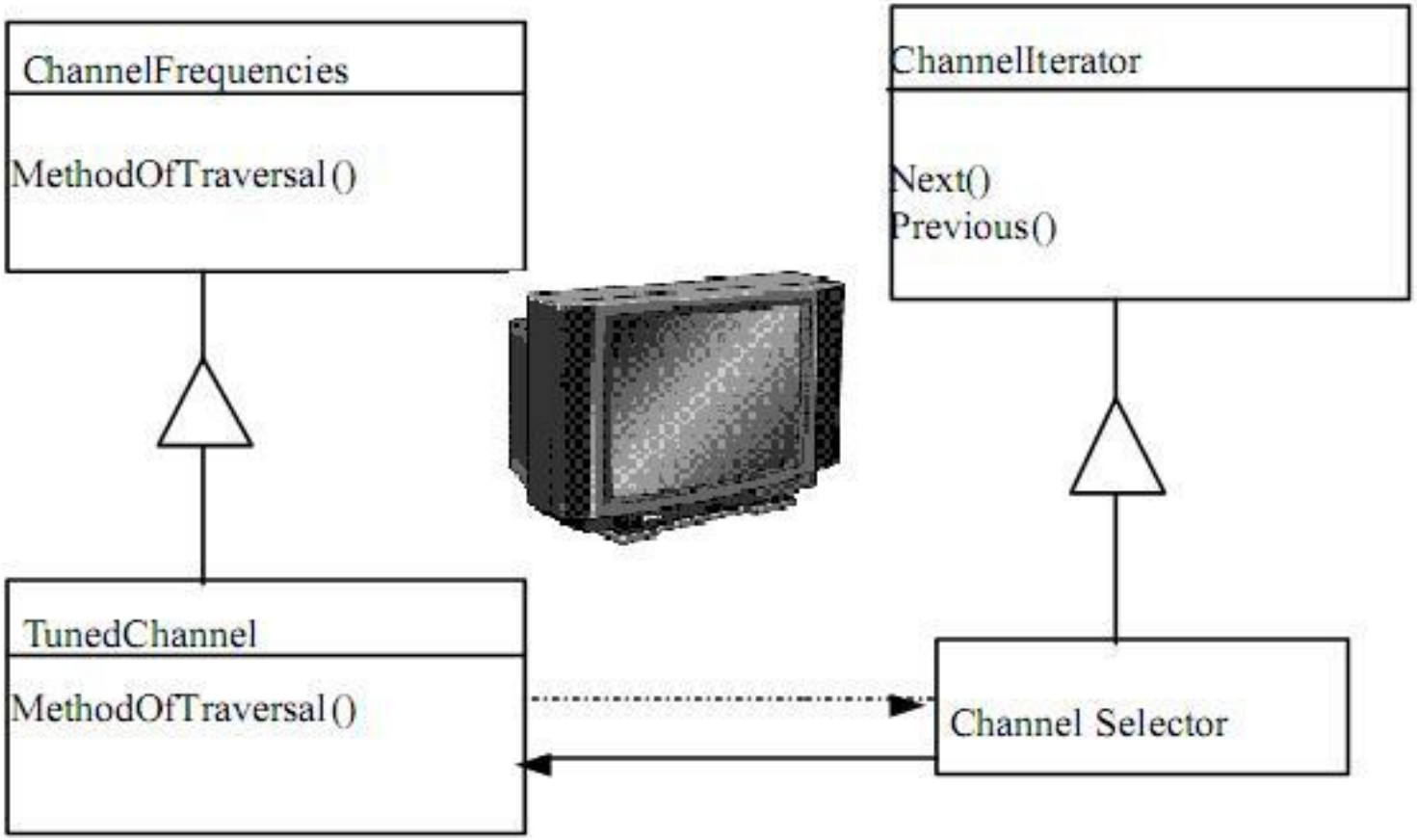
- Are concerned with algorithms and the assignment of responsibilities between objects

- Iterator
- Chain of responsibility
- Observer
- Command
- Strategy

- Problem: Clients that wish to access all members of a collection must perform a specialized traversal for each data structure.

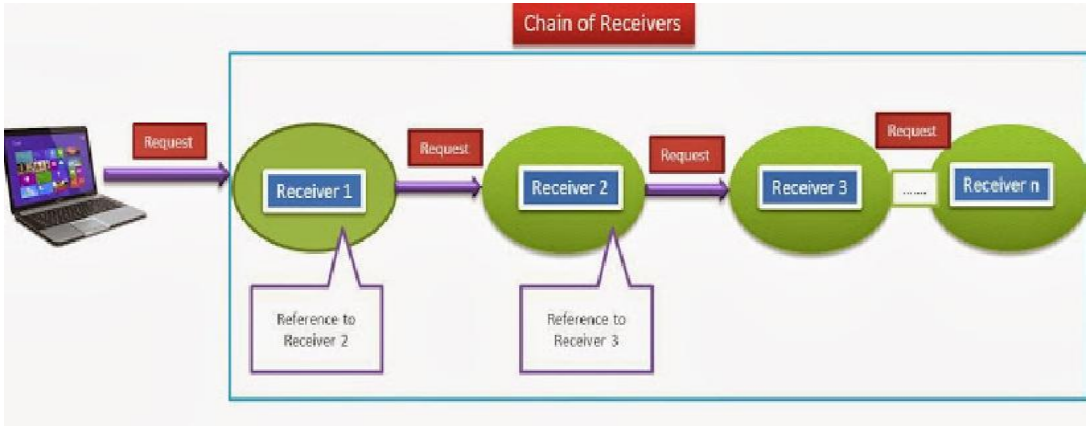
- Solution: Implementations perform traversals. The results are communicated to clients via a standard interface.
- Encapsulation

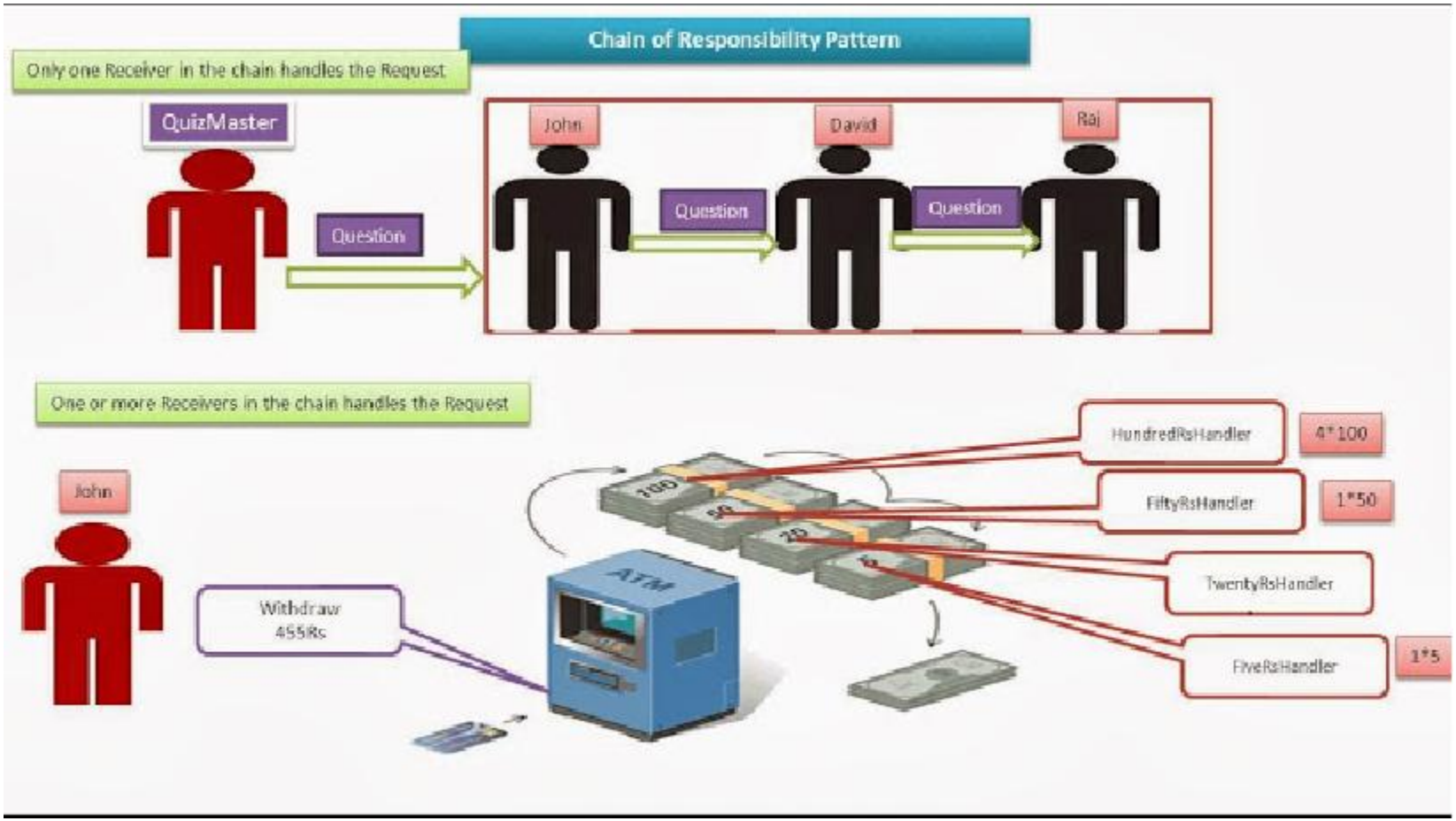

Microsoft Outlook Personal Folders Backup Setup
Start Installation
Microsoft Office Update
Are you ready to install?
Press the Next button to begin or the Back button to reenter the installation information.
< Back
Next >
Cancel

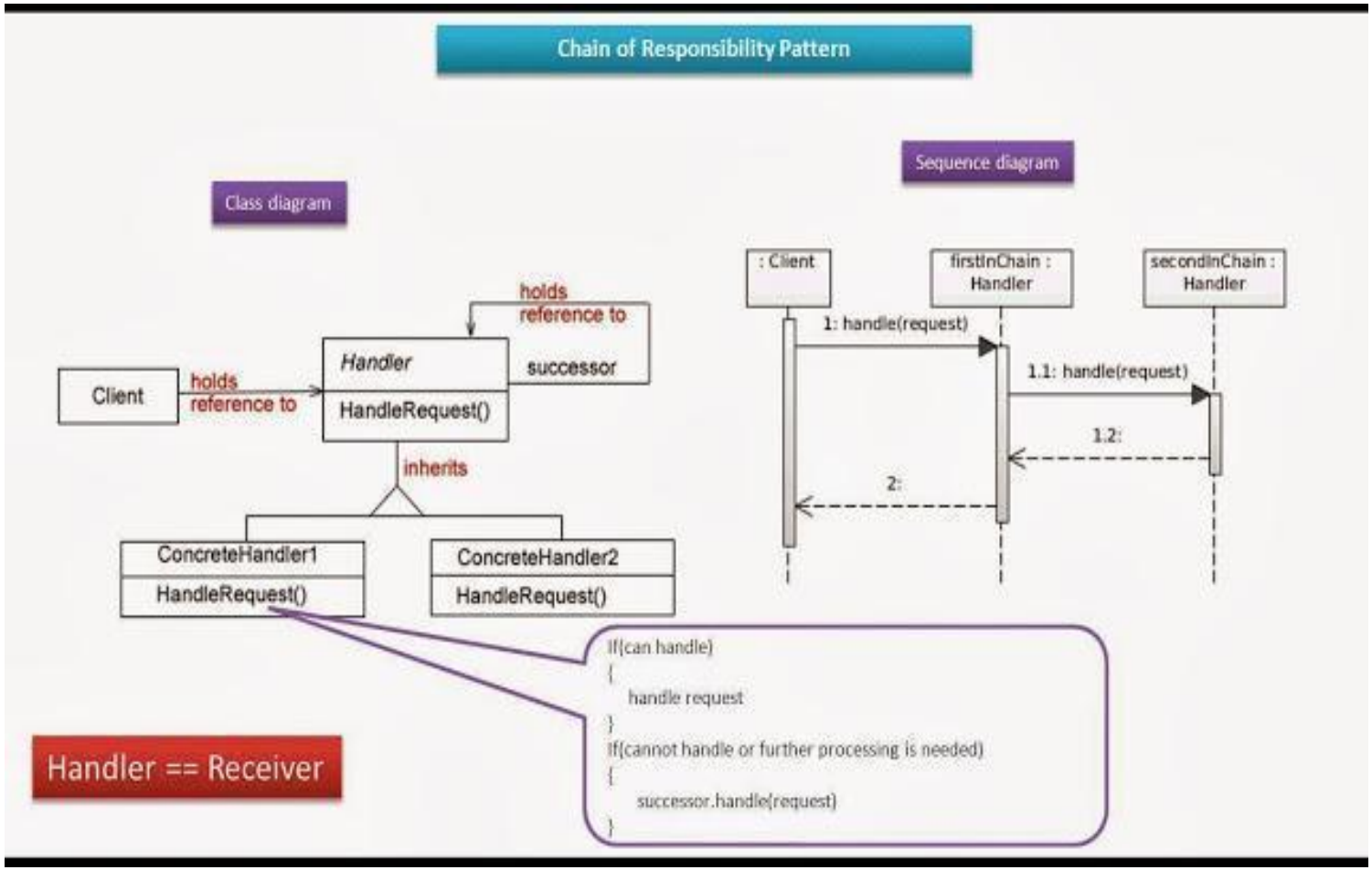

ChannelFrequencies
MethodOfTraversal()
ChannelIterator
Next()
Previous()
TunedChannel
MethodOfTraversal()
Channel Selector

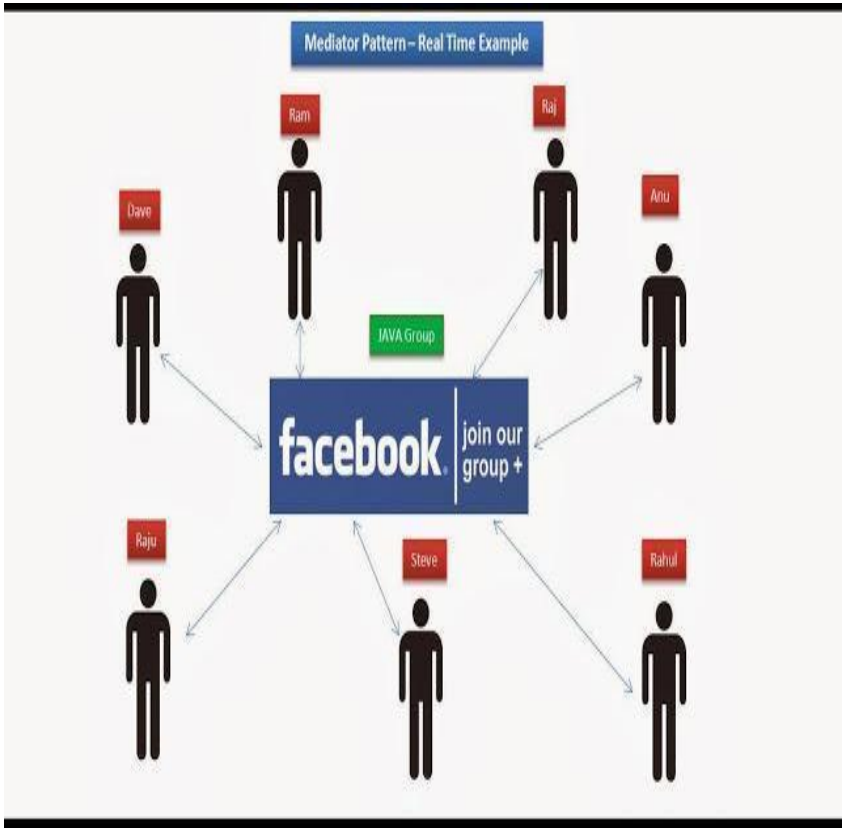
.We have different objects that can do the job but we do not want the client object know which is actually doing it.

