EAST WEST UNIVERSITY

CSE412
Software Engineering

Nishat Tasnim Niloy

Lecturer

Department of Computer Science and Engineering

Faculty of Science and Engineering

Topic 8

Design Pattern

What is a Design Pattern?

- A (Problem, Solution) pair.

- A technique to repeat designer success.

- Borrowed from Civil and Electrical Engineering domains.

Design patterns you have already seen

- Encapsulation (Data Hiding)
- Subclassing (Inheritance)
- Iteration
- Exceptions

Encapsulation pattern

- **Problem:** Exposed fields are directly manipulated from outside, leading to undesirable dependences that prevent changing the implementation.

- **Solution:** Hide some components, permitting only stylized access to the object.

Subclassing pattern

- **Problem:** Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.

- **Solution:** Inherit default members from a superclass; select the correct implementation via run-time dispatching.

Iteration pattern

- **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.

Exception pattern

- **Problem:** Code is cluttered with error-handling code.

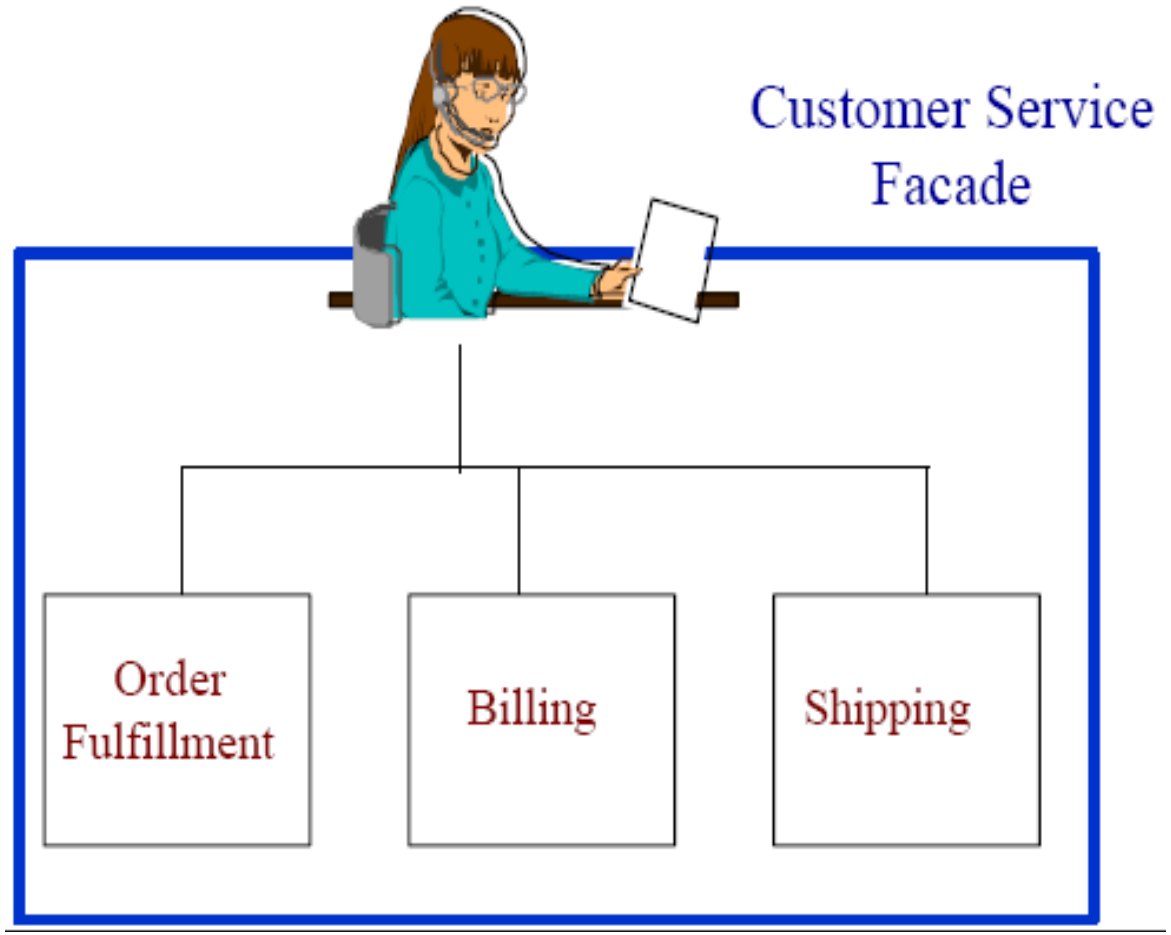
- **Solution:** Errors occurring in one part of the code should often be handled elsewhere. Use language structures for throwing and catching exceptions.

Pattern Categories

- **Creational Patterns** concern the process of object creation.
- **Structural Patterns** concern with integration and composition of classes and objects.
- **Behavioral Patterns** concern with class or object communication.

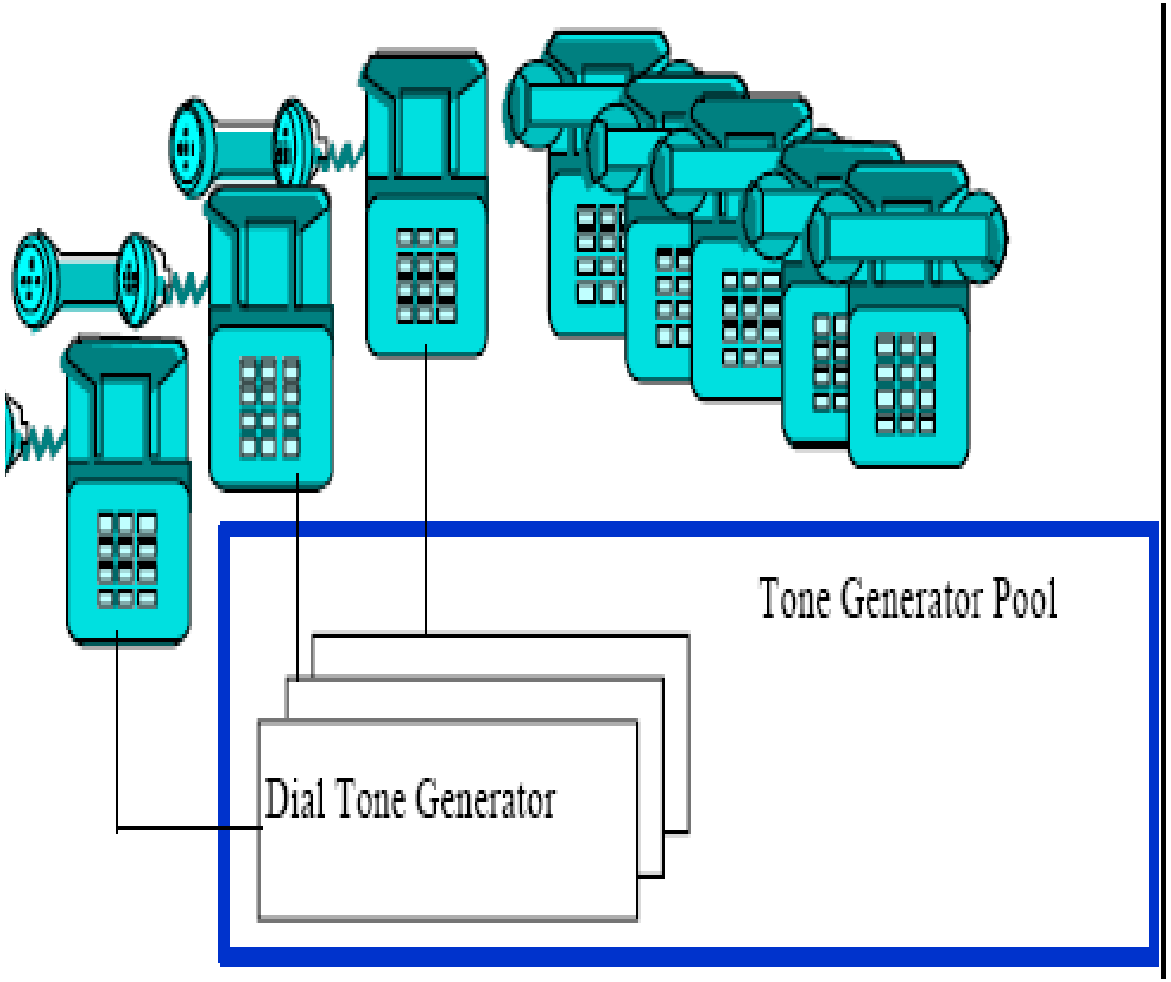
Design pattern example

Facade Pattern



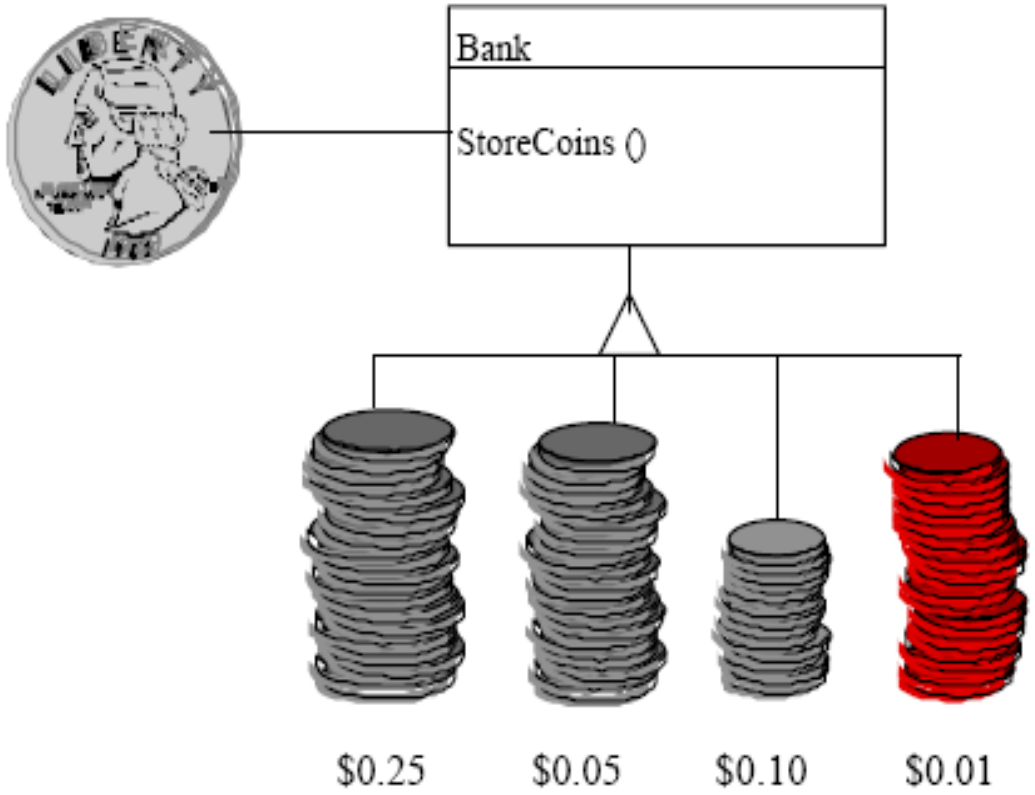
Provide a unified interface to a set of interfaces in a subsystem.

Flyweight Pattern



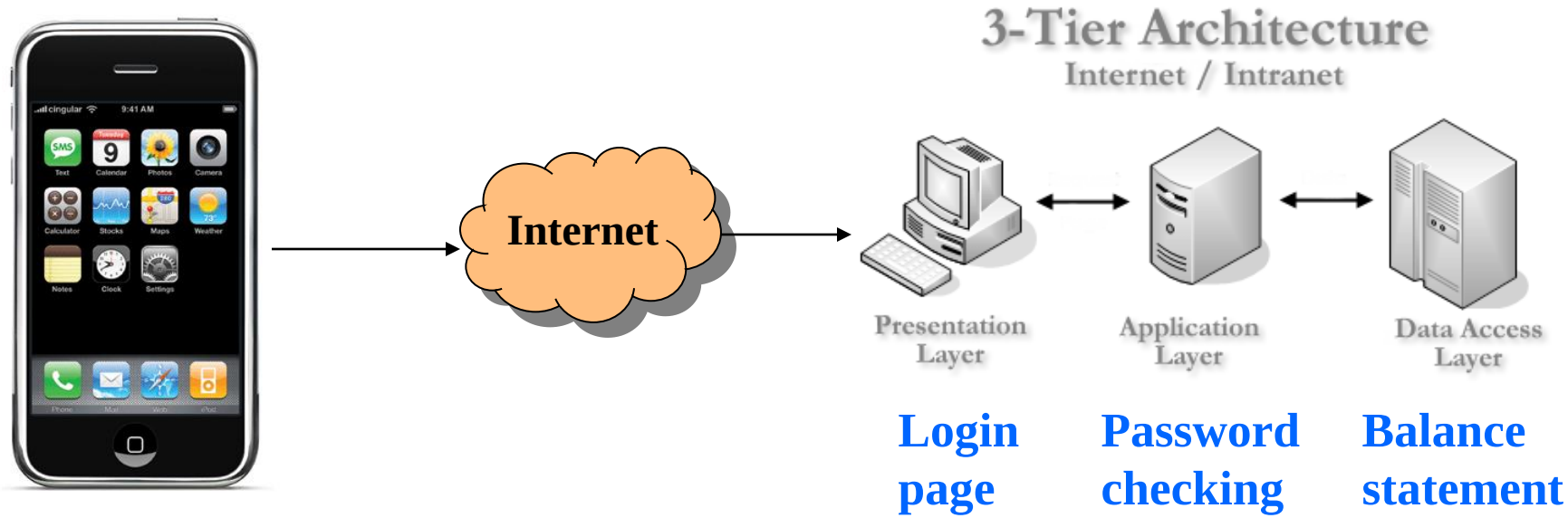
Use sharing to support large numbers of fine-grained objects efficiently

Chain of Responsibility Pattern



Chain the receiving objects and pass the request along the chain until an object handles it.