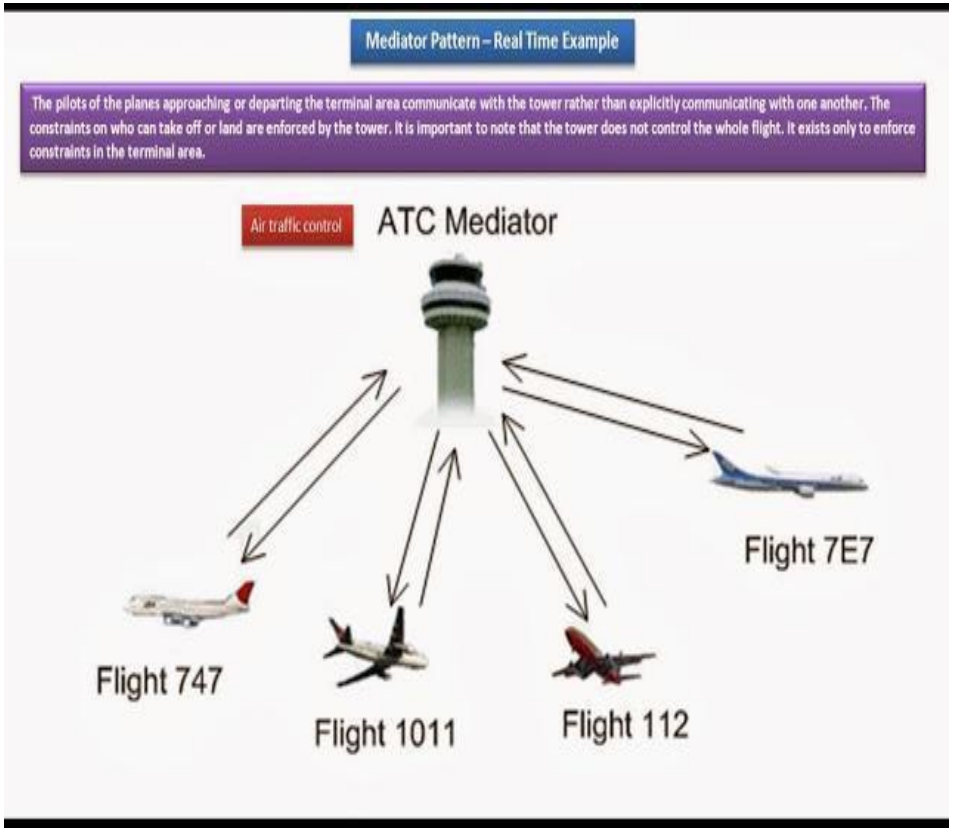
●It created a chain of receiver objects for a request.




Chain of Responsibility Pattern
Only one Receiver in the chain handles the Request
QuizMaster
Question
John
Question
David
Question
Rai
One or more Receivers in the chain handles the Request
John
Withdraw
455Rs
ATM
HundredRsHandler
4*100
FiftyRsHandler
1*50
TwentyRsHandler
FiveRsHandler
1*5


Chain of Responsibility Pattern
Class diagram
Sequence diagram
holds reference to
Handler
successor
Client
holds reference to
HandleRequest()
inherits
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()
: Client
firstInChain : Handler
secondInChain : Handler
1: handle(request)
1.1: handle(request)
1.2:
2:
If(can handle)
{
handle request
}
If(cannot handle or further processing is needed)
{
successor.handle(request)
}
Handler == Receiver


Mediator Pattern – Real Time Example
Ram
Raj
Dave
Anu
JAVA Group
facebook
join our group +
Raju
Steve
Rahul

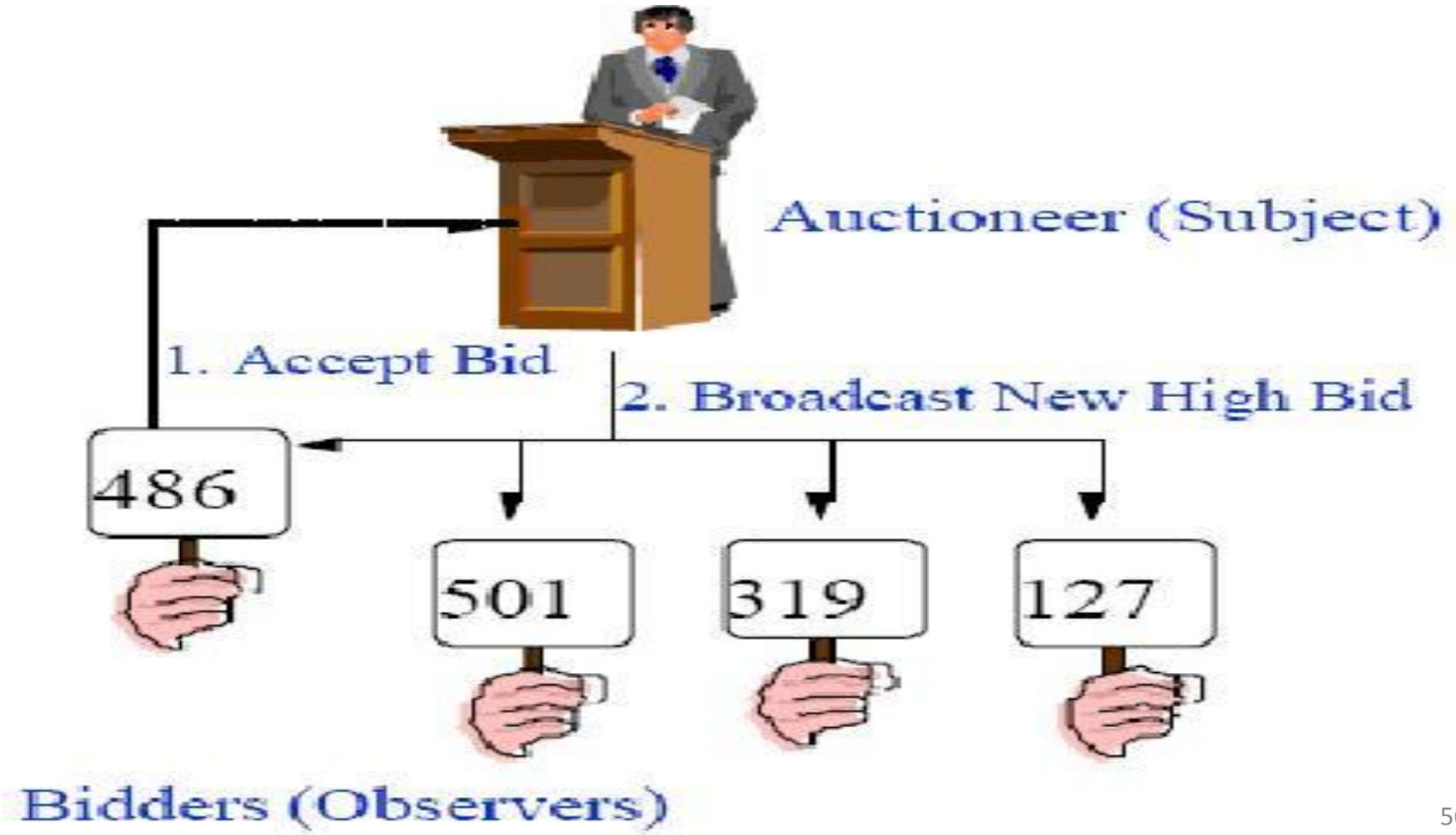

Mediator Pattern – Real Time Example
The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.
Air traffic control
ATC Mediator
Flight 7E7
Flight 747
Flight 1011
Flight 112

- Defines a one-to-many dependency between objects
- Main-idea :Object always require exact data of other object

- POLLING
- DELEGATION

Methods to notify:

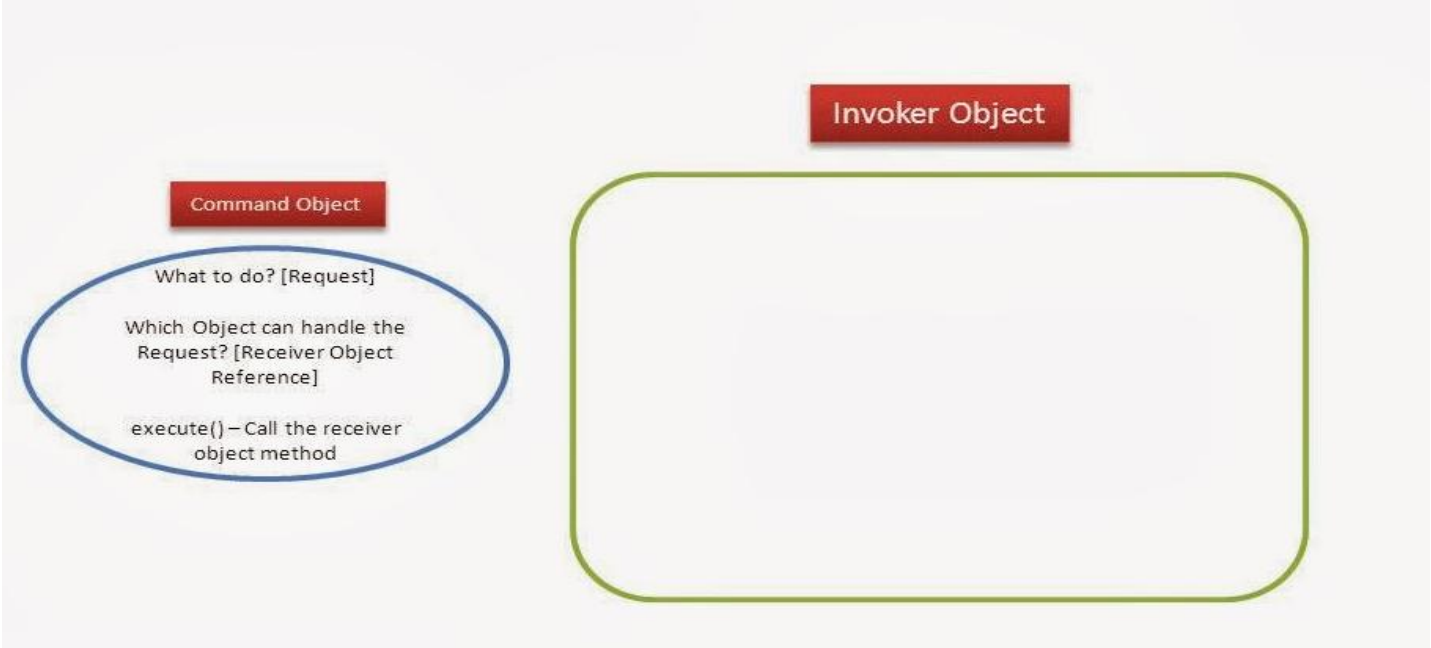
- Pull model
- Push model

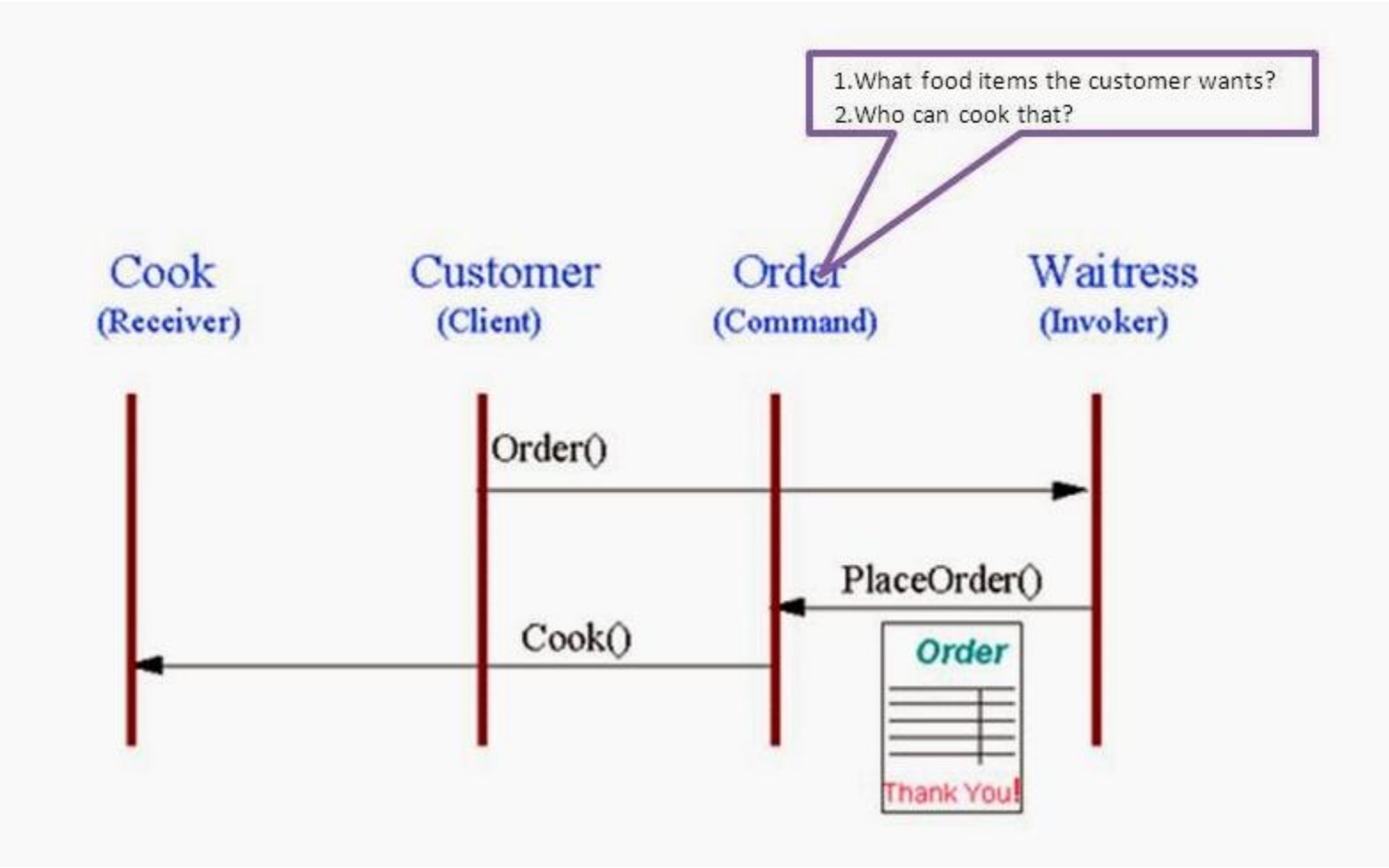

Auctioneer (Subject)
1. Accept Bid
2. Broadcast New High Bid
486
501
319
127
Bidders (Observers)

- When we don’t know how many objects need to change their state
- When an object is able to notify other objects without making assumptions about what those objects are

Encapsulate requests for service from an object inside other object(s) and manipulate requests.