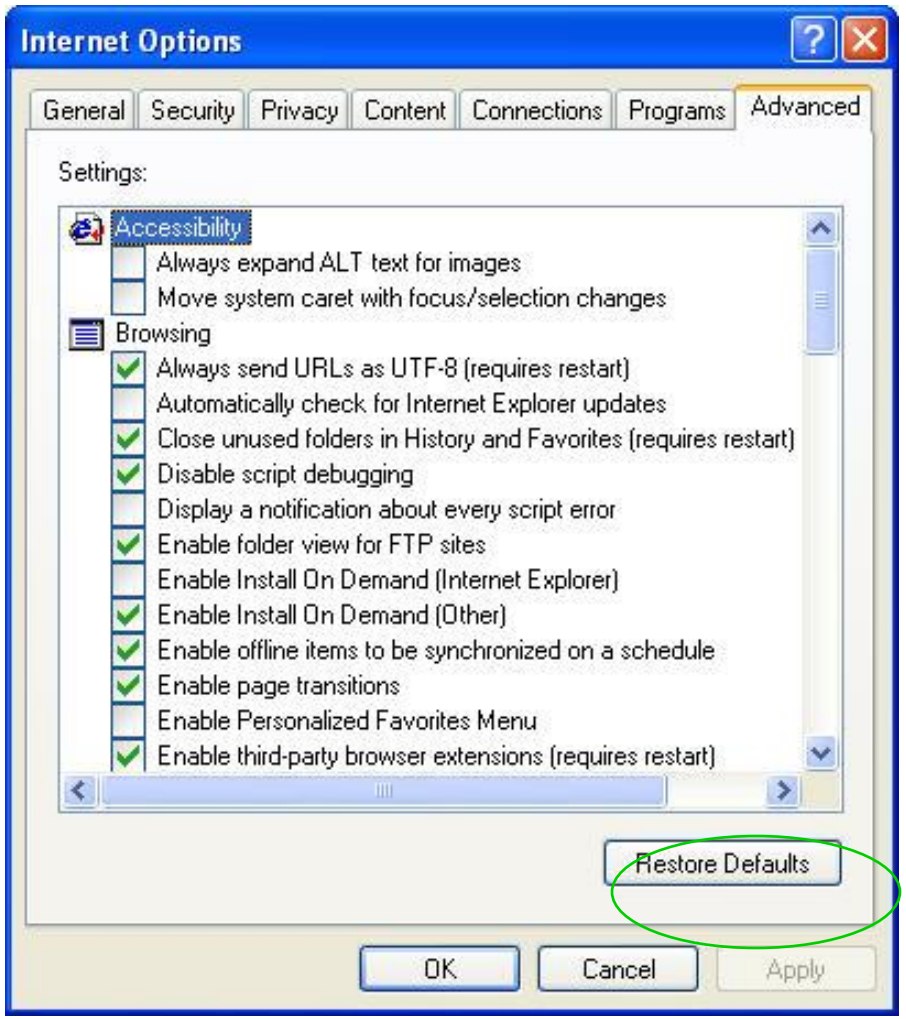
Chain of Responsibility Pattern



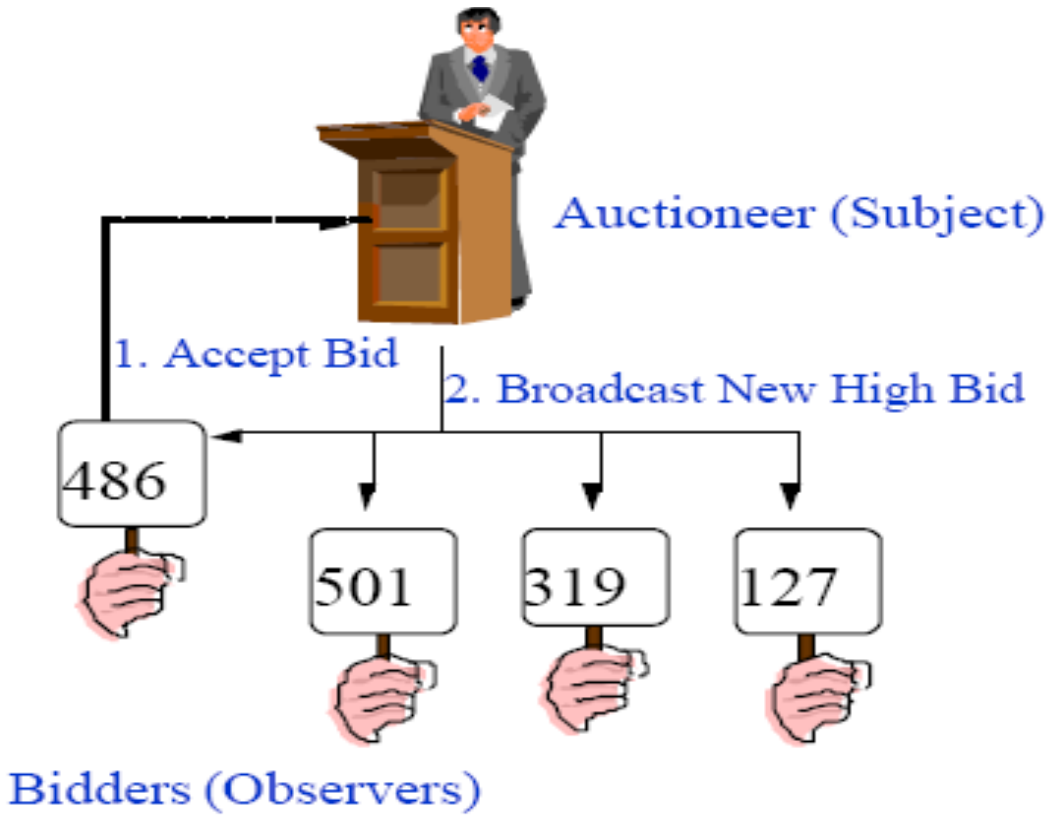
Memento Pattern

Externalize object's state so that object can be restored to this state later.

Memento Pattern

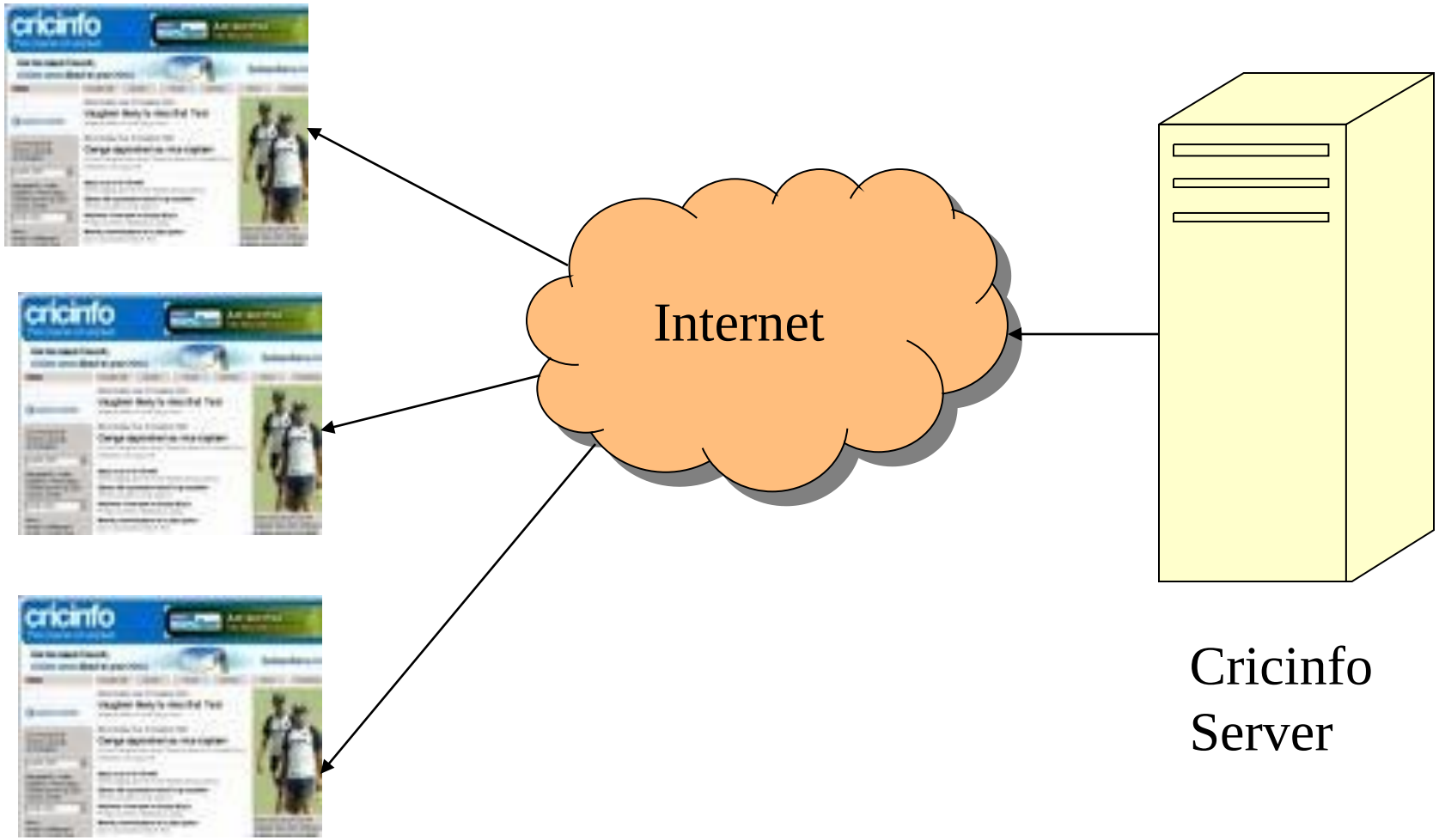


Observer Pattern

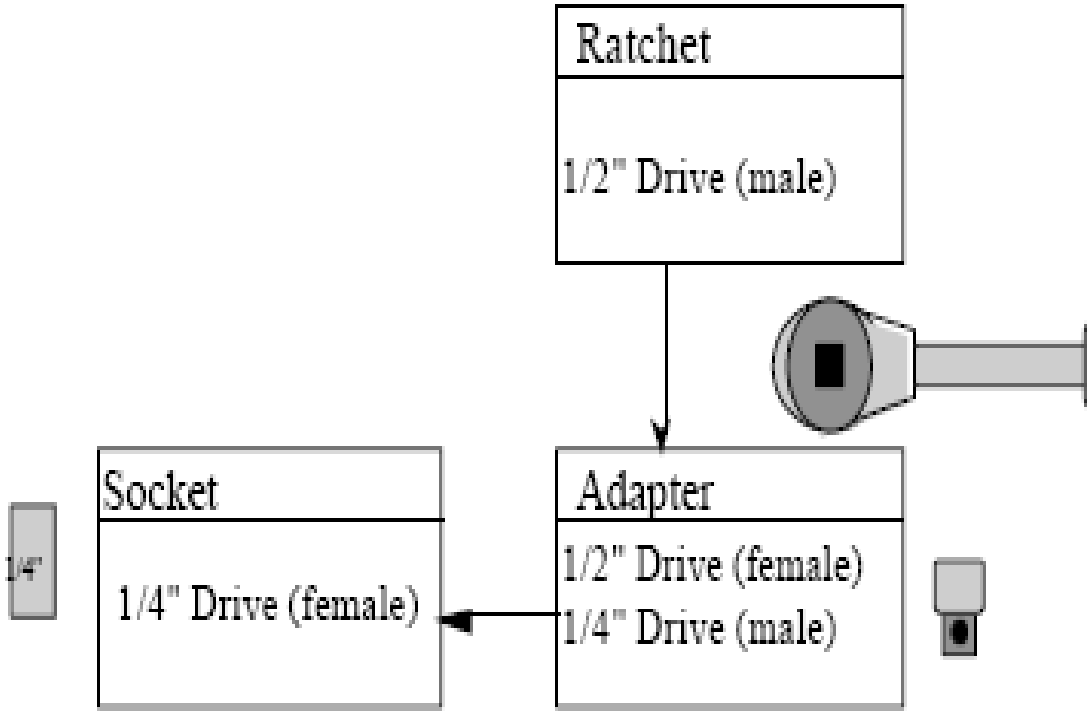


When an object changes its state, all its dependents are notified.