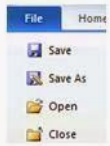
Command objects are mainly helpful in undo/redo operation where the previous state can be saved.




1.What food items the customer wants?
2.Who can cook that?
Cook
(Receiver)
Customer
(Client)
Order
(Command)
Waitress
(Invoker)
Order()
PlaceOrder()
Cook()
Order
Thank You!

Command Pattern – Real time Example

Menu Options [Invoker Object]



Open Command

Open the doc [Request]

Which Object can perform? [Document Object Reference]

execute()

Close Command

Close the doc [Request]

Which Object can perform? [Document Object Reference]

execute()

Save Command

Save the doc [Request]

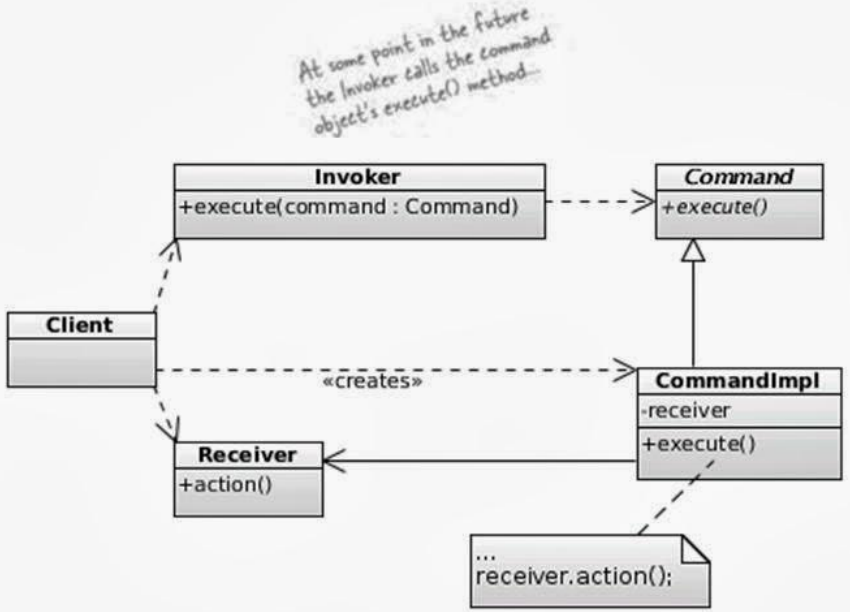
Which Object can perform? [Document Object Reference]

execute()

Document [Receiver Object]

1.Open the document
2.Close the Document
3.Save the Document

Command Pattern – Class Diagram



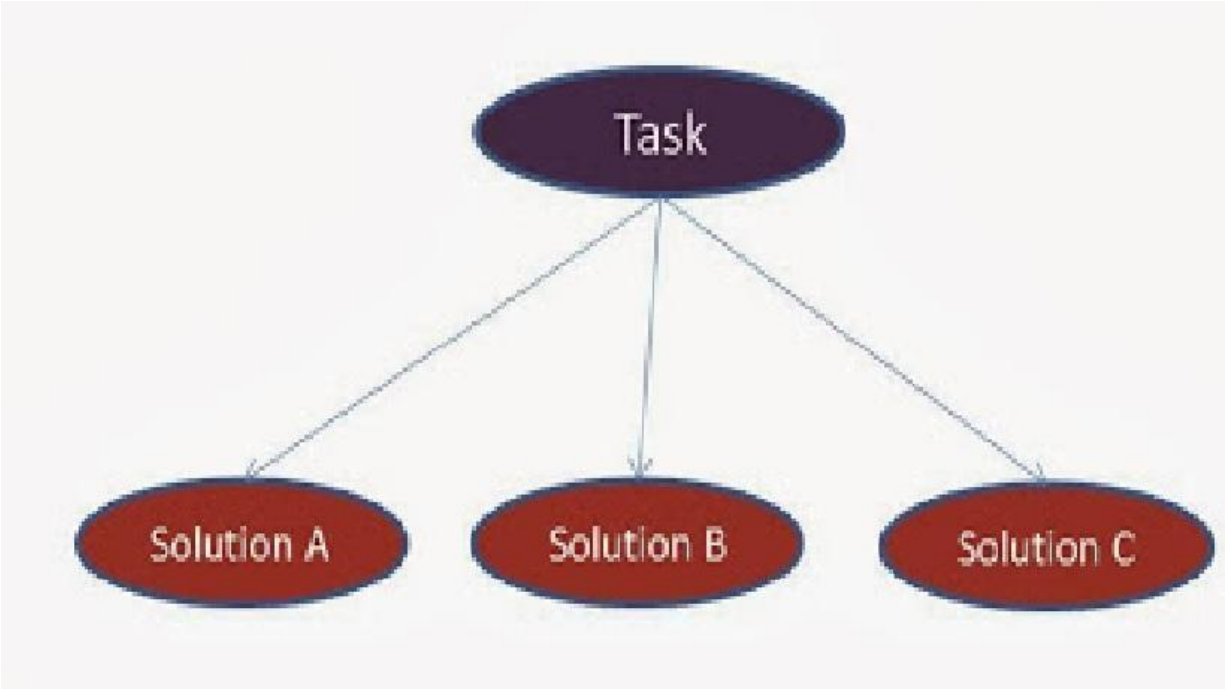
Client will create a Concrete Command and pass the reference of the Receiver via the Constructor.

The client is responsible for creating the command object. The command object consists of a set of actions on a receiver.

The actions and the Receiver are bound together in the command object.

The command object provides one method, execute(), that encapsulates the actions and can be called to invoke the actions on the Receiver.

●Used when there are multiple algorithms for the same task and it is to be chosen at runtime.



Strategy Pattern - Real Time Example

Steve

Cost : 90000 Rs

Shopping Mall

Strategy Pattern - Real Time Example

The traveler must chose the Strategy [options] based on tradeoffs between cost, convenience, and time

House

200 KM

Airport

Steve

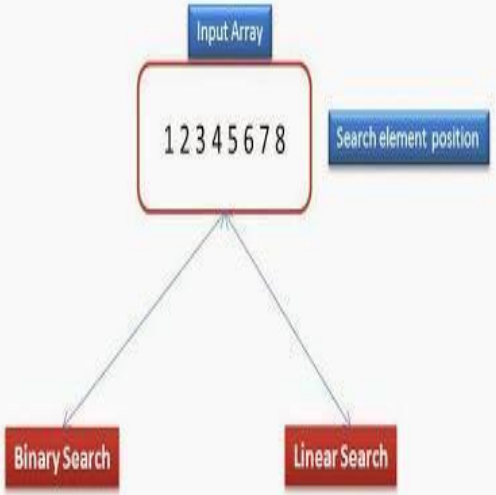
Strategy Pattern – Real Time Example

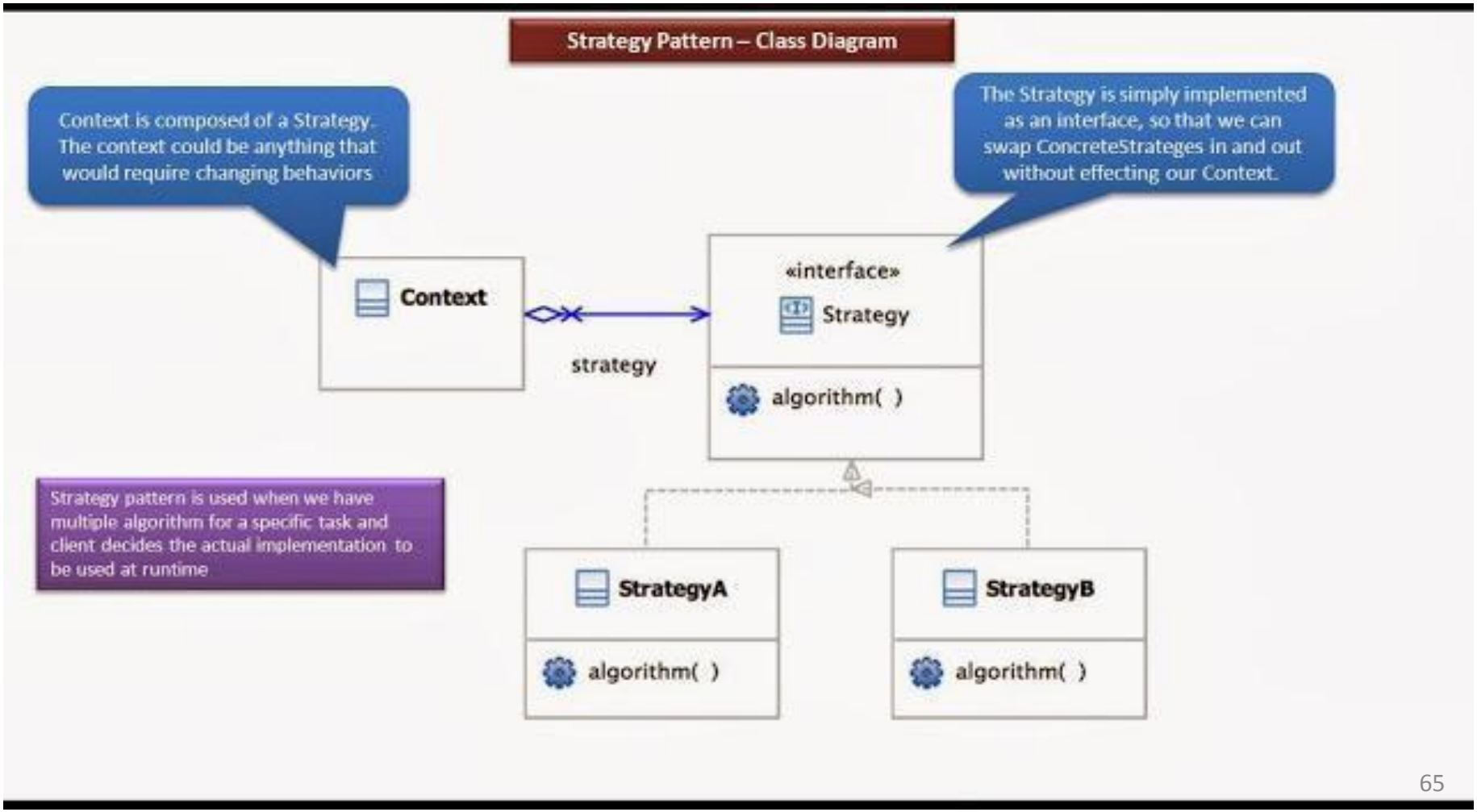
Selection sort: repeatedly pick the smallest element to append to the result.
Insertion sort: repeatedly add new element to the sorted result.
Bubble sort: repeatedly compare neighbor pairs and swap if necessary.

Strategy Pattern – Real Time Example

Linear search: It is a sequential search or it searches line by line. So it is slow and it takes too much time to find out the data.

Binary search: In Binary search it divide the table into two parts, a lower value part and a upper value part, after dividing it will check with the middle value if its lesser than the searched element than it goes to right half else it goes to left half. Therefore it is necessary that the search filed of the table should be in Ascending order. If you sort the list in descending order then binary search fails.

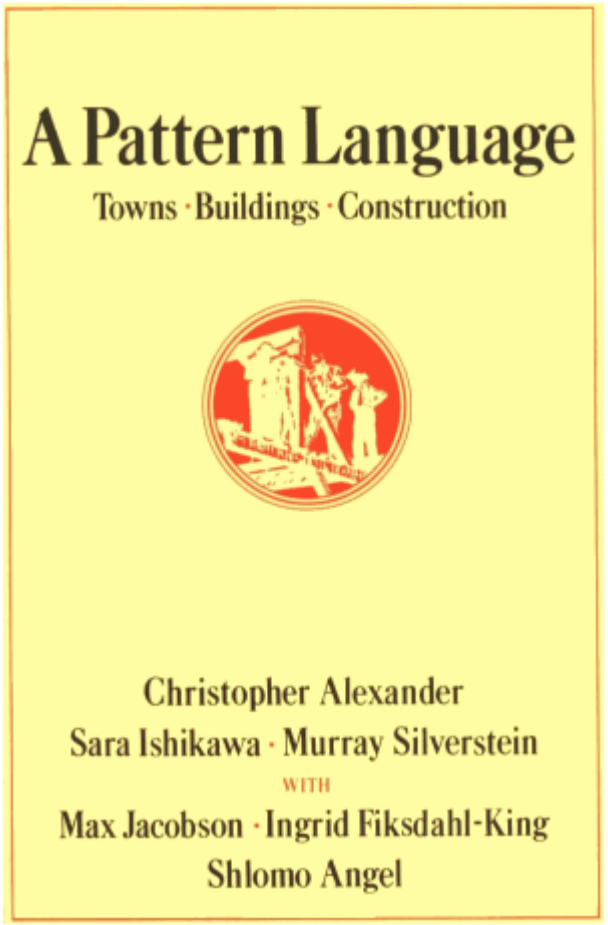



Strategy Pattern – Class Diagram
Context is composed of a Strategy. The context could be anything that would require changing behaviors
The Strategy is simply implemented as an interface, so that we can swap ConcreteStrategies in and out without effecting our Context.
Context
strategy
«interface»
Strategy
algorithm( )
Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime
StrategyA
algorithm( )
StrategyB
algorithm( )

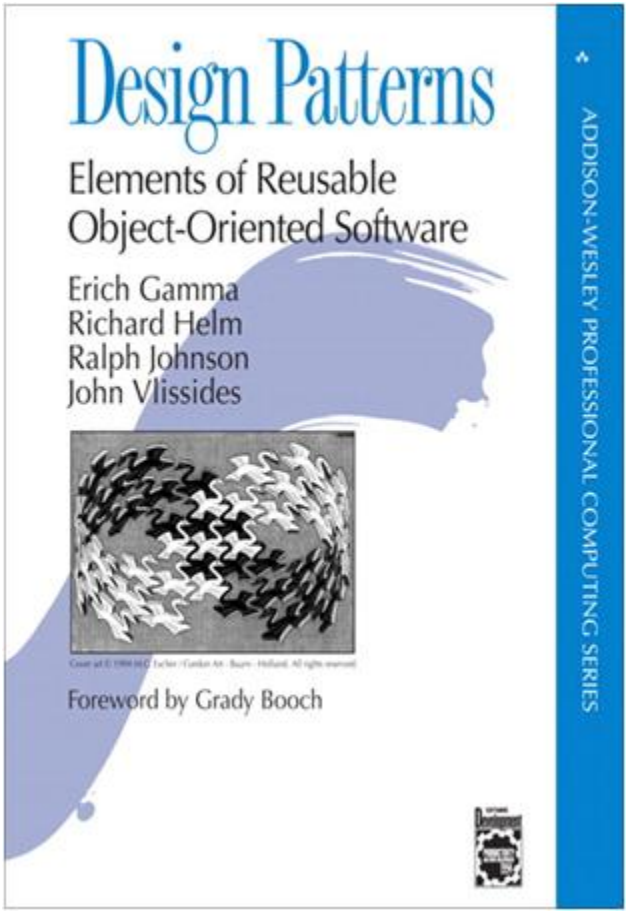
1. Speed
2. Reuse.
3. Documented solutions.
4. Communication Standards
5. Always evolving.