Observer Pattern



Adapter Pattern



Convert the interface of a class into another Interface clients expect.

Structural

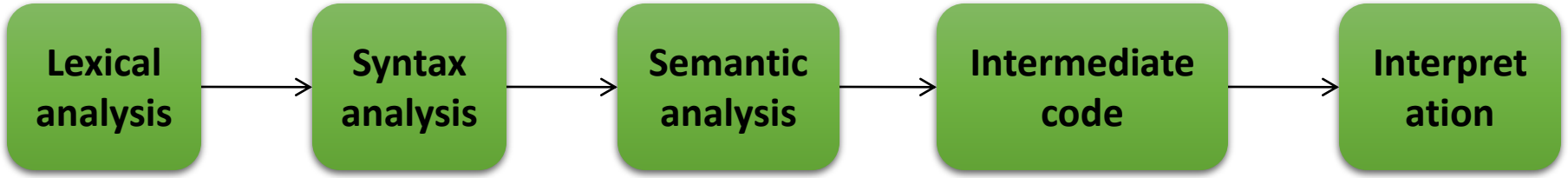
Adapter Pattern

Builder Pattern

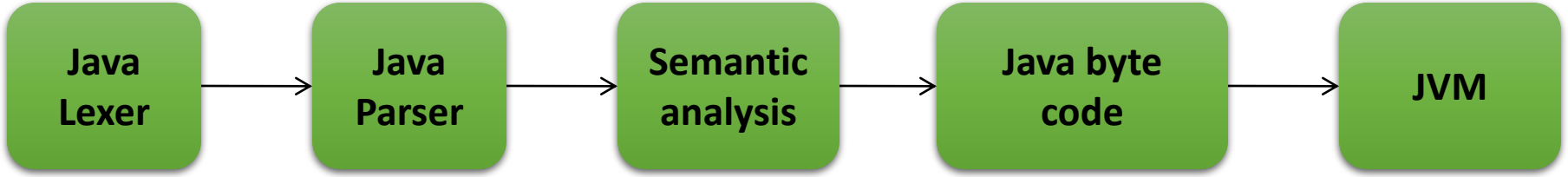
Separate the construction process of a complex object from its representation so that the same construction Process can create different representations.

Builder Pattern

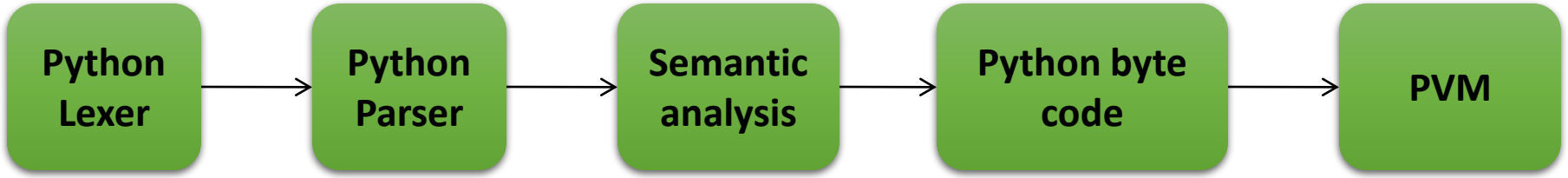
Compiler process



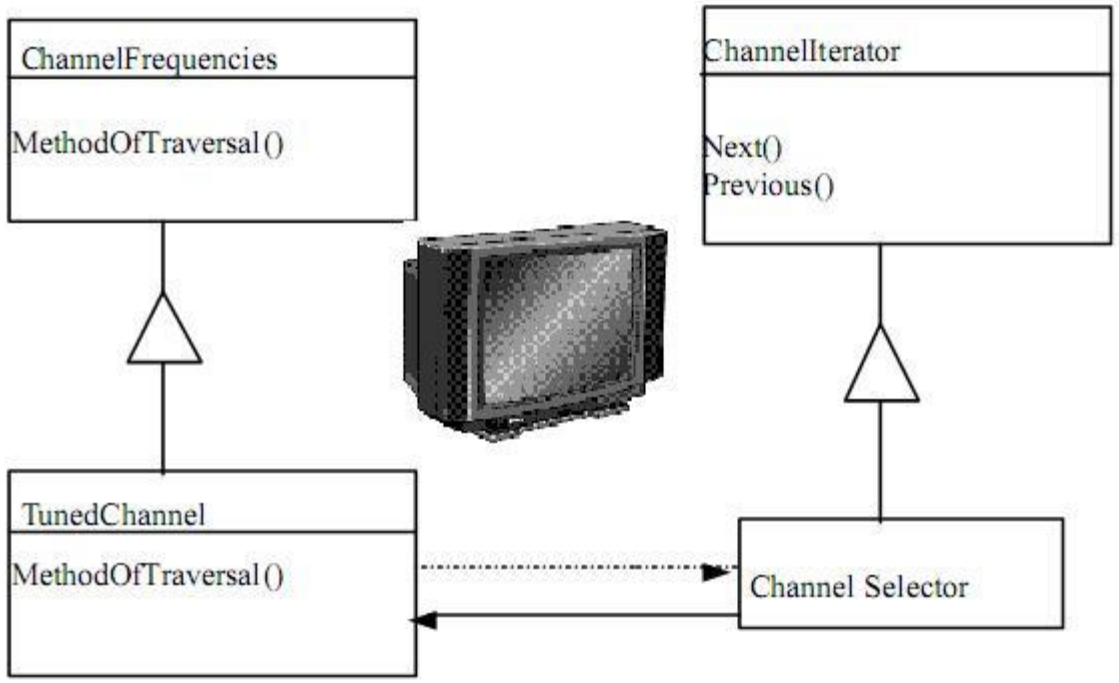
Java Compiler



Python Compiler

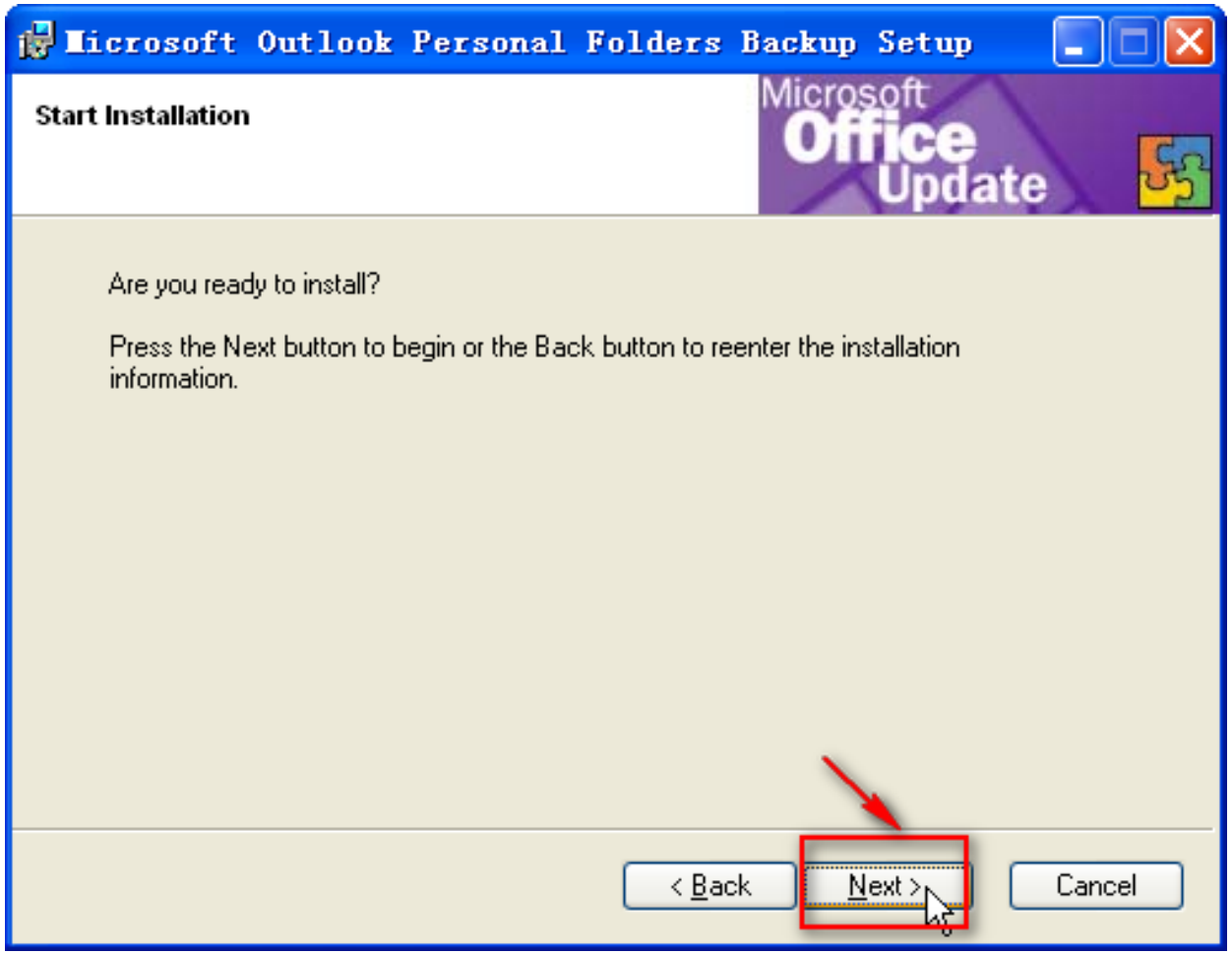


Iterator Pattern

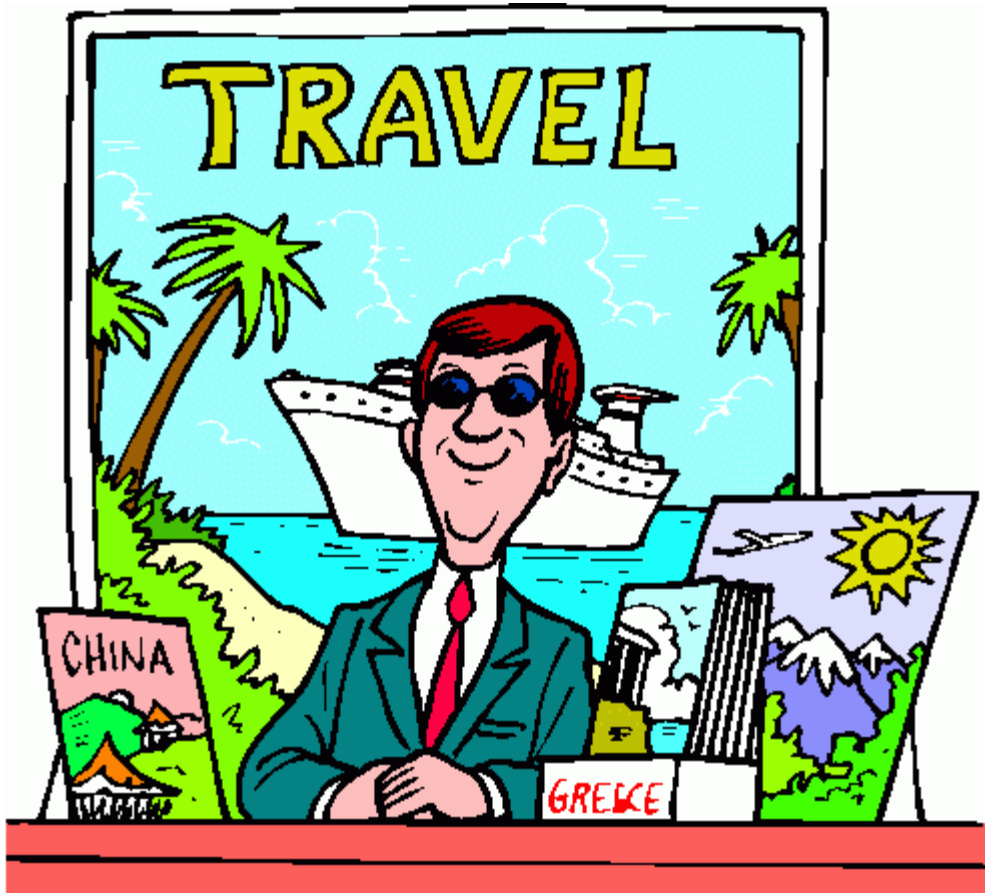


Provide a way to access the elements of a set sequentially.

Iterator Pattern

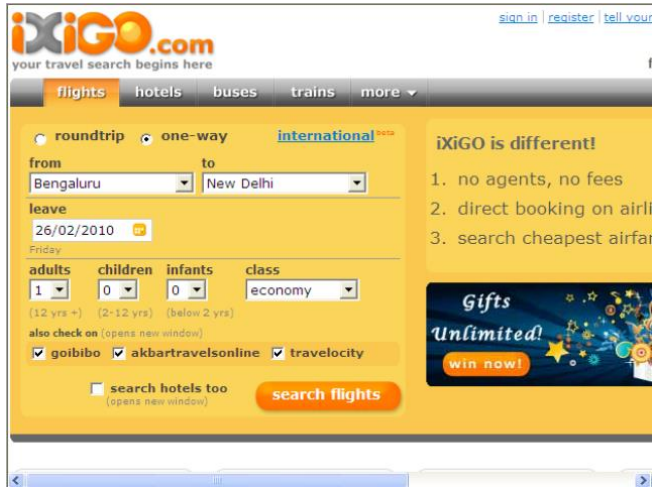


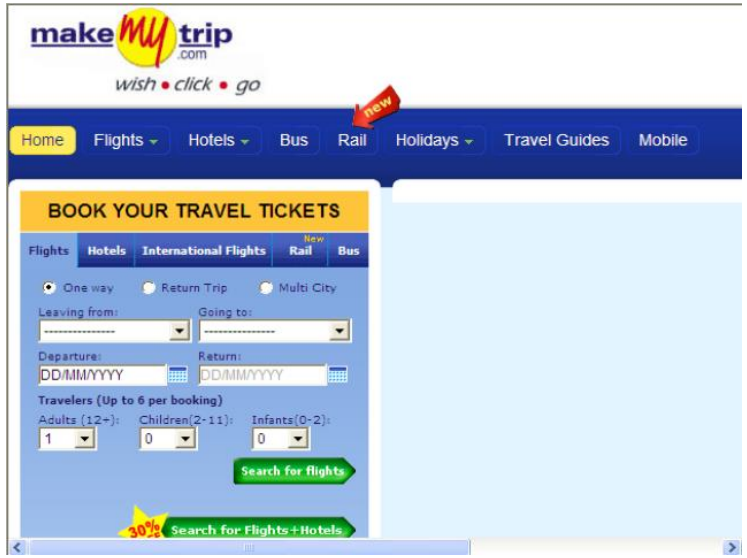
Broker Pattern



Broker component is responsible for coordinating communication between clients and remote servers.

Broker Pattern

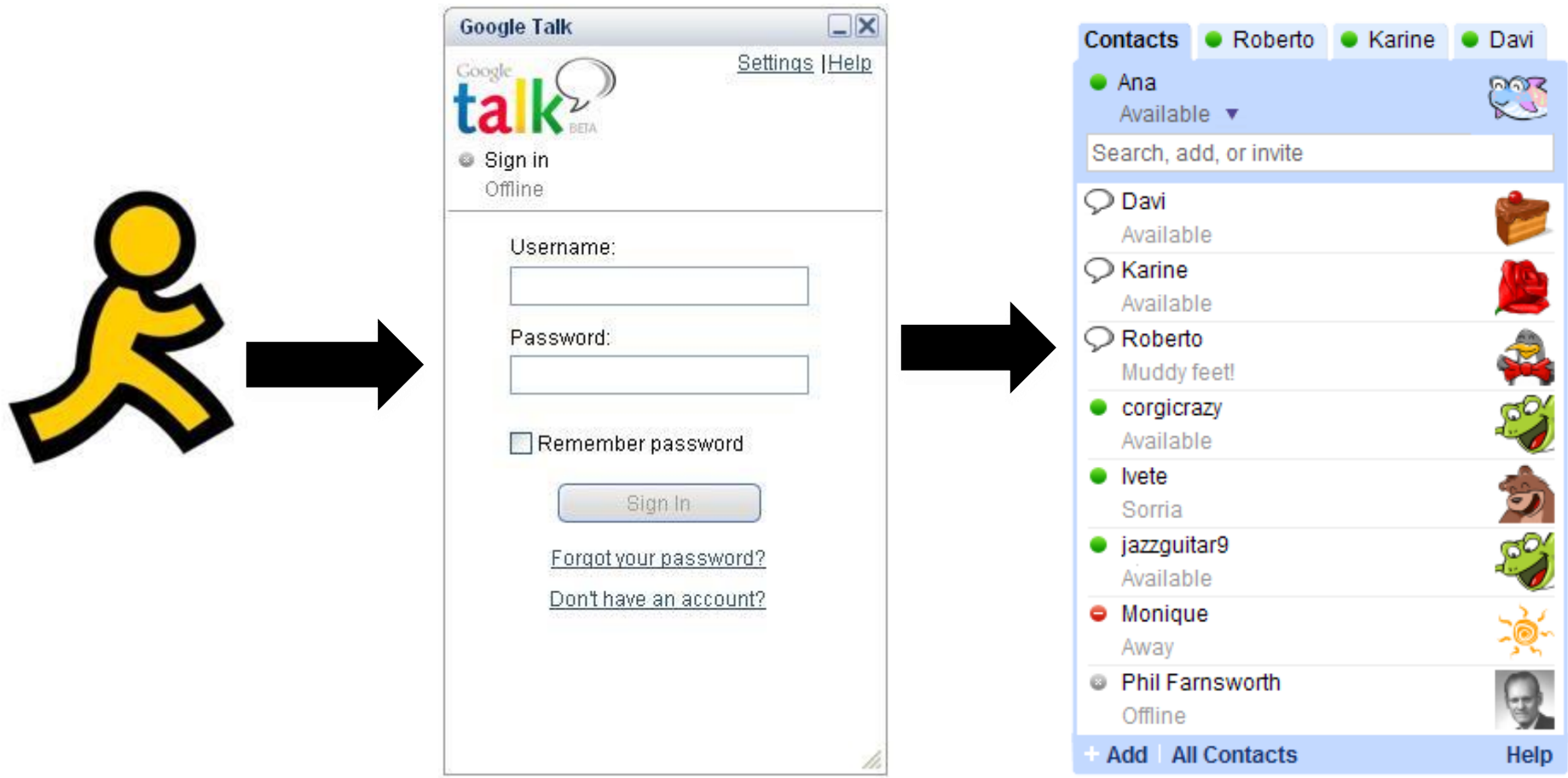




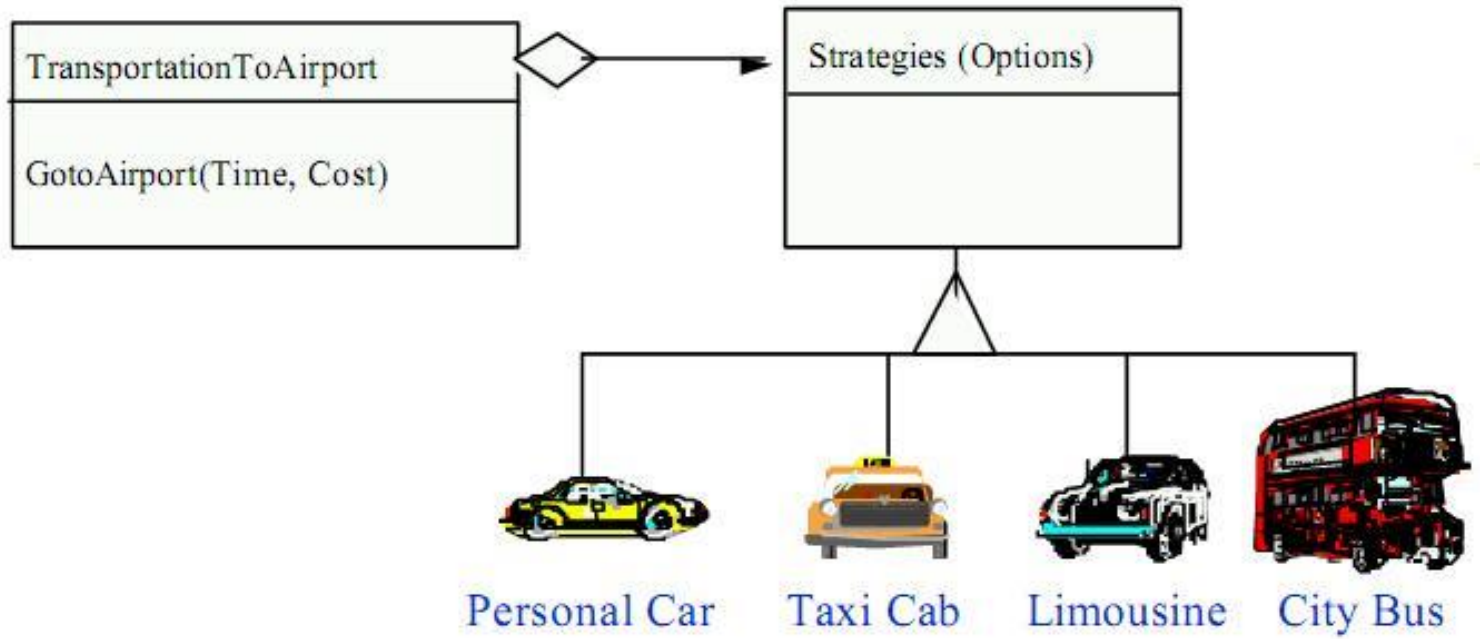
Proxy Pattern

Provide a surrogate or placeholder for another object to control access to it.

Proxy Pattern



Strategy Pattern

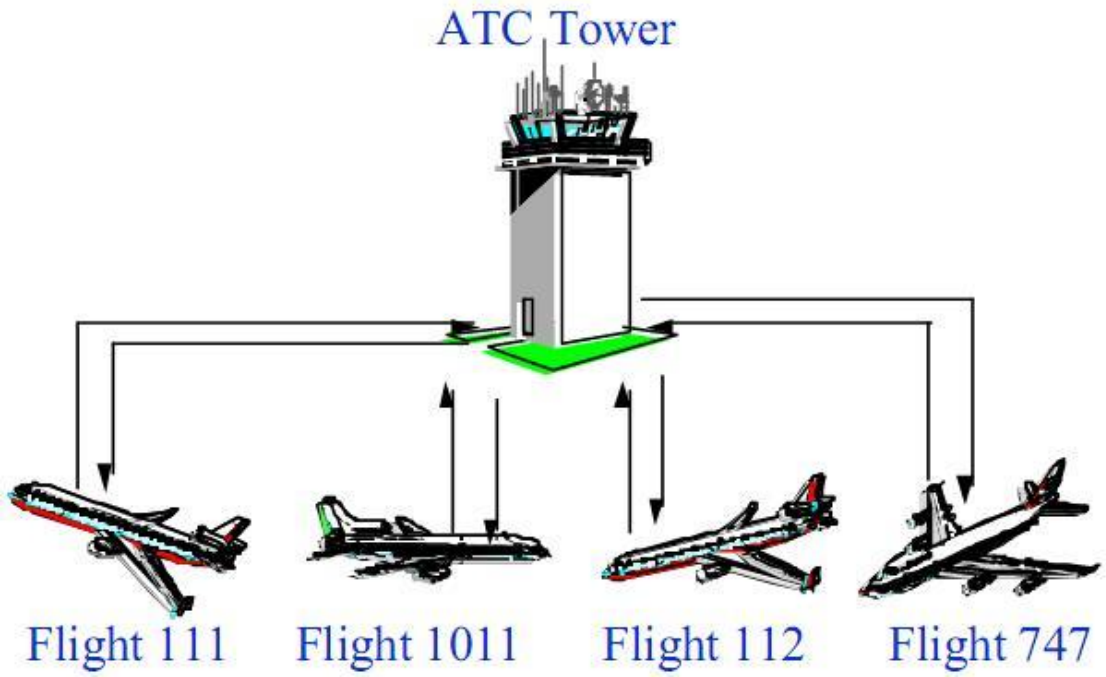


A Strategy defines a set of algorithms that can be used interchangeably.