“Design patterns are a form of complexity. As with all complexity, I'd rather see developers focus on simpler solutions *before* going straight to a complex recipe of design patterns.”

"Design Patterns" solution is to turn the programmer into a fancy macro processor.


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

≠


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

You could read Design Patterns like any number of other software developers before you. But we humbly suggest that you should go deeper and read A Pattern Language, too, because ideas are more important than code.

Name:
Problem: Should describe when to apply the pattern
Solution: Should describe the elements that make up the design, relationships, responsibilities and collaborations
Consequences: Should describe the results and trade-offs of applying the pattern

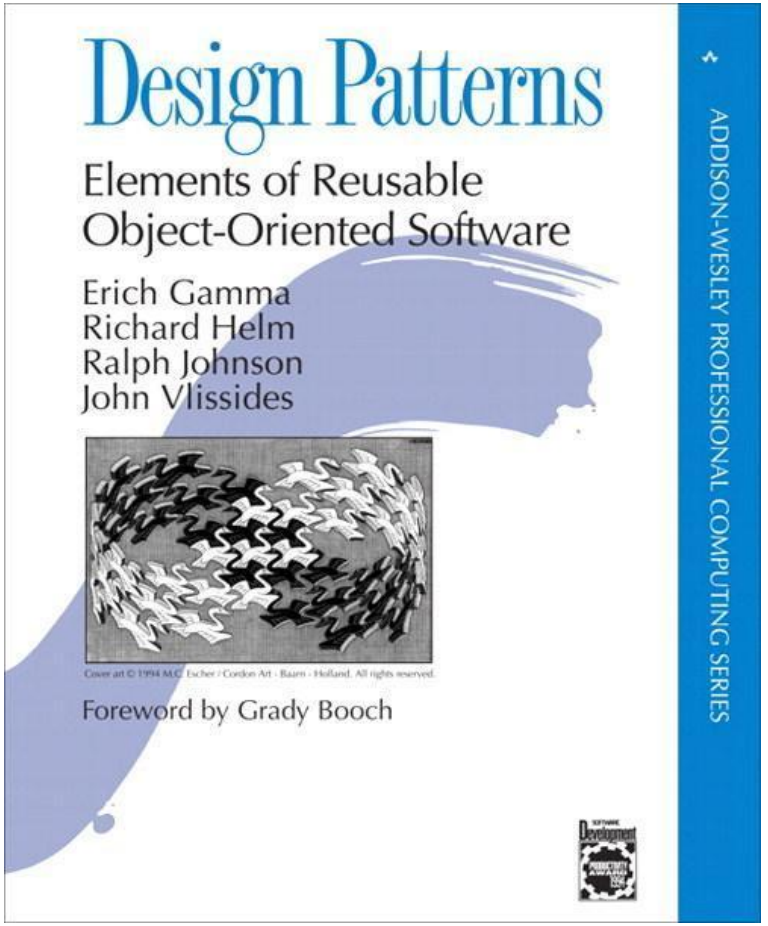
Real-time example:

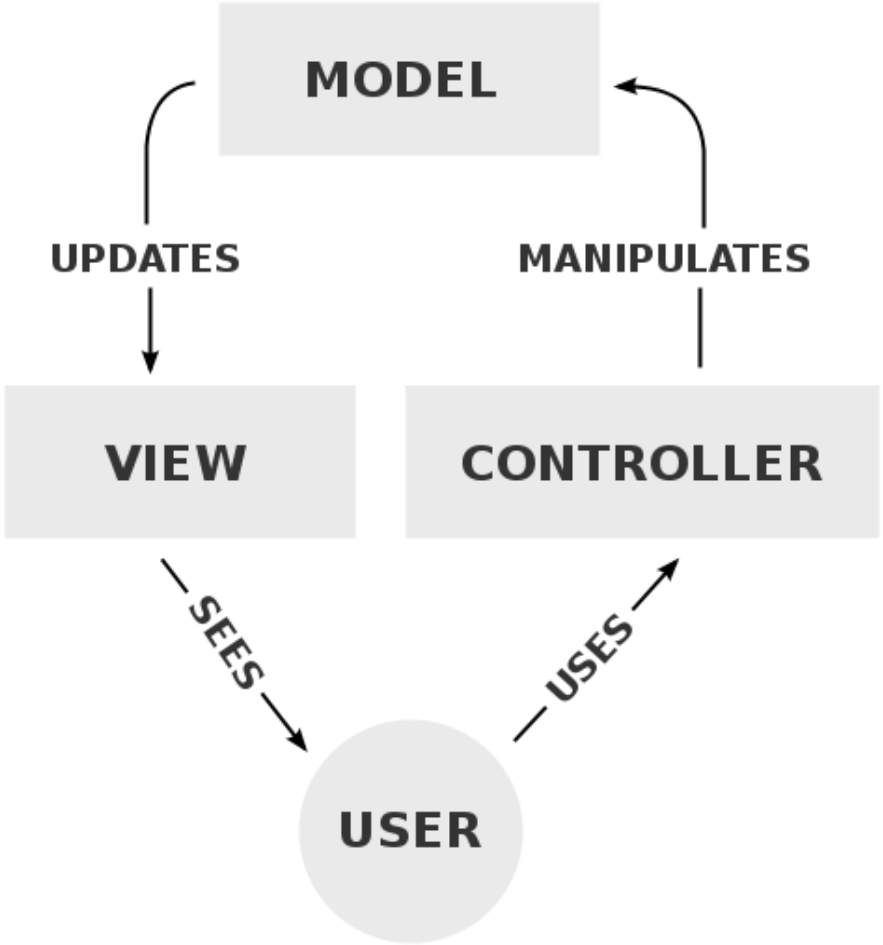
Nicknamed the

BIBLE

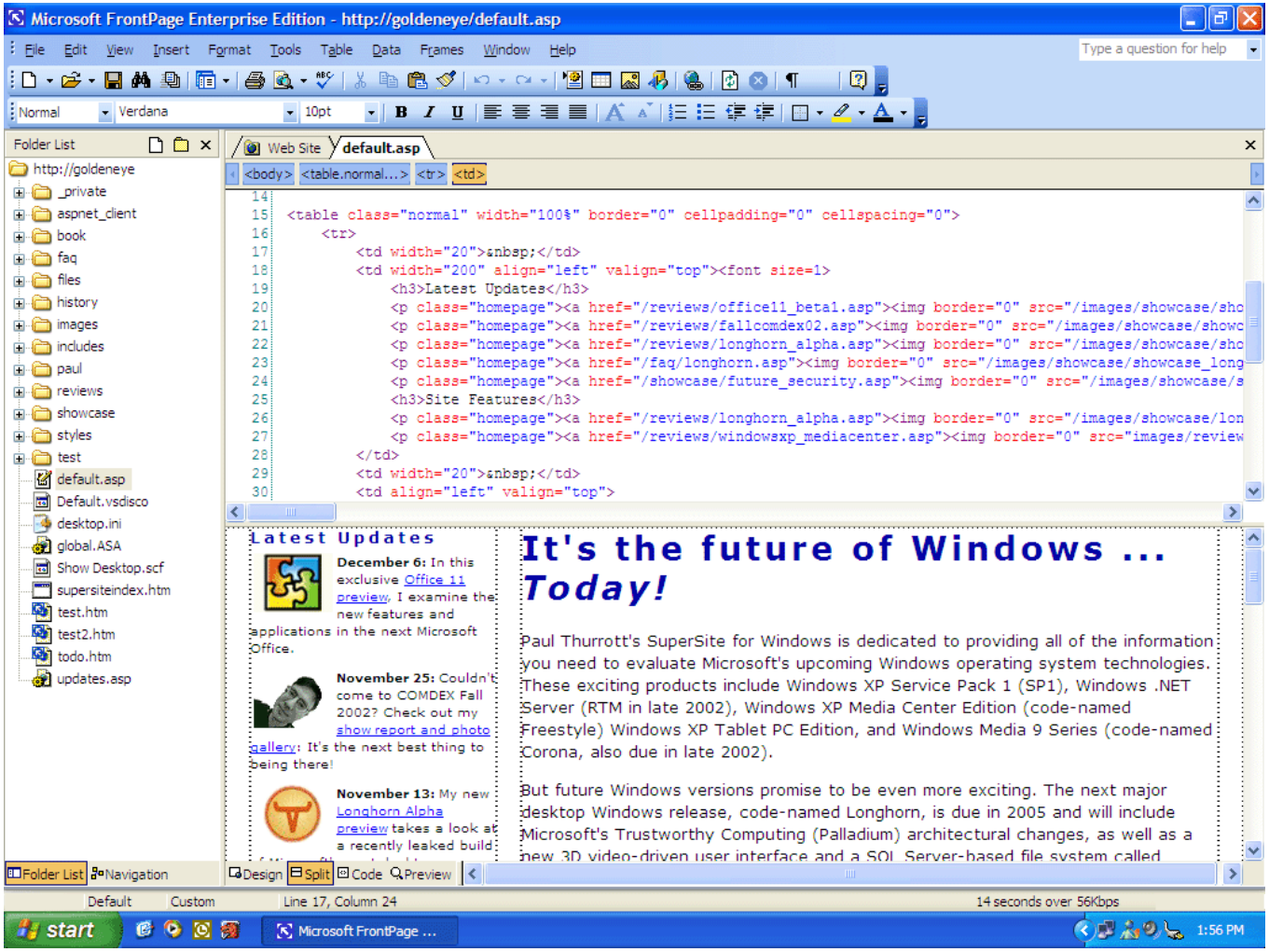
Of

Software Design Patterns




MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER

- MVC used in app **frameworks**, interactive systems
- In MVC, computational and representational aspects strictly **separated**
- To be **maintained throughout system's evolution**
- Offers **skeleton** for interactive systems
- **Addresses Non-functional rqmts**: flexibility, changeability of UI


Microsoft FrontPage Enterprise Edition - http://goldeneye/default.asp
File Edit View Insert Format Tools Table Data Frames Window Help
Type a question for help
Normal Verdana 10pt
Folder List
http://goldeneye
_private
aspnet_client
book
faq
files
history
images
includes
paul
reviews
showcase
styles
test
default.asp
Default.vsdisco
desktop.ini
global.ASA
Show Desktop.scf
supersiteindex.htm
test.htm
test2.htm
todo.htm
updates.asp
Web Site
default.asp
<body> <table.normal...> <tr> <td>
<table class="normal" width="100%" border="0" cellpadding="0" cellspacing="0">
<tr>
<td width="20"> </td>
<td width="200" align="left" valign="top"><font size=1>
<h3>Latest Updates</h3>
<h3>Site Features</h3>
<td width="20"> </td>
<td align="left" valign="top">
Latest Updates
December 6: In this exclusive Office 11 preview, I examine the new features and applications in the next Microsoft Office.
November 25: Couldn't come to COMDEX Fall 2002? Check out my show report and photo gallery: It's the next best thing to being there!
November 13: My new Longhorn Alpha preview takes a look at a recently leaked build
It's the future of Windows ... Today!
Paul Thurrott's SuperSite for Windows is dedicated to providing all of the information you need to evaluate Microsoft's upcoming Windows operating system technologies. These exciting products include Windows XP Service Pack 1 (SP1), Windows .NET Server (RTM in late 2002), Windows XP Media Center Edition (code-named Freestyle) Windows XP Tablet PC Edition, and Windows Media 9 Series (code-named Corona, also due in late 2002).
But future Windows versions promise to be even more exciting. The next major desktop Windows release, code-named Longhorn, is due in 2005 and will include Microsoft's Trustworthy Computing (Palladium) architectural changes, as well as a
Folder List Navigation
Design Split Code Preview
Default Custom Line 17, Column 24
14 seconds over 56Kbps
start
Microsoft FrontPage ...
1:56 PM

AGENDA COVERED

- Patterns and Design Patterns
- Need of usage
- Reason behind their division
- Basic confusion
- Types of design patterns
- Design pattern elements

Patterns

- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
- Especially good for describing how and why to resolve nonfunctional issues
- Patterns facilitate reuse of successful software architectures and designs

Design pattern

- A general reusable solution to a commonly occurring problem within a given context in software design
- Describes recurring design structures
- Describes the context of usage

Need of usage

- Speed up the development process by providing tested, proven development paradigms

- Reusing design patterns helps to prevent issues that can cause major problems

- Improves code readability for coders and architects who are familiar with the patterns

Reason behind their division

- The problems are different
- The contexts are different
- The designs we choose are different
- The OOPs concepts used to solve the problems are different

Basic confusion

- Not a finished design that can be directly transformed to code
- Just a template which shows how to solve the problem in different situations

Design patterns existing…

There are many types of design patterns, like

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to ***exploit application characteristic*** on a computing platform.
- **Computational design patterns** addressing concerns related to ***key computation identification***.
- **Execution patterns** that address concerns related to ***supporting application execution***, including strategies in executing streams of tasks and building blocks to support task synchronization.