Strategy Pattern

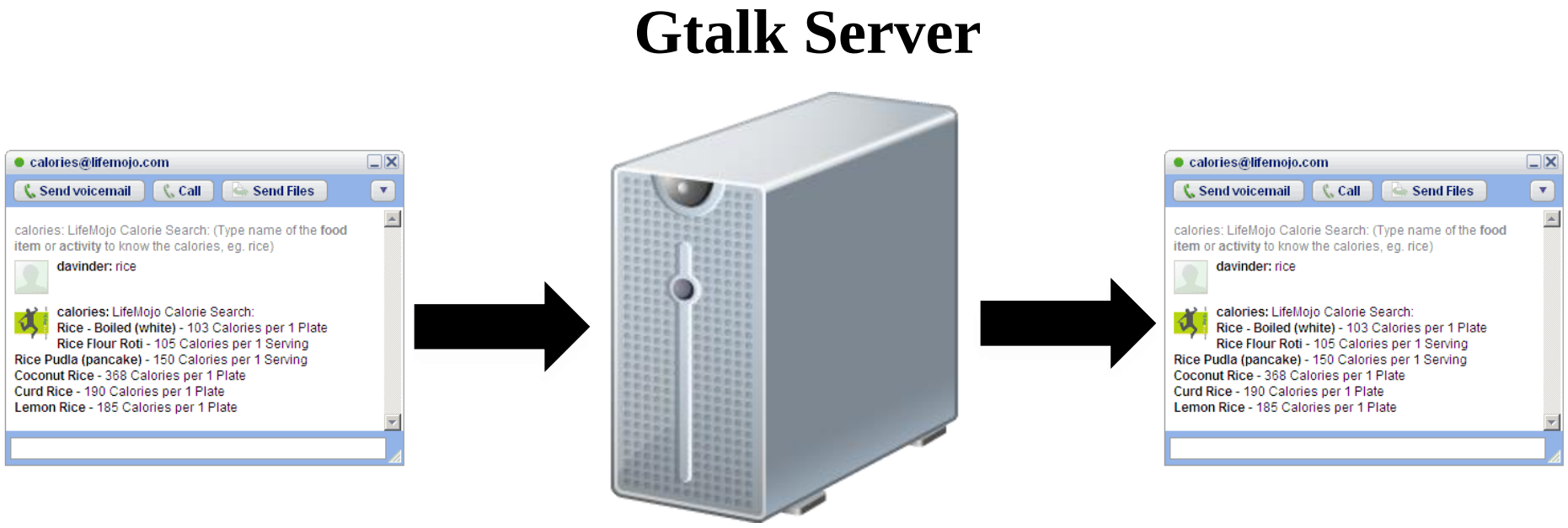
Multiple interchangeable weapons available to attack an enemy.

Mediator Pattern

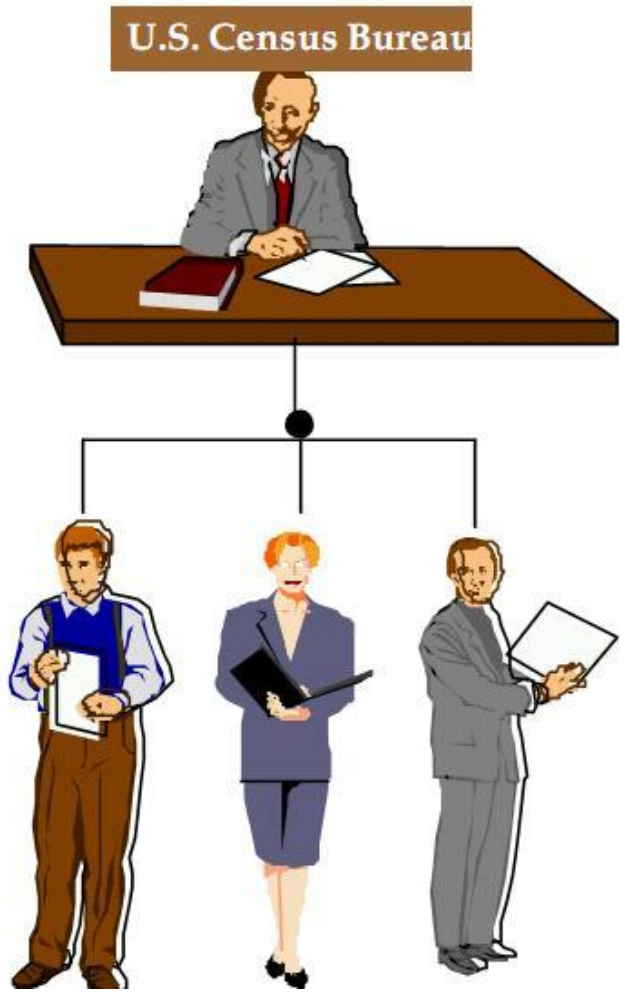


Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than one another.

Mediator Pattern



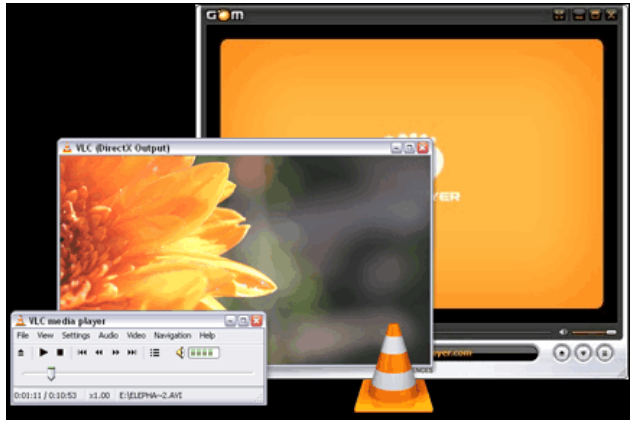
Master-Slave Pattern



Master component distributes work to identical slave components and computes a final result from the results when the slaves return.

Master-Slave Pattern

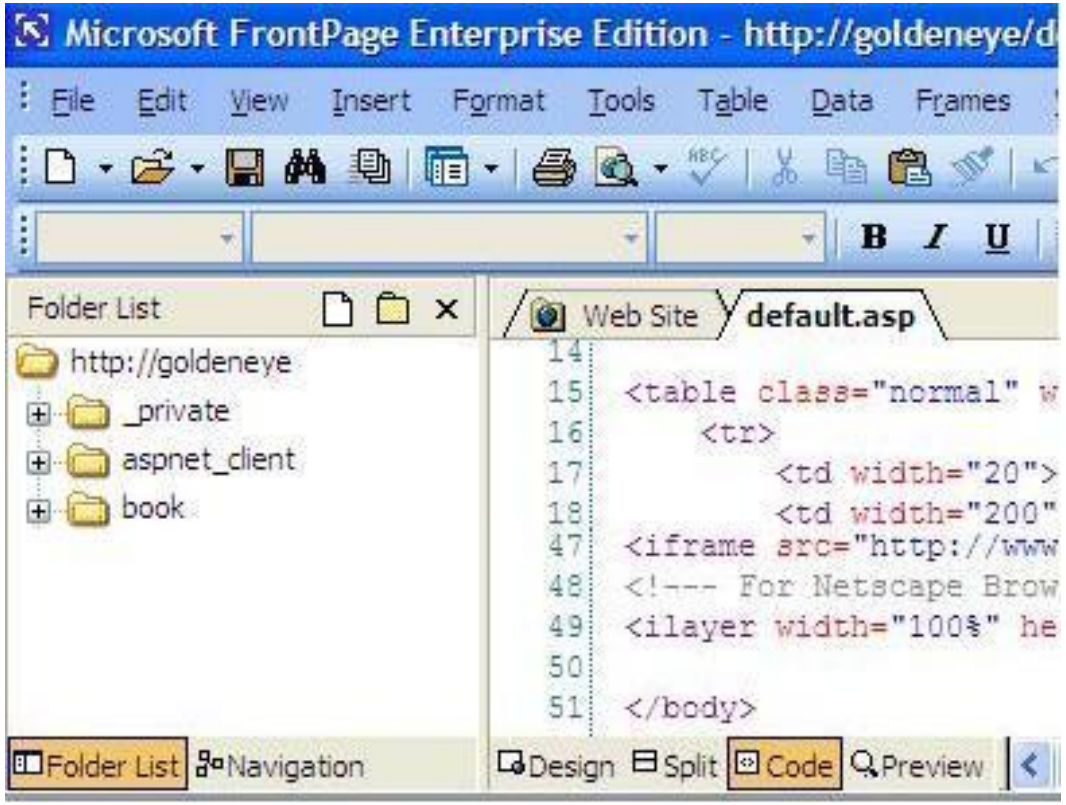
Movie players



High-resolution Game players

Graphics partitioning

MVC Pattern



MVC structures interactive applications.

Design pattern examples in Detail

Builder

An Object Creational Pattern

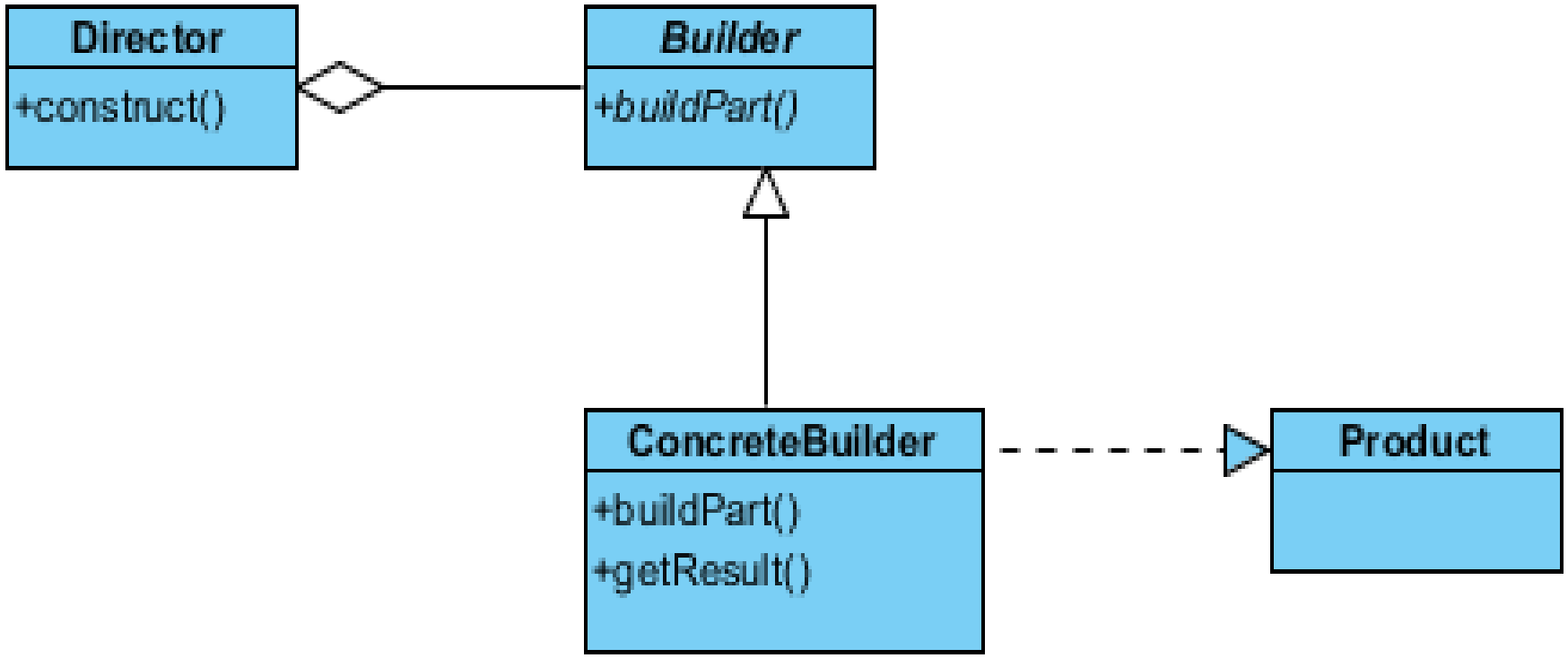
Intent / Applicability

- Separate the construction of a complex object from its representation so that the same construction process can create different representations

- Use the Builder pattern when:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - the construction process must allow different representations for the object that is constructed

Reference: Design Patterns, Gamma, et. al., Addison Wesley, 1995, pp 97-98

UML Structure



Example: building different types of airplanes

- **AerospaceEngineer**: director
- **AirplaneBuilder**: abstract builder
- **Airplane**: product
- Sample concrete builders:
  - **CropDuster**
  - **FighterJet**
  - **Glider**
  - **Airliner**

Director

```
package builder;
/** "Director" */
public class AerospaceEngineer {

	private AirplaneBuilder airplaneBuilder;

	public void setAirplaneBuilder(AirplaneBuilder ab) {
		airplaneBuilder = ab;
	}

	public Airplane getAirplane() {
		return airplaneBuilder.getAirplane();
	}

	public void constructAirplane() {
		airplaneBuilder.createNewAirplane();
		airplaneBuilder.buildWings();
		airplaneBuilder.buildPowerplant();
		airplaneBuilder.buildAvionics();
		airplaneBuilder.buildSeats();
	}
}
```

Abstract Builder

```
package builder;
/** "AbstractBuilder" */
public abstract class AirplaneBuilder {

	protected Airplane airplane;
	protected String customer;
	protected String type;

	public Airplane getAirplane() {
		return airplane;
	}

	public void createNewAirplane() {
		airplane = new Airplane(customer, type);
	}

	public abstract void buildWings();

	public abstract void buildPowerplant();

	public abstract void buildAvionics();

	public abstract void buildSeats();

}
```

Product

```
package builder;
/** "Product" */
public class Airplane {

	private String type;
	private float wingspan;
	private String powerplant;
	private int crewSeats;
	private int passengerSeats;
	private String avionics;
	private String customer;

	Airplane (String customer, String type){
		this.customer = customer;
		this.type = type;
	}

	public void setWingspan(float wingspan) {
		this.wingspan = wingspan;
	}
```

Product (continued)

```
        public void setPowerplant(String powerplant) {
                this.powerplant = powerplant;
        }

        public void setAvionics(String avionics) {
                this.avionics = avionics;
        }

        public void setNumberSeats(int crewSeats, int passengerSeats) {
                this.crewSeats = crewSeats;
                this.passengerSeats = passengerSeats;
        }

        public String getCustomer() {
                return customer;
        }

        public String getType() {
                return type;
        }
}
```