Design patterns we existing…

- **Implementation strategy patterns** addressing concerns related to implementing source code to ***support program organization***, and the common ***data structures specific to parallel programming.***
- **Structural design patterns** addressing concerns related to ***high-level structures of applications being developed.***

Design patterns types

Creational pattern:

- Deal with object creation mechanisms
- Reduces the complexity of design by controlling the object creation
- Further divided into two categories
  - Object creational pattern
  - Class creational pattern

Design Patterns types(Cond)…

Structural pattern:

- Ease the design by identifying a simple way to realize the relationship between entities

Behavioral pattern:

- Identify common communication between objects
- Flexibility in carrying the communication between the objects increases

Elements of design patterns

There are 4 elements for a design pattern. They are

- **Name**: Describes the name of the design pattern being used in that context
- **Problem**: Describes when to apply the pattern
- **Solution**: Describes the elements that make up the design, relationships, responsibilities and collaborations
- **Consequences**: Describes the results and trade-offs of applying the pattern

---

CREATIONAL PATTERNS:

Introduction to patterns
Factory Method

Abstract Factory
Builder
Prototype
Singleton
Object Pool
Lazy Initialization

STRUCTURAL PATTERNS:

Decorator
Composite
Proxy
FlyWeight
Facade
Bridge
Adapter

- ---
- **BEHAVIORAL PATTERNS:**
- ---
- Strategy
- Iterator
- Template Method
- Mediator
- Observer
- Chain of Responsibility
- Memento
- Command
- State
- Visitor
- Interpreter

DESIGN PATTERNS

Factory Method

&

Abstract Factory

AGENDA

- Factory Design Pattern
- AbstractFactory Design Pattern

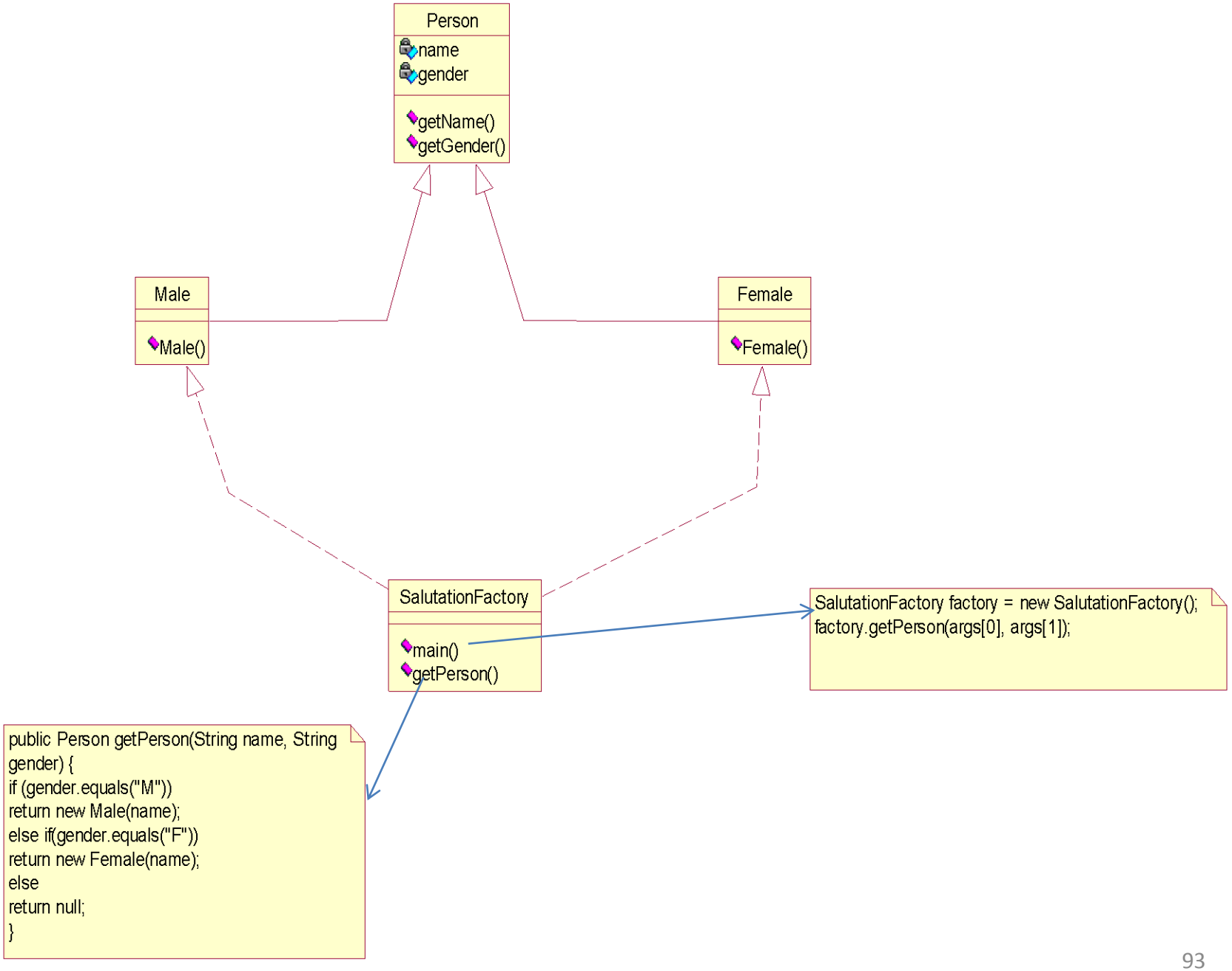
PATTERN STANDARD FORMAT

- Name
- Problem
- Solution
- Consequences
- Real-time example

FACTORY METHOD

INTENT

Defines an interface for creating objects, but let subclasses to decide which class to instantiate


Person
name
gender
getName()
getGender()
Male
Male()
Female
Female()
SalutationFactory
main()
getPerson()
SalutationFactory factory = new SalutationFactory();
factory.getPerson(args[0], args[1]);
public Person getPerson(String name, String
gender) {
if (gender.equals("M"))
return new Male(name);
else if(gender.equals("F"))
return new Female(name);
else
return null;
}

PARTICIPANTS

The classes that participate to the Factory pattern are:

AbstractProduct	Declares a interface for operations that create abstract products
ConcreteCreator	Implements operations to create concrete products.
ConcreteProduct	Defines a product to be created by the corresponding ConcreteFactory

APPLICABILITY (When to use?)

The Factory patterns can be used in following cases:

- When a class does not know which class of objects it must create.
- A class specifies its sub-classes to specify which objects to create.
- In programmer's language, you can use factory pattern where you have to create an object of any one of sub-classes depending on the data provided.

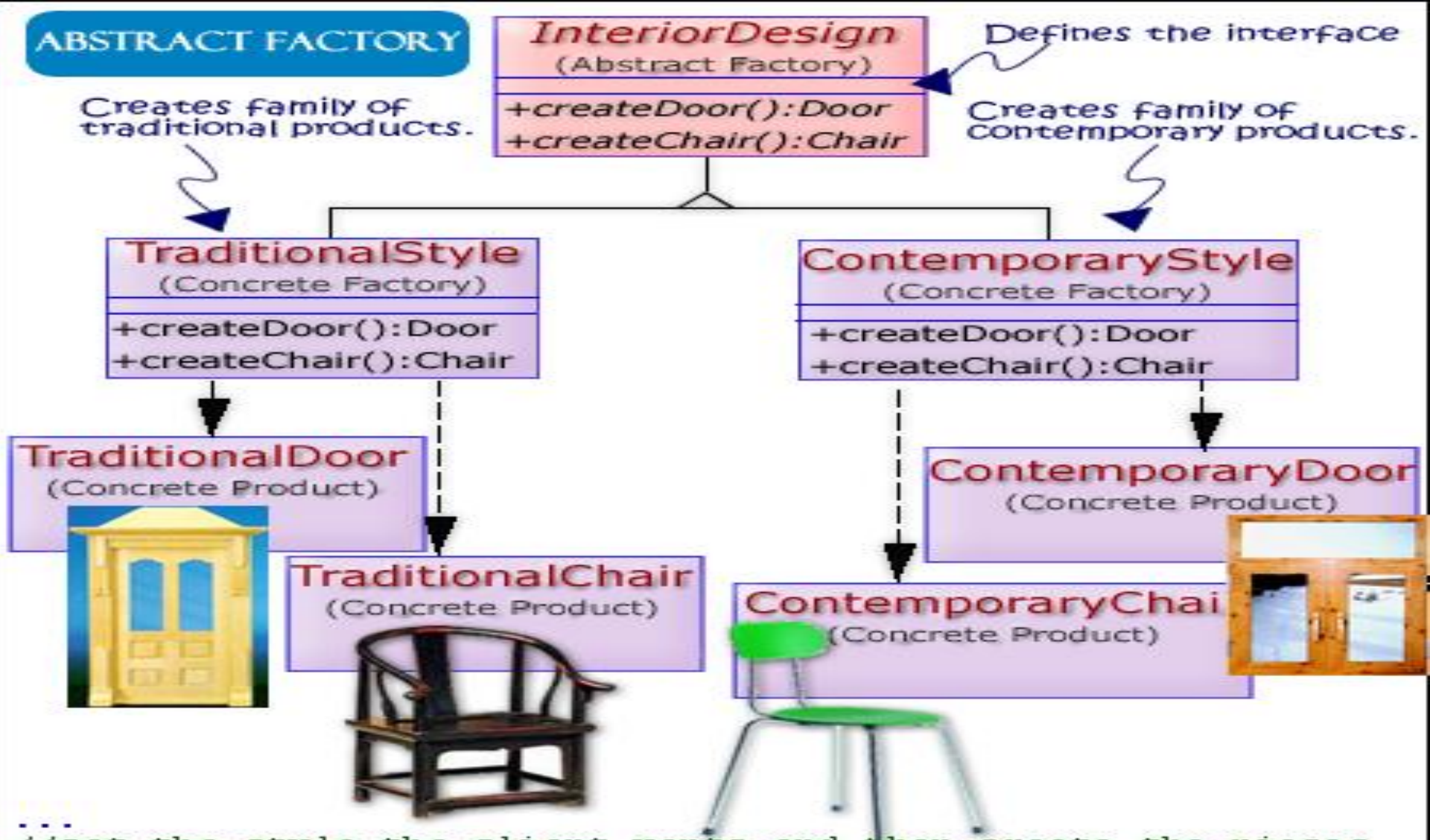
CONSEQUENCES

- The client code deals only with the product interface, therefore it can work with any user defined Concrete Product classes
- New concrete classes can be added without recompiling the existing client code
- It may lead to many subclasses if the product objects requires one or more additional objects (Parallel class hierarchy)

ABSTRACT FACTORY

INTENT

Provide an interface for creating families of related or dependent objects without specifying their concrete classes




```
...
//set the style the client wants and then create the pieces
InteriorDesign design = new TraditionalStyle();
/*create pieces for a design: door, chair.
When do that, the client ask for a door and receives a
traditional one*/
//create a TraditionalDoor
Door door = design.createDoor();
//create a TraditionalChair
Chair chair = design.createChair();
...
```

PARTICIPANTS

The classes that participate to the Abstract Factory pattern are:

AbstractFactory	Declares a interface for operations that create abstract products
ConcreteFactory	Implements operations to create concrete products.
ConcreteProduct	Defines a product to be created by the corresponding ConcreteFactory
Client	Uses the interfaces declared by the AbstractFactory class.

APPLICABILITY (When to use?)

- A system should be independent of how its products are created, composed, or represented
- A family of related product objects is designed to be used together
- You want to provide a class library of objects, but reveal only their interfaces

CONSEQUENCES

1. Concrete class isolation (Good)
   - Client does not interact with the implementation classes
   - Client only manipulates instances through the abstract interfaces

2. Product families easily exchanged (Good)
   - Only have to change the concrete factory
   - Can be done at run time

CONSEQUENCES

3. Products are more consistent (Good)
   - Helps the products in each product family consistently be applied together (assuming they work well together)
   - Only one family at a time

4. Difficult to support new kinds of products (Bad)
   - Extending existing abstract factories to make new products is difficult and time consuming
   - The family of products available is fixed by Abstract Factory interface

Agenda

DESIGN PATTERNS

- Singleton Design Pattern
- Object Pool Design Pattern

Singleton Design Pattern

Intent

To ensure that a class has only one instance and provides a global access point

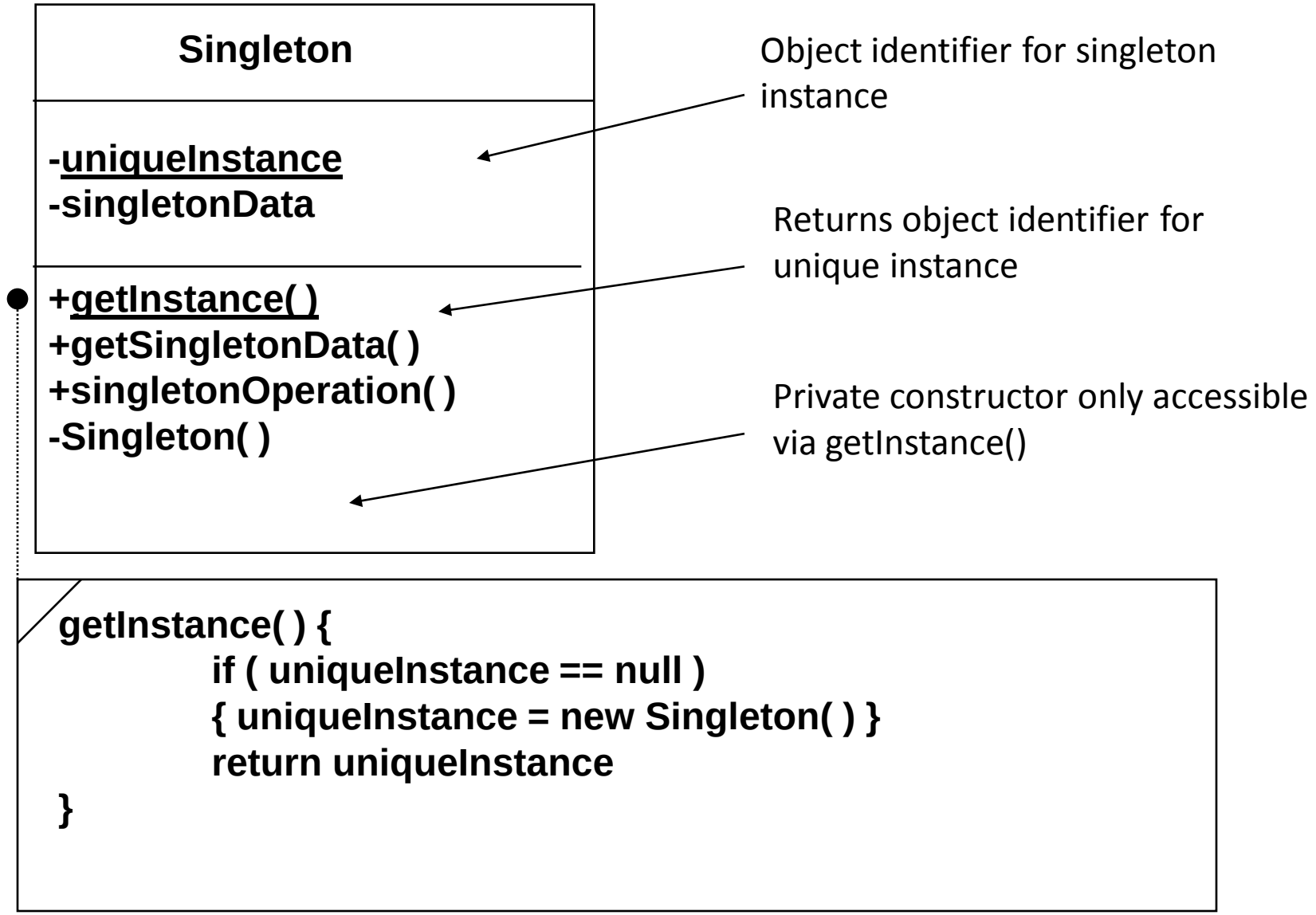
Problem

How can we guarantee that one and only one instance of a class can be created?

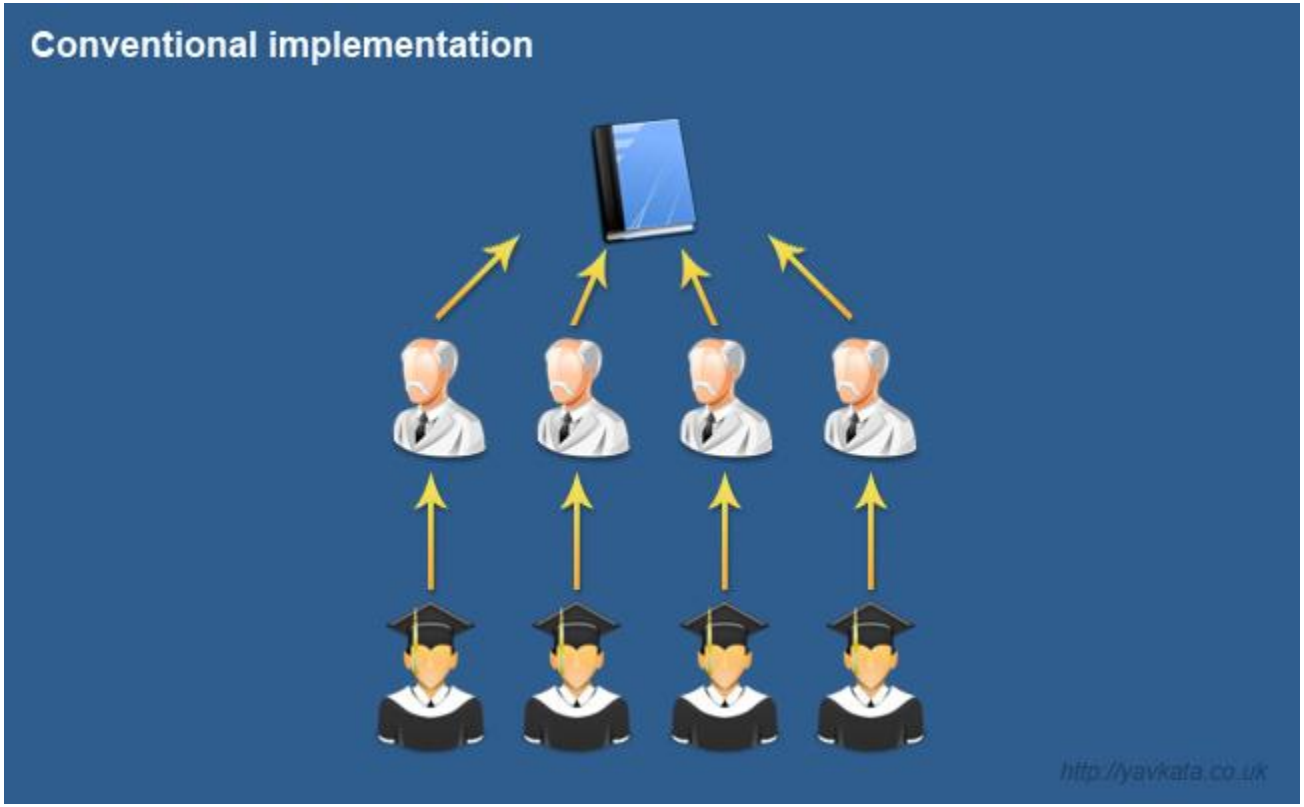
Solution

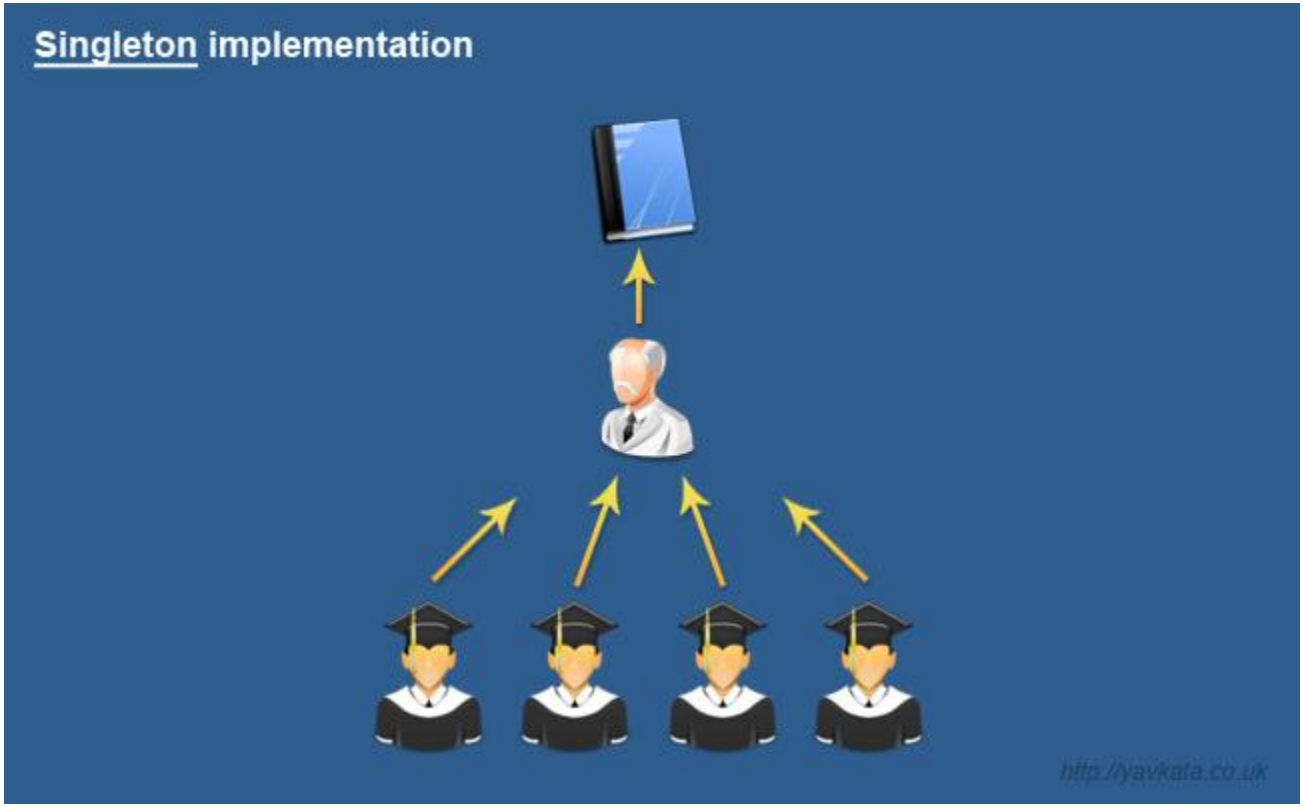
- Create a class with a class operation getInstance().
- When class is first accessed, this creates relevant object instance and returns object identity to client.
- On subsequent calls of getInstance(), no new instance is created, but identity of existing object is returned.

Singleton Structure



Example




Singleton implementation
http://yavkata.co.uk

Benefits

- Controlled access to the sole instance
- Permits a variable number of instances

Applicability

Singleton pattern can be applied when there must be exactly one instance of a class and it must be accessible to clients

Consequences

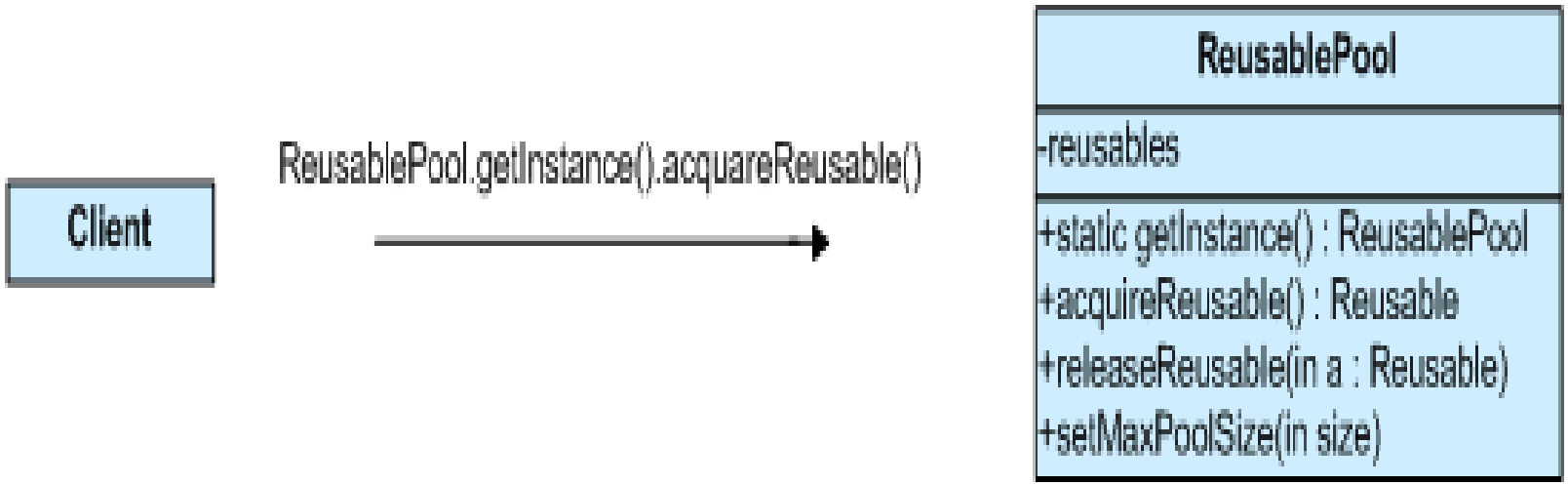
- Reduced name space
- Permits refinement of operations and representations
- More flexible than class operations

Object Pool

Intent

Reuse and share the objects that are expensive to create.

Structure


Client
ReusablePool.getInstance().acquareReusable()
ReusablePool
-reusables
+static getInstance() : ReusablePool
+acquireReusable() : Reusable
+releaseReusable(in a : Reusable)
+setMaxPoolSize(in size)

Applicability

- Your application requires objects which are "expensive" to create.

- Several parts of your application require the same objects at different times.

Example

1. `myShoes = shelf.acquireShoes();`

2. `client.wear(myShoes);`

SHELF (OBJECT POOL)

4. `shelf.releaseShoes(myShoes);`

3. `client.play();`

Known Uses

Instantiation of objects that represent:

- database connections
- socket connections
- threads

Specific problems

- Limited number of resources in the pool
- Handling situations when creating a new resource fails
- Synchronization
- Expired resources(unused but still reserved)

Benefits

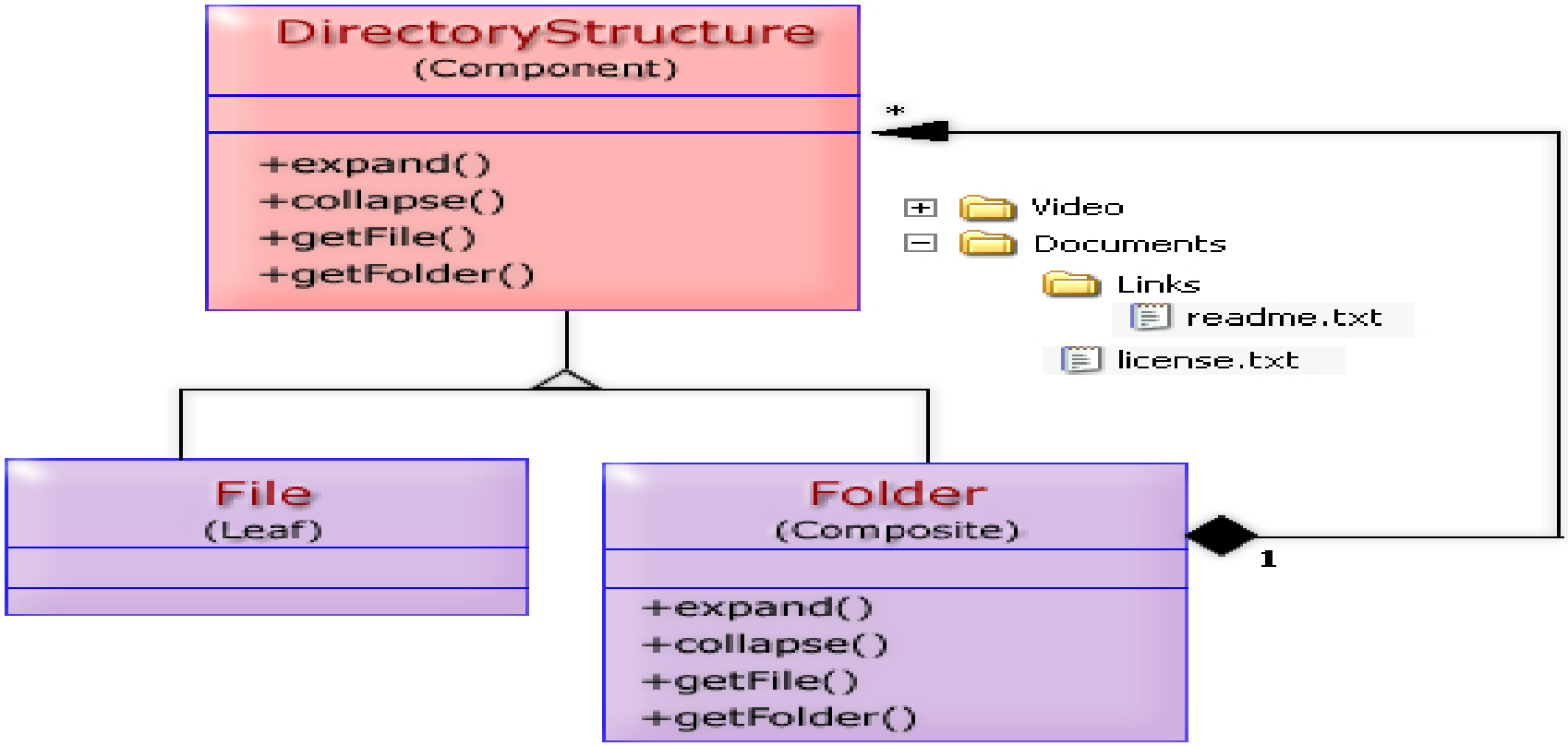
Can offer a performance boost where:

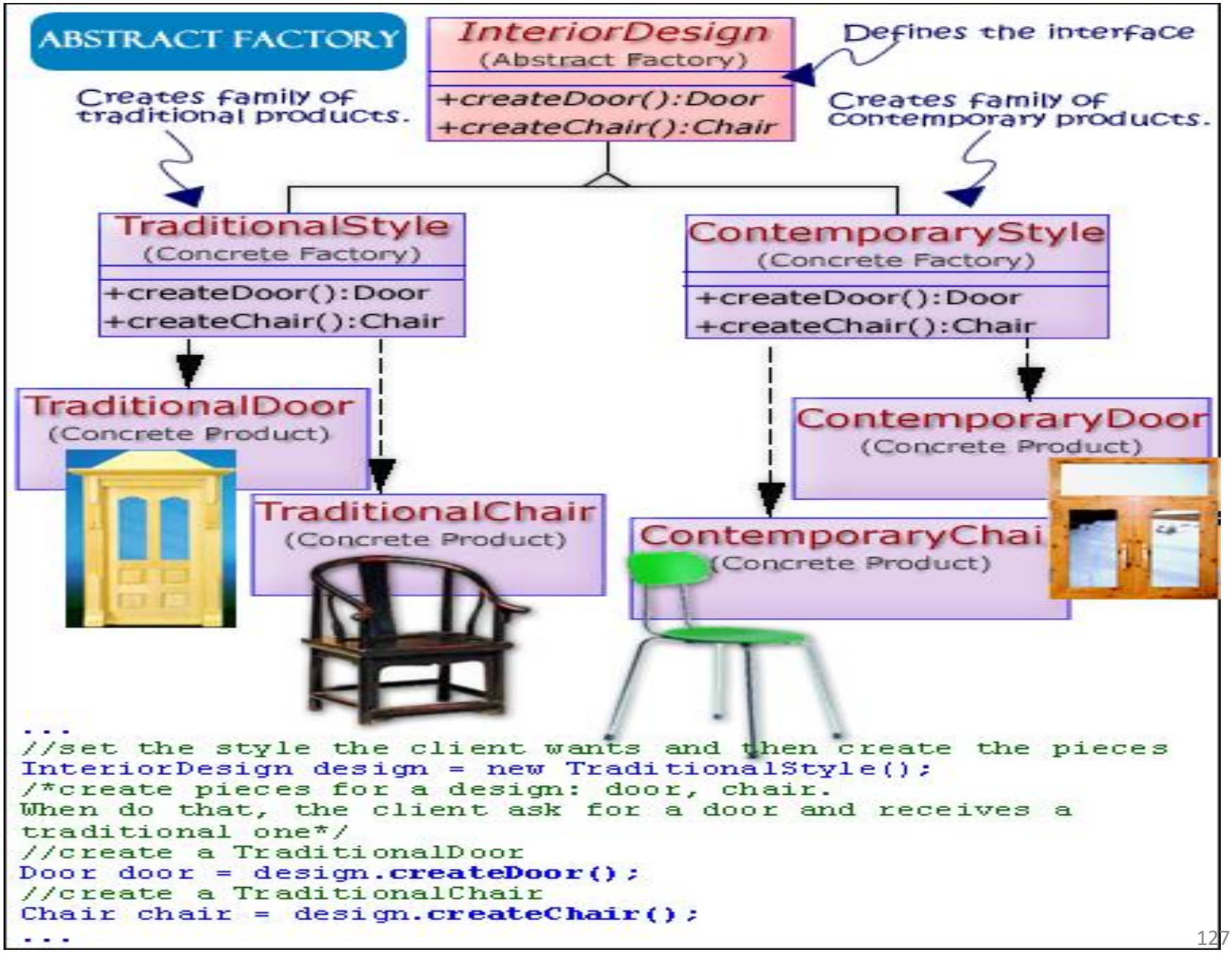
- object instantiation is cheaper
- number of instances at any one time is small
- Can make initialization time predictable where it would otherwise be unpredictable (e.g. when squiring resources over a network)

Real-World Illustrations

- Shoe shelf at a bowling club
- Library

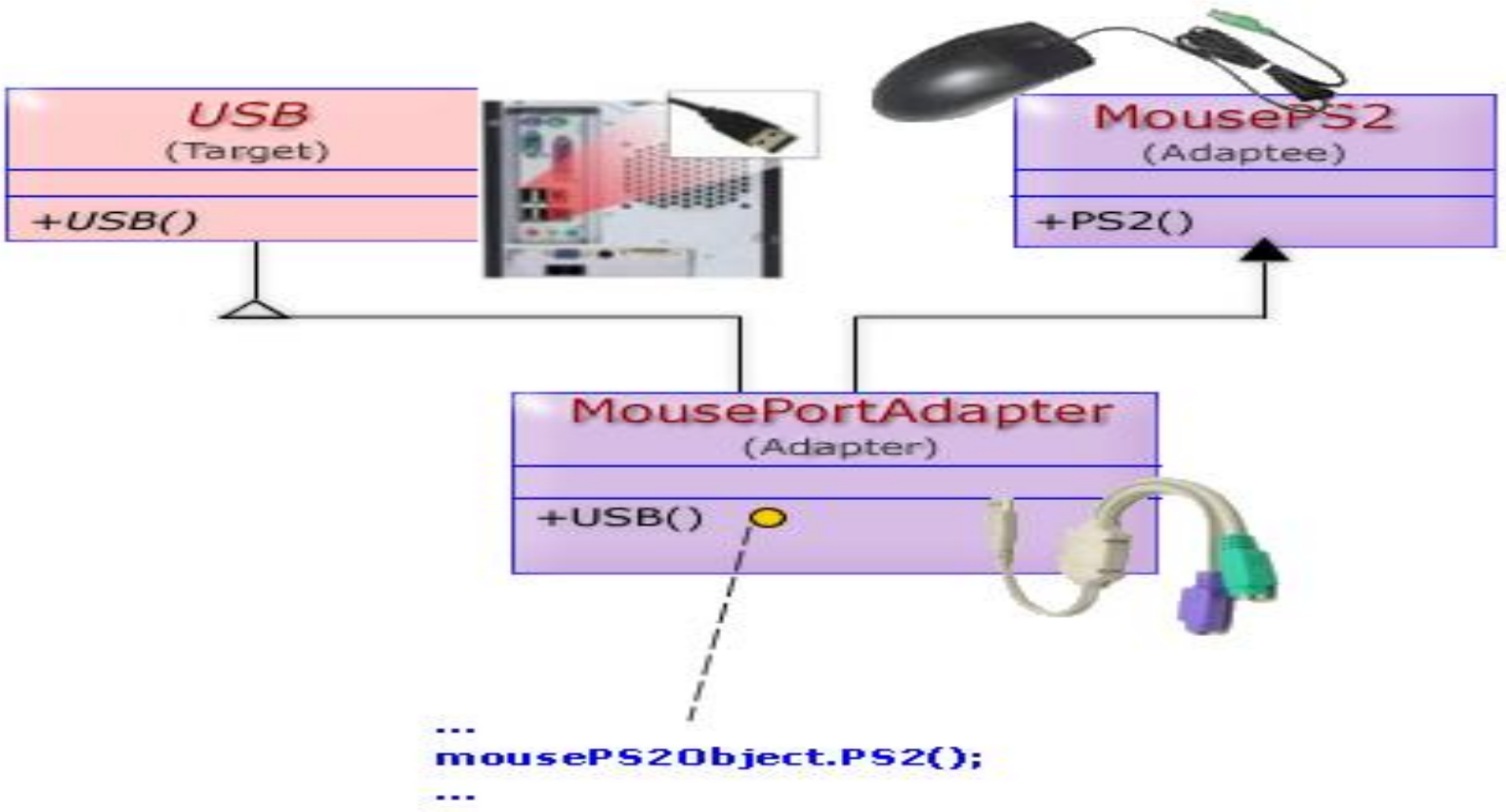
COMPOSITE


DirectoryStructure
(Component)
+expand()
+collapse()
+getFile()
+getFolder()
*
Video
Documents
Links
readme.txt
license.txt
File
(Leaf)
Folder
(Composite)
+expand()
+collapse()
+getFile()
+getFolder()
1

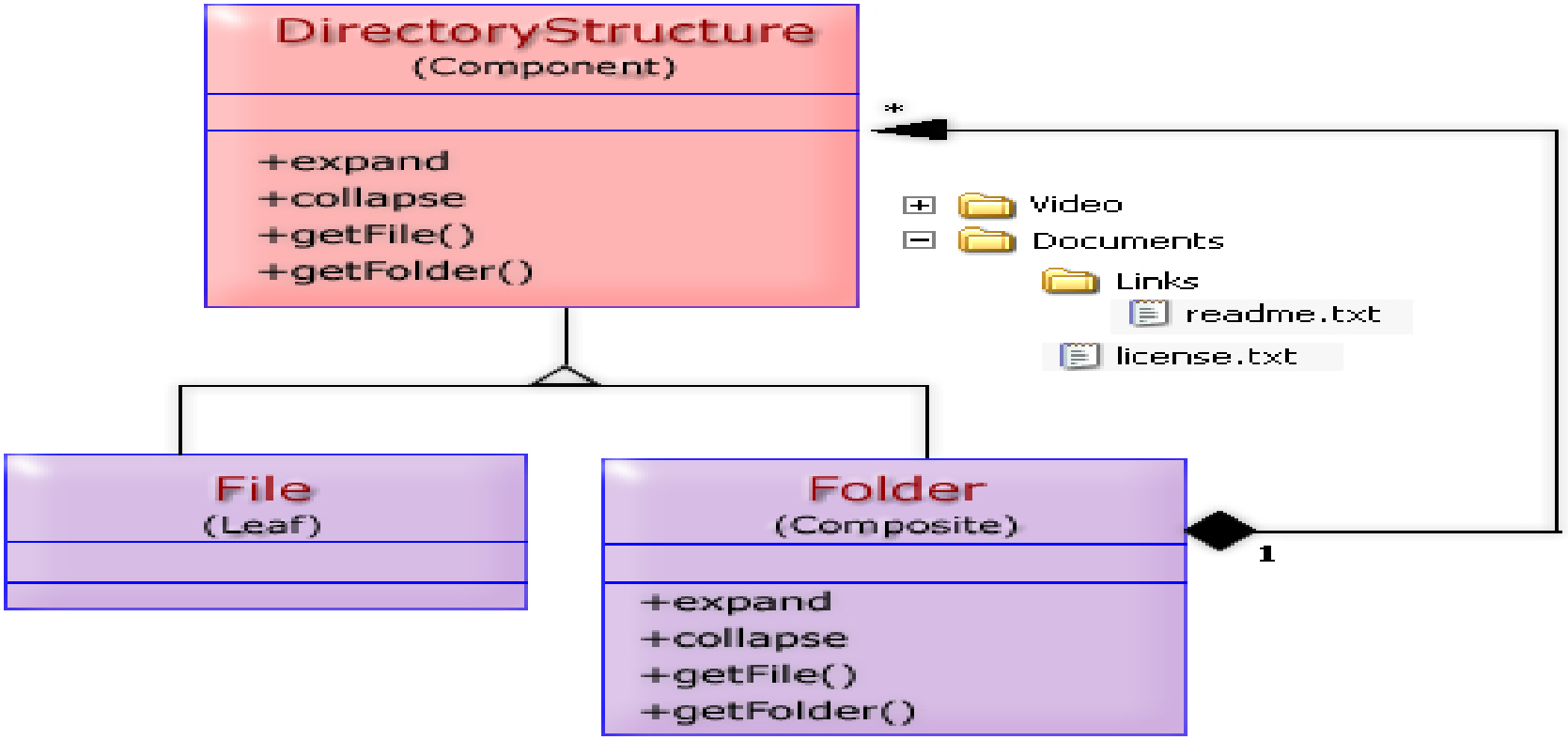


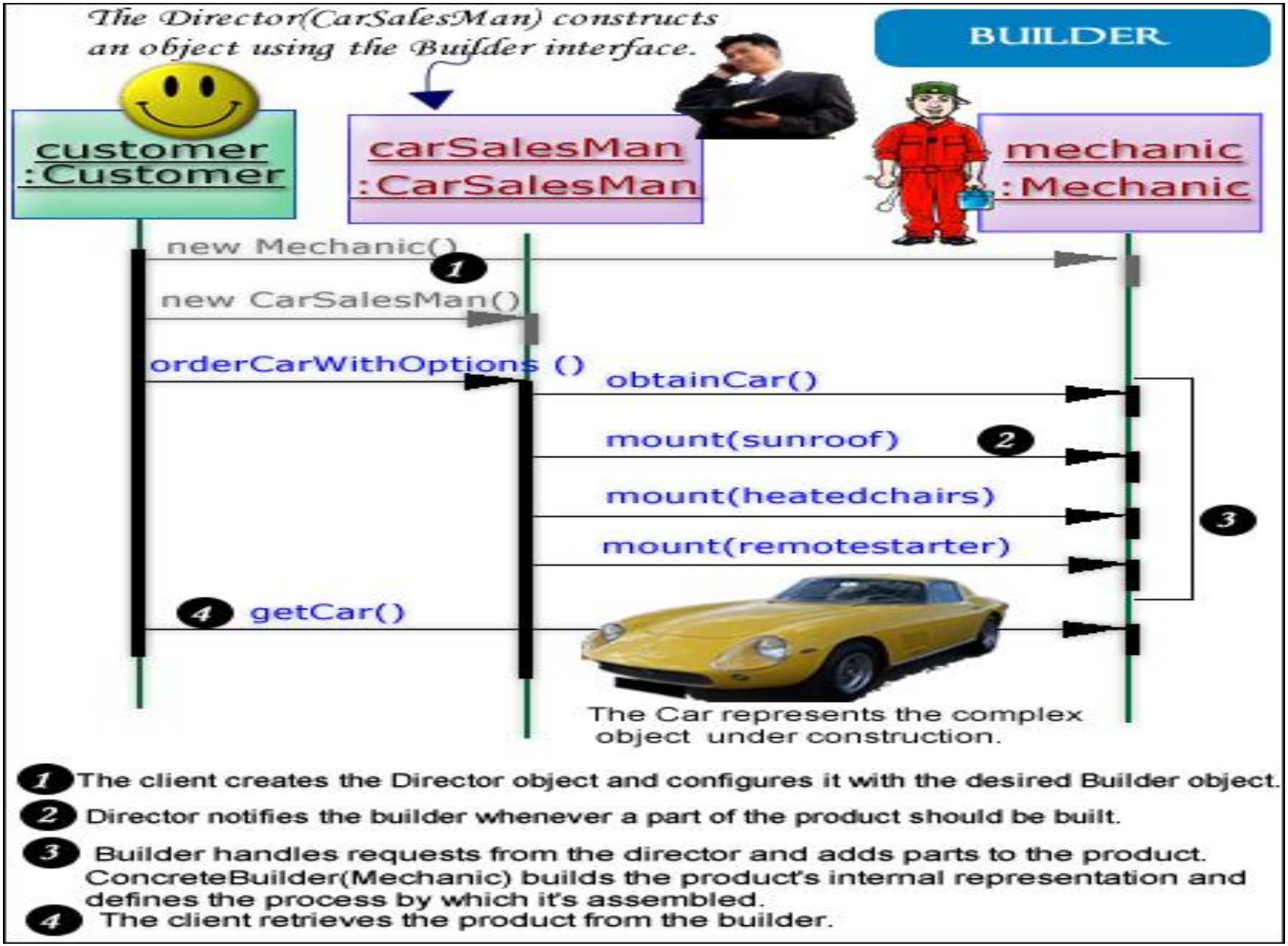

```
...
//set the style the client wants and then create the pieces
InteriorDesign design = new TraditionalStyle();
/*create pieces for a design: door, chair.
When do that, the client ask for a door and receives a
traditional one*/
//create a TraditionalDoor
Door door = design.createDoor();
//create a TraditionalChair
Chair chair = design.createChair();
...
```