Concrete Builder 1

```
package builder;
/** "ConcreteBuilder" */
public class CropDuster extends AirplaneBuilder {

	CropDuster (String customer){
		super.customer = customer;
		super.type = "Crop Duster v3.4";
	}

	public void buildWings() {
		airplane.setWingspan(9f);
	}

	public void buildPowerplant() {
		airplane.setPowerplant("single piston");
	}

	public void buildAvionics() {}

	public void buildSeats() {
		airplane.setNumberSeats(1,1);
	}

}
```

Concrete Builder 2

```
package builder;
/** "ConcreteBuilder" */
public class FighterJet extends AirplaneBuilder {

        FighterJet (String customer){
                super.customer = customer;
                super.type = "F-35 Lightning II";
        }

        public void buildWings() {
                airplane.setWingspan(35.0f);
        }

        public void buildPowerplant() {
                airplane.setPowerplant("dual thrust vectoring");
        }

        public void buildAvionics() {
                airplane.setAvionics("military");
        }

        public void buildSeats() {
                airplane.setNumberSeats(1,0);

        }

}
```

Concrete Builder 3

```
package builder;
/** "ConcreteBuilder" */
public class Glider extends AirplaneBuilder {

	Glider (String customer){
		super.customer = customer;
		super.type = "Glider v9.0";
	}

	public void buildWings() {
		airplane.setWingspan(57.1f);
	}

	public void buildPowerplant() {}

	public void buildAvionics() {}

	public void buildSeats() {
		airplane.setNumberSeats(1,0);

	}

}
```

Concrete Builder 4

```
package builder;
/** "ConcreteBuilder" */
public class Airliner extends AirplaneBuilder {

	Airliner (String customer){
		super.customer = customer;
		super.type = "787 Dreamliner";
	}

	public void buildWings() {
		airplane.setWingspan(197f);
	}

	public void buildPowerplant() {
		airplane.setPowerplant("dual turbofan");
	}

	public void buildAvionics() {
		airplane.setAvionics("commercial");
	}

	public void buildSeats() {
		airplane.setNumberSeats(8,289);

	}

}
```

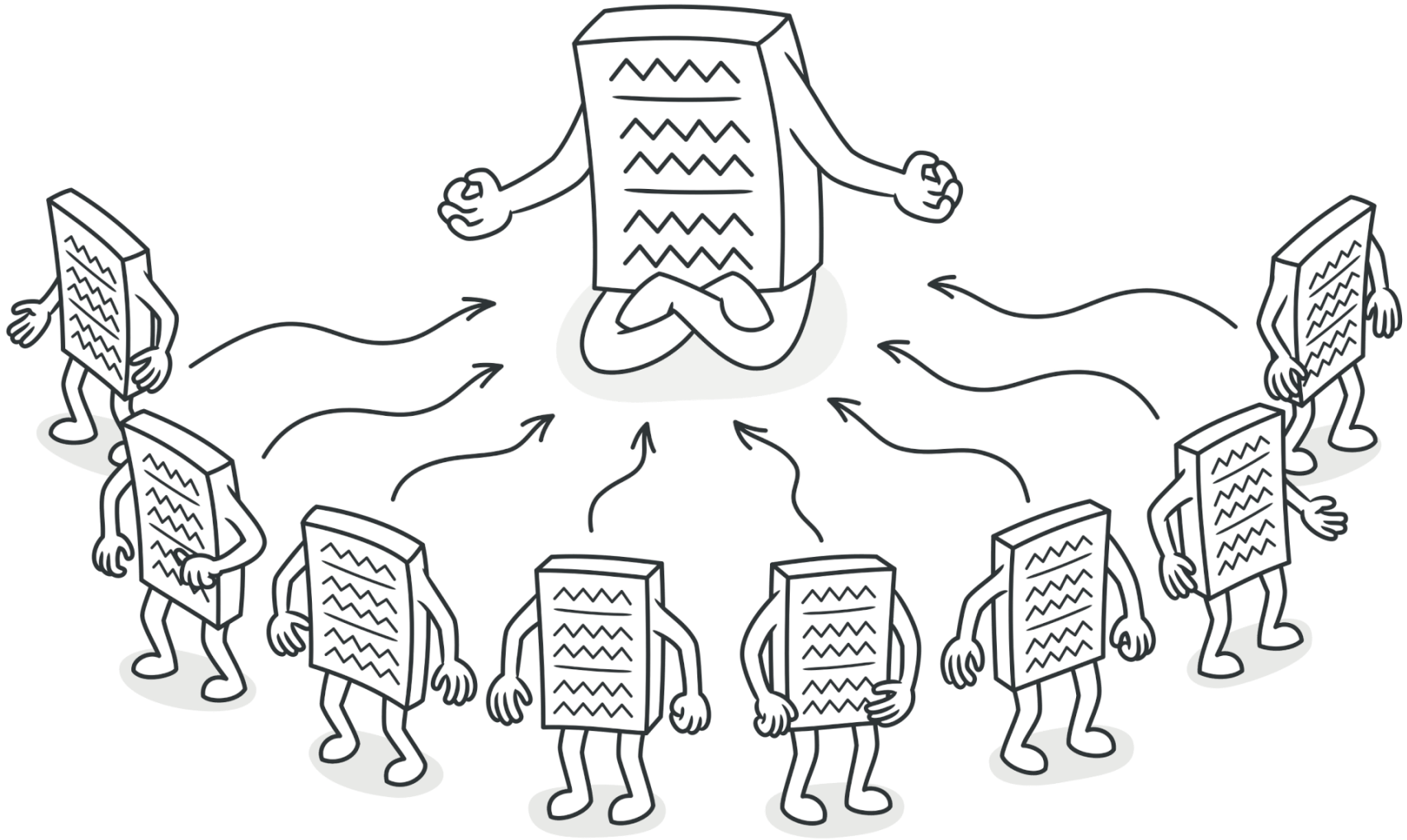
Client Application

```
package builder;
/** Application in which given types of airplanes are being constructed.
*/
public class BuilderExample {
      public static void main(String[] args) {
            // instantiate the director (hire the engineer)
            AerospaceEngineer aero = new AerospaceEngineer();

            // instantiate each concrete builder (take orders)
            AirplaneBuilder crop = new CropDuster("Farmer Joe");
            AirplaneBuilder fighter = new FighterJet("The Navy");
            AirplaneBuilder glider = new Glider("Tim Rice");
            AirplaneBuilder airliner = new Airliner("United Airlines");

            // build a CropDuster
            aero.setAirplaneBuilder(crop);
            aero.constructAirplane();
            Airplane completedCropDuster = aero.getAirplane();
            System.out.println(completedCropDuster.getType() +
                        " is completed and ready for delivery to " +
                        completedCropDuster.getCustomer());

            // the other 3 builds removed to fit the code on one slide
      }
}
```

Builder: Advantages / Disadvantages

- Advantages:
  - Allows you to vary a product's internal representation
  - Encapsulates code for construction and representation
  - Provides control over steps of construction process

- Disadvantages:
  - Requires creating a separate Concrete Builder for each different type of Product



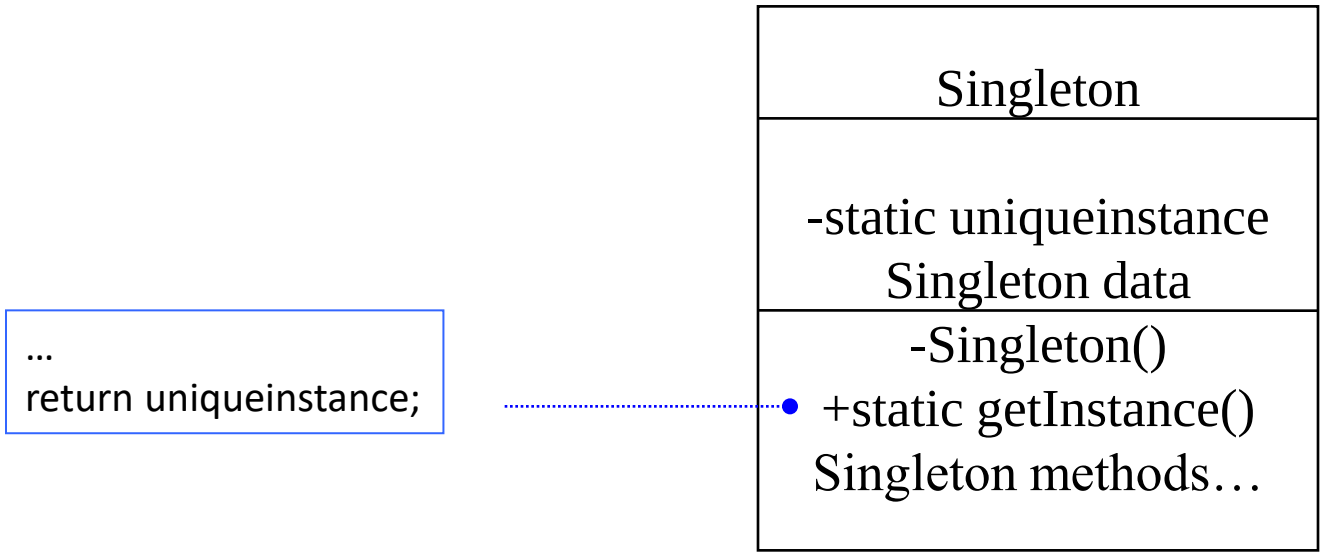
Singleton Pattern

Singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it
- Motivation
  - Important for some classes to have exactly one instance. E.g., although there are many printers, should just have one print spooler
  - Ensure only one instance available and easily accessible
    - global variables gives access, but doesn’t keep you from instantiating many objects
  - Give class responsibility for keeping track of its sole instance

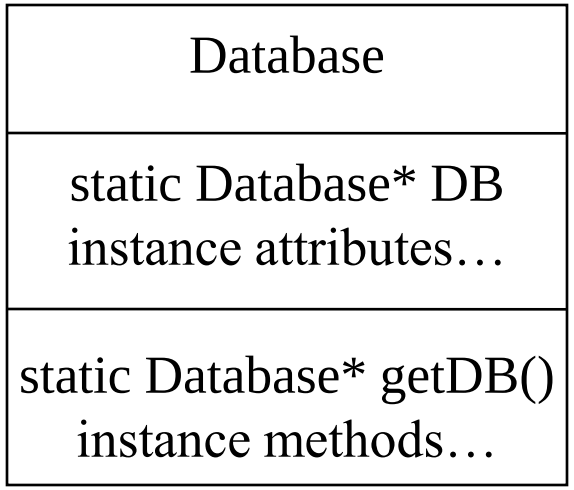
Design Solution

- Defines a getInstance() operation that lets clients access its unique instance
- May be responsible for creating its own unique instance



Singleton Example (Java)

- Database




```
public class Database {
private static Database DB;
   ...
private Database() { ... }
public static Database getDB() {
   if (DB == null)
     DB = new Database();
   return DB;
}
   ...
}
```

In application code…

```
Database db = Database.getDB();
db.someMethod();
```

Singleton Example (C++)

```
class Database
{
private:
   static Database *DB;
   ...
   private Database() { ... }
public:
   static Database *getDB()
   { if (DB == NULL)
       DB = new Database());
     return DB;
   }
   ...
}
Database *Database::DB=NULL;
```