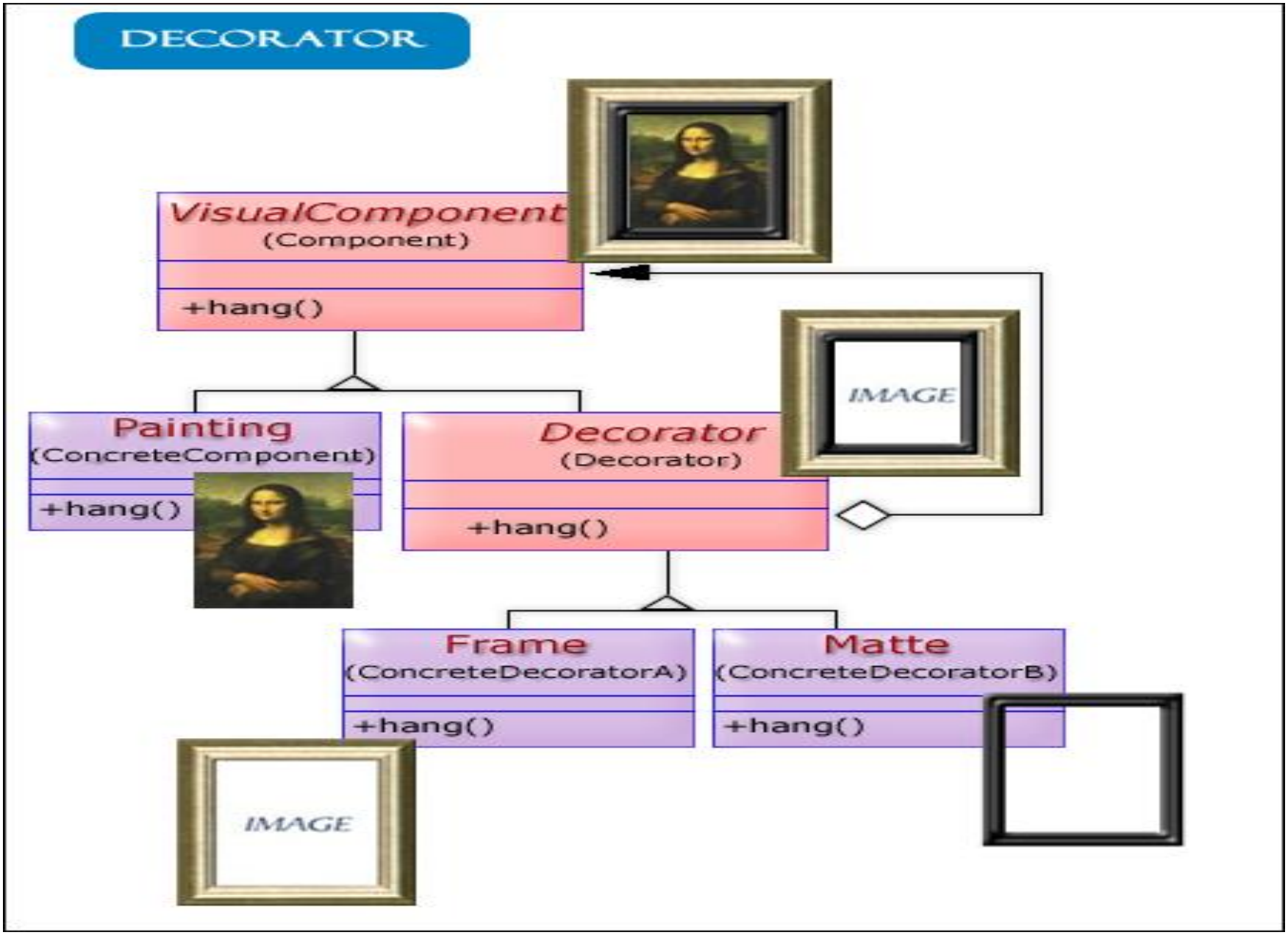
ADAPTER


USB
(Target)
+USB()
MousePS2
(Adaptee)
+PS2()
MousePortAdapter
(Adapter)
+USB()
...
mousePS2Object.PS2();
...

BRIDGE


DirectoryStructure
(Component)
+expand
+collapse
+getFile()
+getFolder()
*
Video
Documents
Links
readme.txt
license.txt
File
(Leaf)
Folder
(Composite)
+expand
+collapse
+getFile()
+getFolder()
1


The Director(CarSalesMan) constructs an object using the Builder interface.
BUILDER
customer :Customer
carSalesMan :CarSalesMan
mechanic :Mechanic
new Mechanic()
1
new CarSalesMan()
orderCarWithOptions ()
obtainCar()
mount(sunroof)
2
mount(heatedchairs)
mount(remotestarter)
3
4 getCar()
The Car represents the complex object under construction.
1 The client creates the Director object and configures it with the desired Builder object.
2 Director notifies the builder whenever a part of the product should be built.
3 Builder handles requests from the director and adds parts to the product. ConcreteBuilder(Mechanic) builds the product's internal representation and defines the process by which it's assembled.
4 The client retrieves the product from the builder.


DECORATOR
VisualComponent
(Component)
+hang()
Painting
(ConcreteComponent)
+hang()
Decorator
(Decorator)
+hang()
IMAGE
Frame
(ConcreteDecoratorA)
+hang()
Matte
(ConcreteDecoratorB)
+hang()
IMAGE

FAÇADE

Client

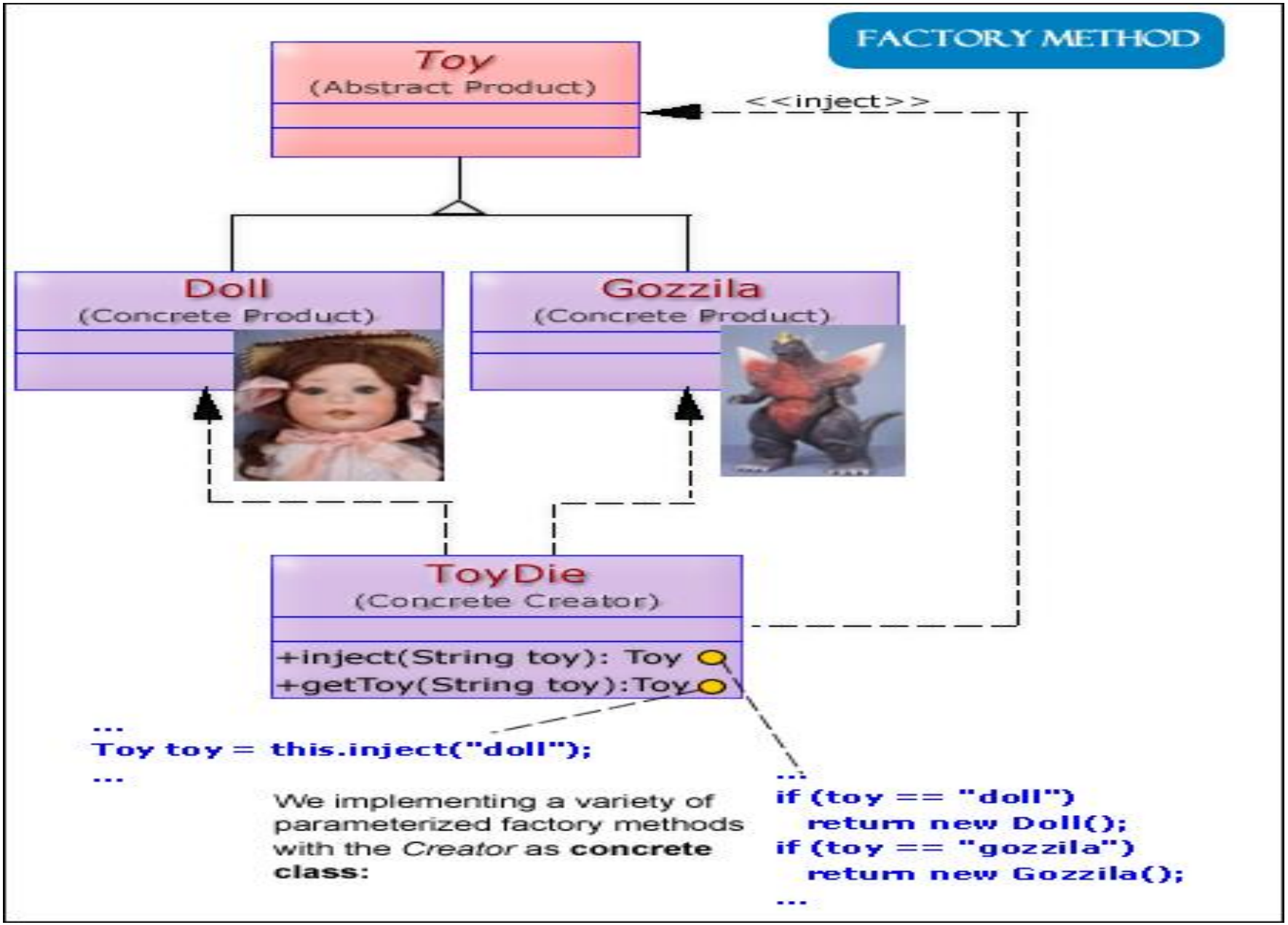
911
(Façade)

Ambulance
(SubsystemClass)

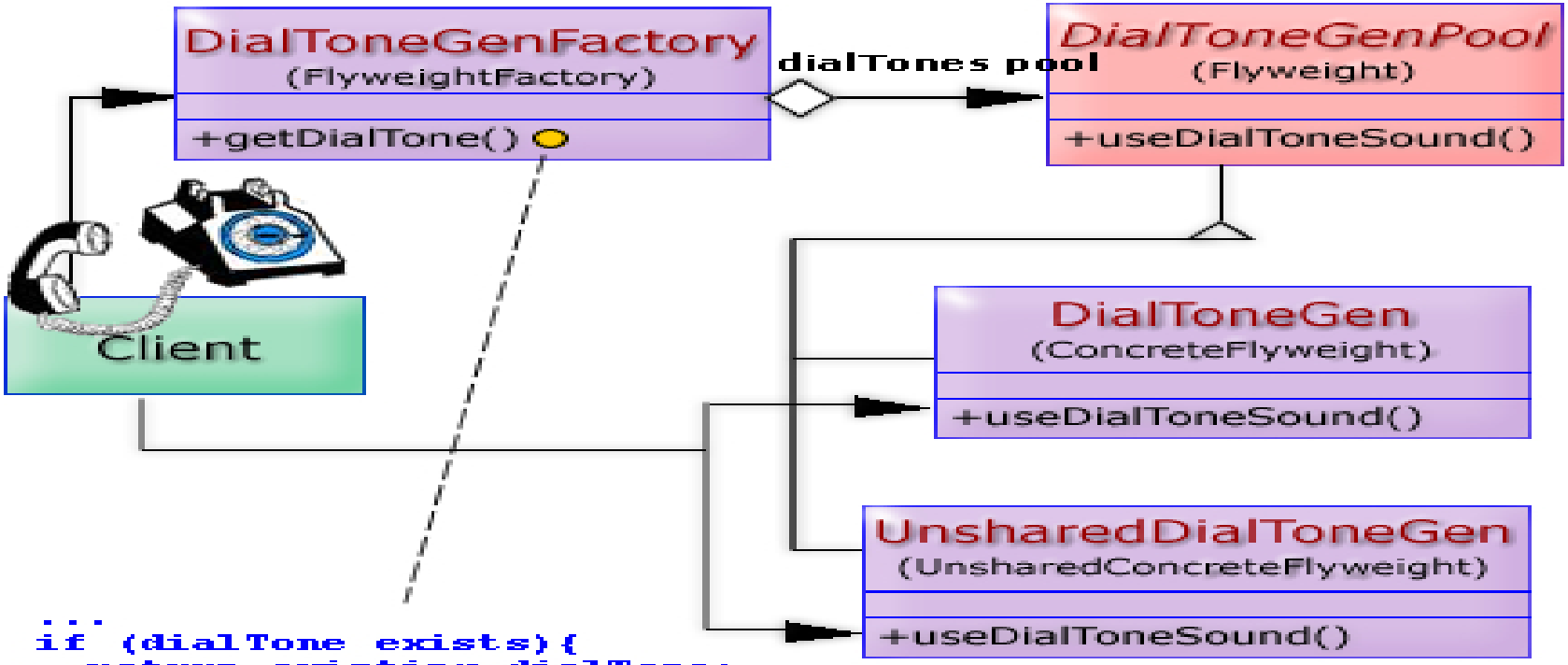
Police
(SubsystemClass)

FireBrigade
(SubsystemClass)

Emergency Service


FACTORY METHOD
Toy
(Abstract Product)
<<inject>>
Doll
(Concrete Product)
Gozzila
(Concrete Product)
ToyDie
(Concrete Creator)
+inject(String toy): Toy
+getToy(String toy):Toy
...
Toy toy = this.inject("doll");
...
We implementing a variety of parameterized factory methods with the *Creator* as **concrete class:**
...
if (toy == "doll")
return new Doll();
if (toy == "gozzila")
return new Gozzila();
...

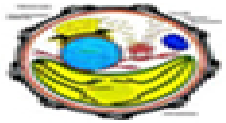
FLYWEIGHT




```
...
if (dialTone exists){
   return existing dialTone;
} else {
   create new dialTone;
   add it to pool of dialTone;
   return the new dialTone;
}
```

PROTOTYPE

Declares an interface for cloning itself.
This typically involves defining a "clone" function that returns a copy of the original object.



Cell
(Abstract Prototype)

+split()

SingleCell
(Concrete Prototype)

+SingleCell()
+SingleCell(Cell)
+split()

This is a "Copy Constructor" that does construction of the compatible object and the copying of the data.

```
...
return new SingleCell(this)
```

- The returned object has the same data and state as the original object.

SINGLETON

FootballWorld Championship
(Singleton)

- worldChampionTeam

- FootballWorldChampionship()
- getWorldChampionTeam()

```
...
return worldChampionTeam
```

- Ensure a class only has one instance, and provide a global point of access to it.


Thank you ...