In application code…

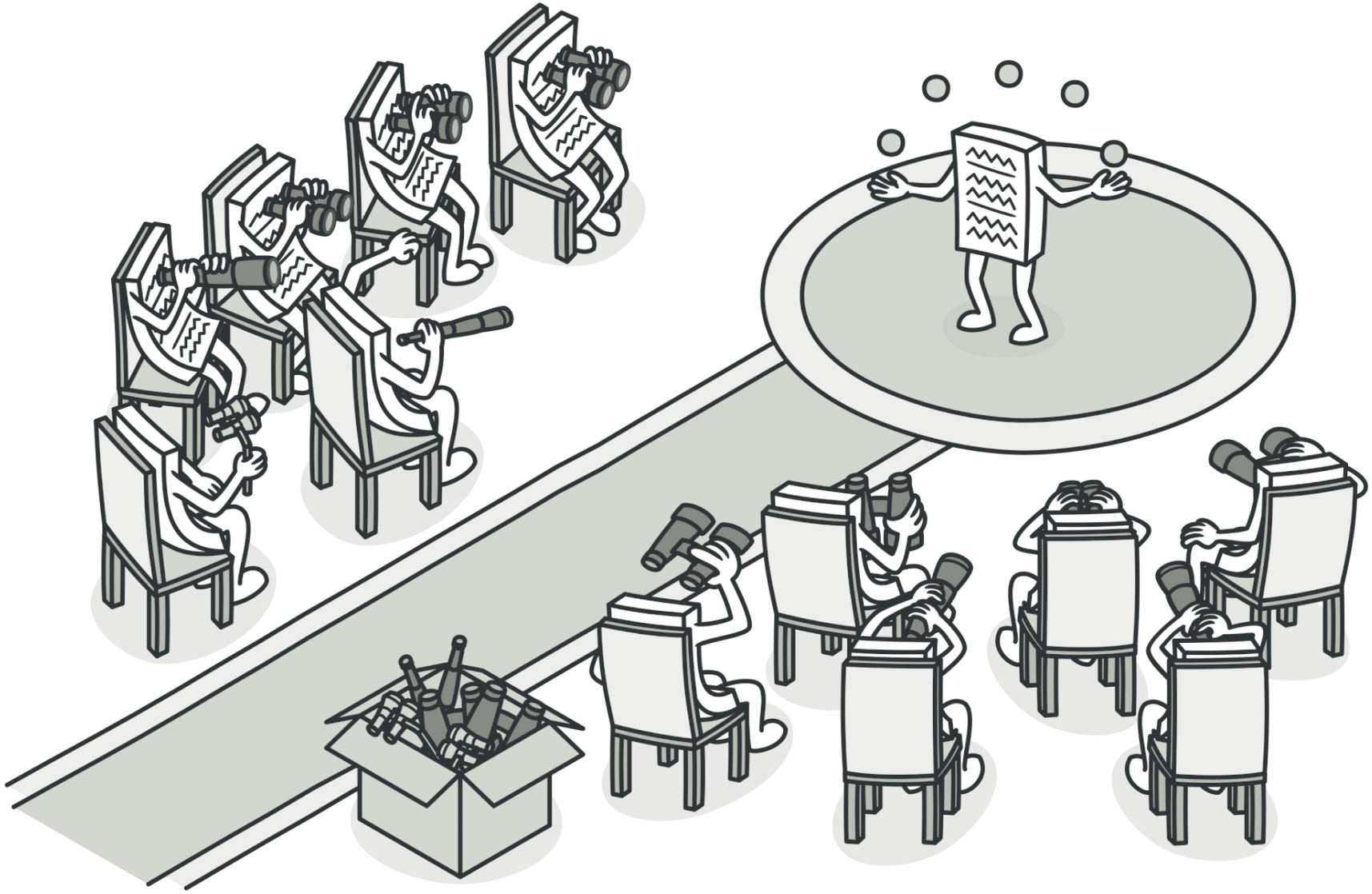
```
Database *db =
       Database.getDB();
Db->someMethod();
```

Implementation

- Declare all of class's constructors private
  - prevent other classes from directly creating an instance of this class
- Hide the operation that creates the instance behind a class operation (getInstance)
- Variation: Since creation policy is encapsulated in getInstance, it is possible to vary the creation policy

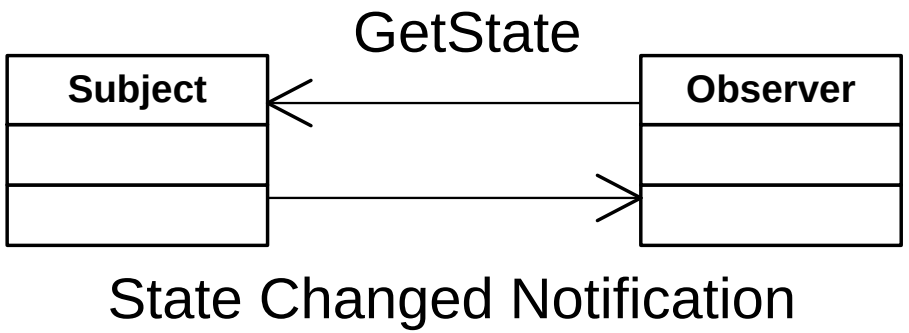
Singleton Consequences

- Ensures only one (e.g., Database) instance exists in the system
- Can maintain a pointer (need to create object on first get call) or an actual object
- Can also use this pattern to control fixed multiple instances
- Much better than the alternative: global variables



Observer Design Pattern

Problem

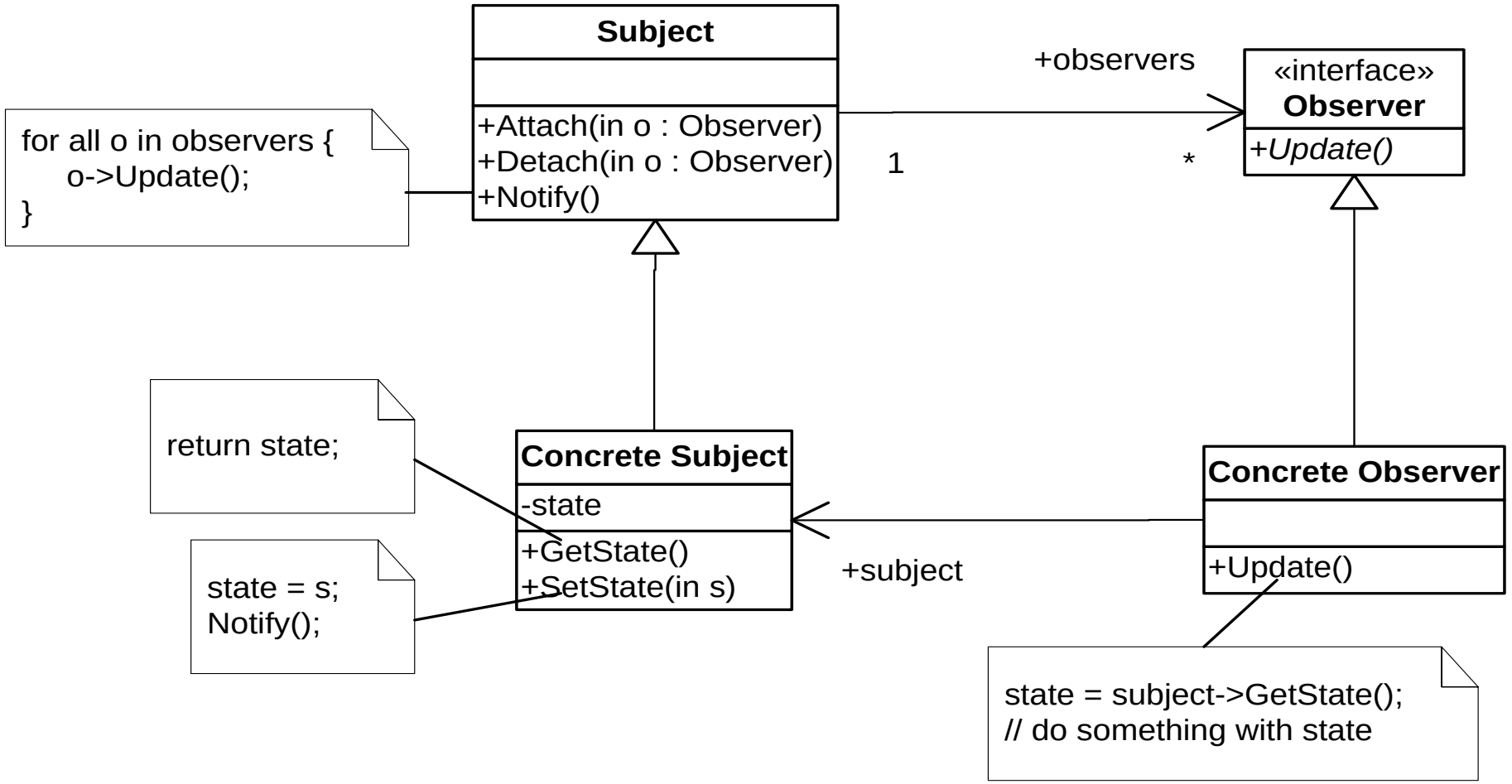


- An object (Observer AKA: Listener) needs to be notified whenever another object (Subject) changes state so it can:
  - Keep its own state in sync with the Subject

  or
  - Perform some action in response to changes in the Subject

- Example: Synchronizing data in Inventory Tracker GUI

Solution


Subject
+Attach(in o : Observer)
+Detach(in o : Observer)
+Notify()
+observers
1
*
«interface»
Observer
+Update()
for all o in observers {
o->Update();
}
return state;
state = s;
Notify();
Concrete Subject
-state
+GetState()
+SetState(in s)
+subject
Concrete Observer
+Update()
state = subject->GetState();
// do something with state

Consequences

- Can flexibly add and remove observers to an object
- Subject is not tightly coupled to Observer
  - This is an example of dependency inversion
- Supports broadcast communication
- Changing the state of an object can be far more expensive than you expect
- Can eliminate the call back from Observer to subject by passing an Event object that contains the necessary information about the change

Known Uses: Session listeners

- A `Session` object stores information about the current user session
  - What user is logged in and what their privileges are
  - Time at which the session started
  - History
    - URL history in web browser
    - Command history for undo/redo
- Other parts of the application might need to be notified when the Session state changes
  - User logged in
  - User logged out

Known Uses: Java Observers

- Support for the Observer pattern is built into Java

```
interface Observer {
   void update(Observable o, Object arg);
}

class Observable {
   void addObserver(Observer o) { … }
   void deleteObserver(Observer o) { … }
   void notifyObservers(Object arg) { … }
}
```

PropertyChangeListener in Java

- Observer and Observable were deprecated in Java 9
- Use PropertyChangeListener instead
  - Available since JDK 1.1
  - More flexible than Observer and Observable
    - Allows you to pass a PropertyChangeEvent so the Observer doesn't have to call back on the observed object to find out what changed
- Many other classes you can use instead of PropertyChangeListener for specific types of events
  - See EventListener and it's subinterfaces and implementing classes

Java Observer Using PropertyChangeListener

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

public class Screen implements PropertyChangeListener {
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        String message = String.format("Changed %s from %s to %s",
                evt.getPropertyName(),
                evt.getOldValue().toString(),
                evt.getNewValue().toString());
        System.out.println(message);
    }
}
```

Many Ways to Implement an Observable Object Using PropertyChangeListener

- Role your own
- Use PropertyChangeSupport to manage listeners (observers)
  - Simple way: Extend PropertyChangeSupport
  - **My preferred way:**
    - **Create your own “Observable” interface with add and remove listener methods**
    - **Delegate add and remove methods to an instance of PropertychangeSupport**

Observable Interface

- This is an interface you create, not the deprecated Java Observable class

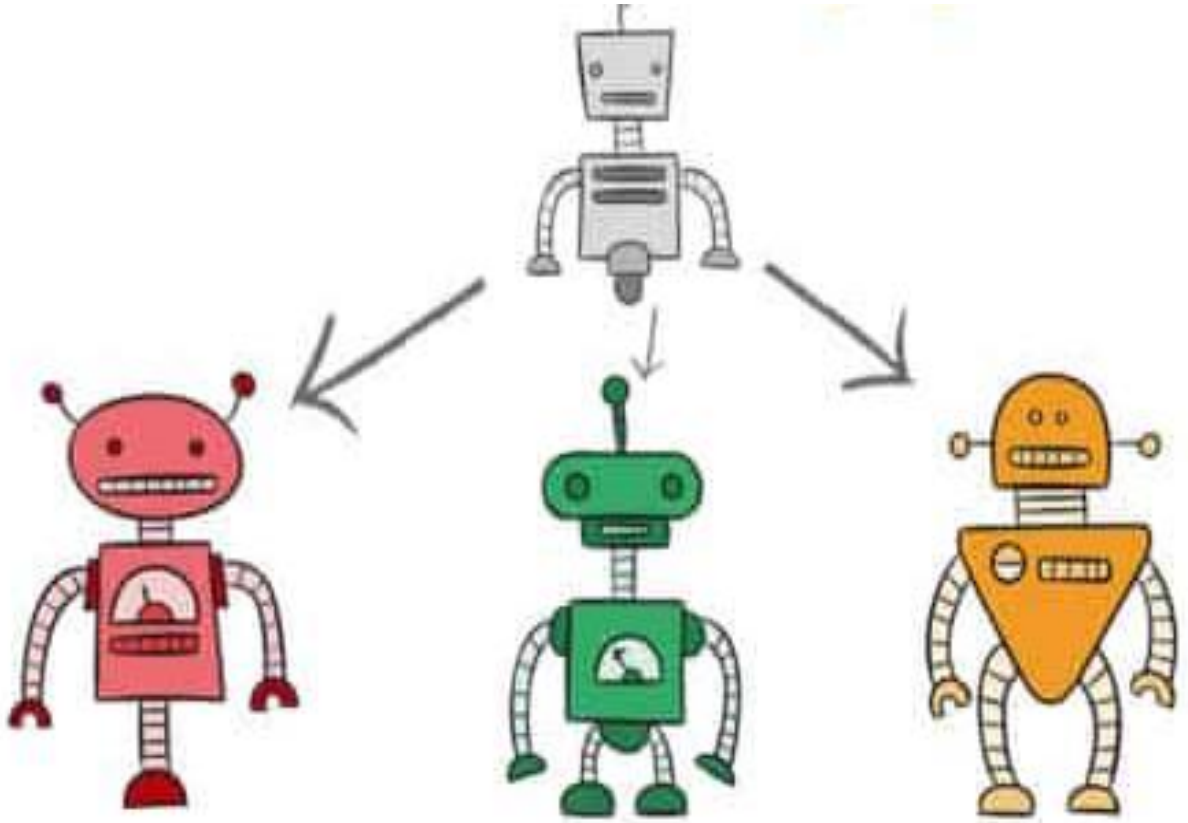
```
import java.beans.PropertyChangeListener;

public interface Observable {
    void addListener(PropertyChangeListener listener);
    void removeListener(PropertyChangeListener listener);
}
```

Observable Class

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class DataStore implements Observable {

    private PropertyChangeSupport listenerManager = new PropertyChangeSupport(this);

    private String data;

    @Override
    public void addListener(PropertyChangeListener listener) {
        listenerManager.addPropertyChangeListener(listener);
    }

    @Override
    public void removeListener(PropertyChangeListener listener) {
        listenerManager.removePropertyChangeListener(listener);
    }


    public String getData(){
        return data;
    }

    public void setData(String data){
        PropertyChangeEvent event = new PropertyChangeEvent(this, "data", this.data, data);

        this.data = data;

        listenerManager.firePropertyChange(event);
    }
}
```

Observable Class (continued)

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class DataStore implements Observable {

    ...

    private String data;

    ...


    public String getData()
    {
        return data;
    }

    public void setData(String data)
    {
        PropertyChangeEvent event = new PropertyChangeEvent(this, "data", this.data, data);

        this.data = data;

        listenerManager.firePropertyChange(event);
    }
}
```

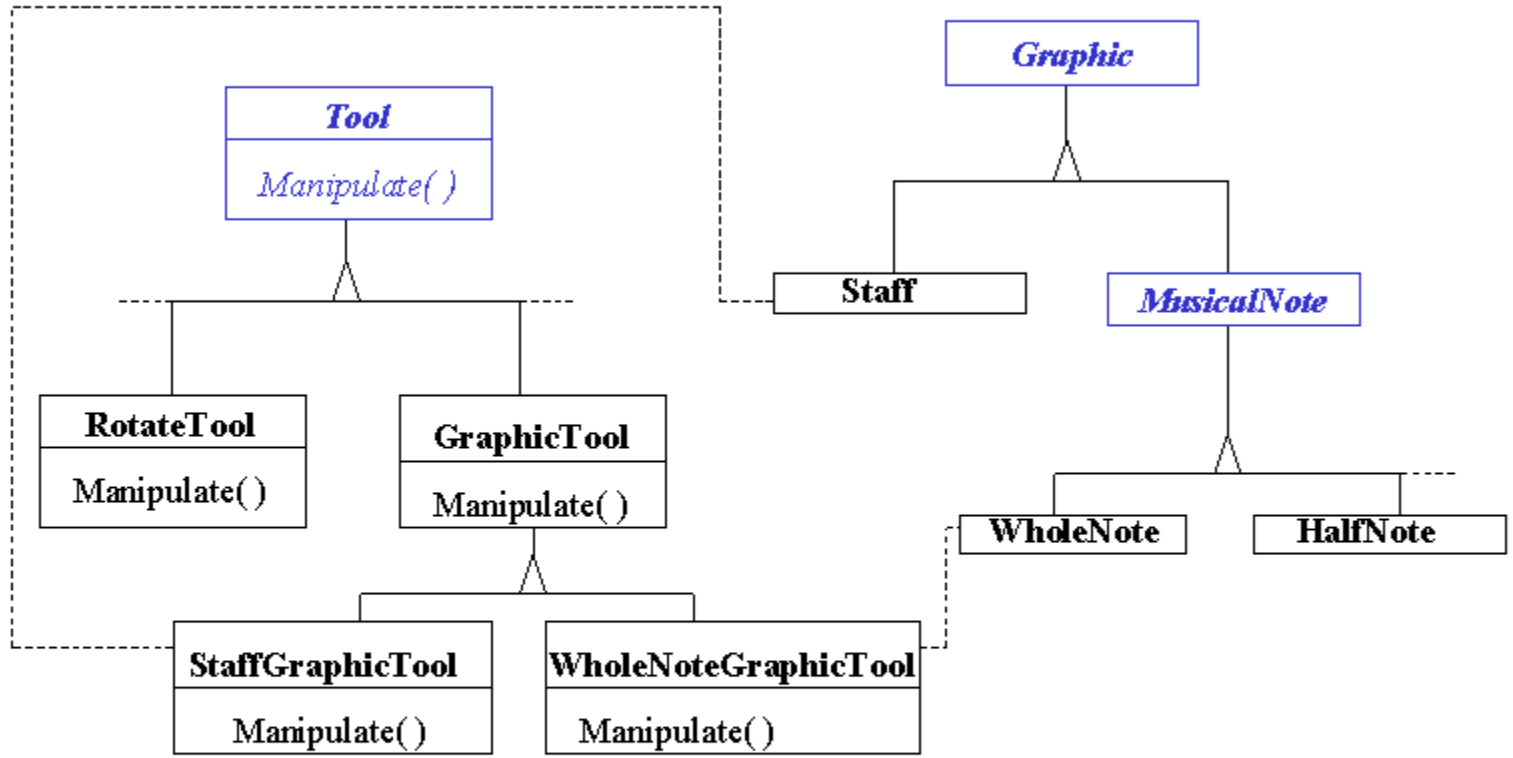


Prototype Pattern

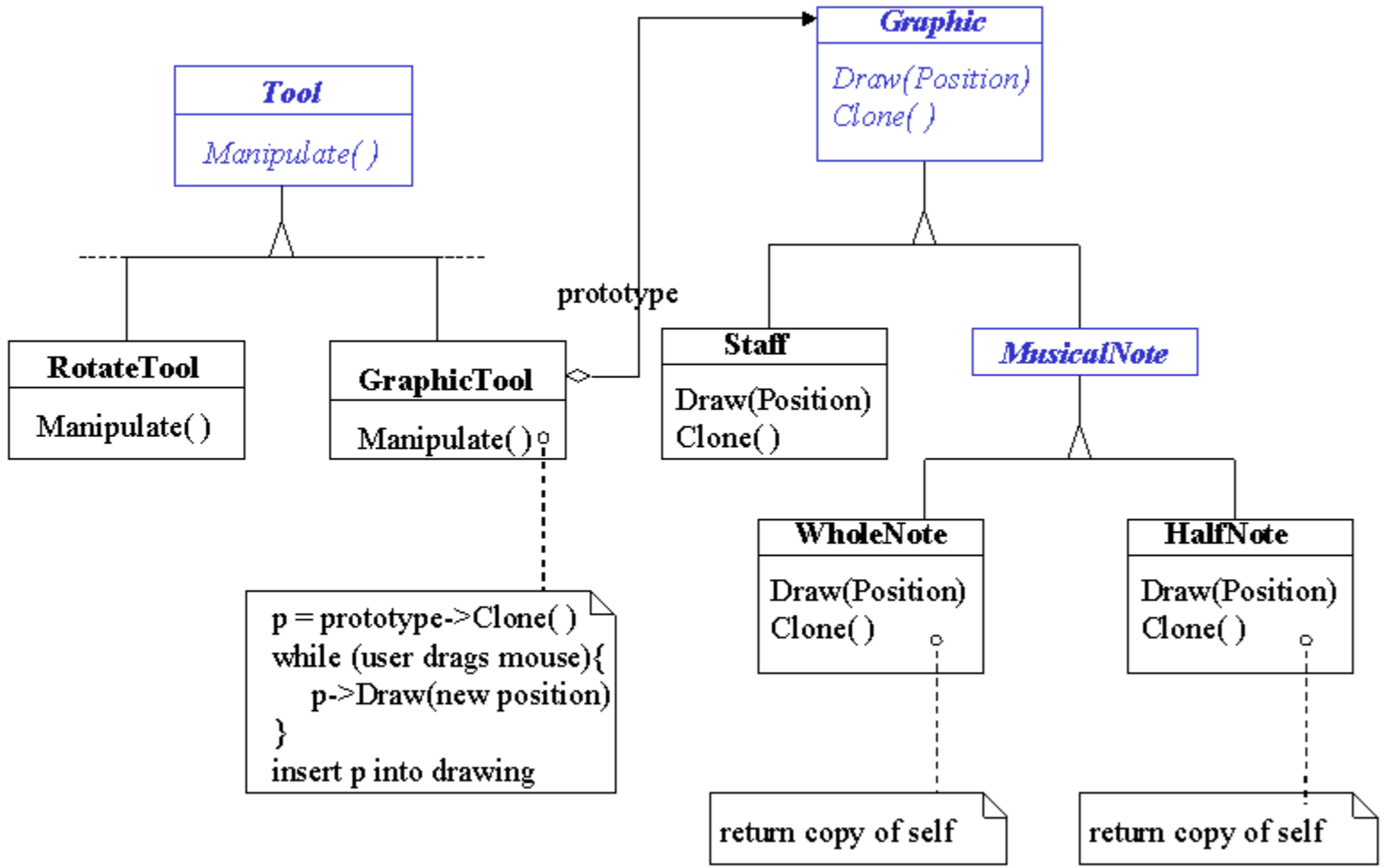
Prototype Pattern

- A creational pattern
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Problem


Tool
Manipulate( )
Graphic
Staff
MusicalNote
RotateTool
Manipulate( )
GraphicTool
Manipulate( )
WholeNote
HalfNote
StaffGraphicTool
Manipulate( )
WholeNoteGraphicTool
Manipulate( )

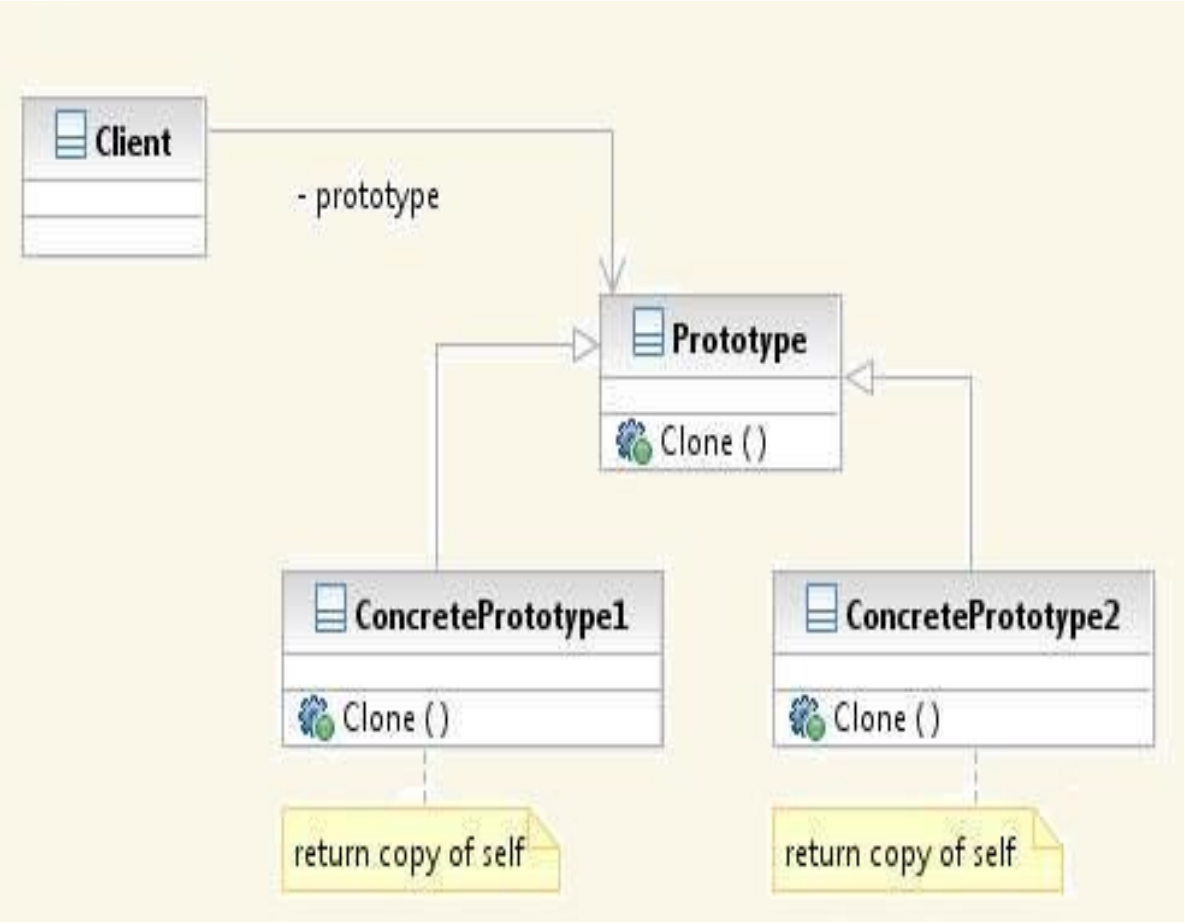
Prototype solution



Prototype Pattern UML

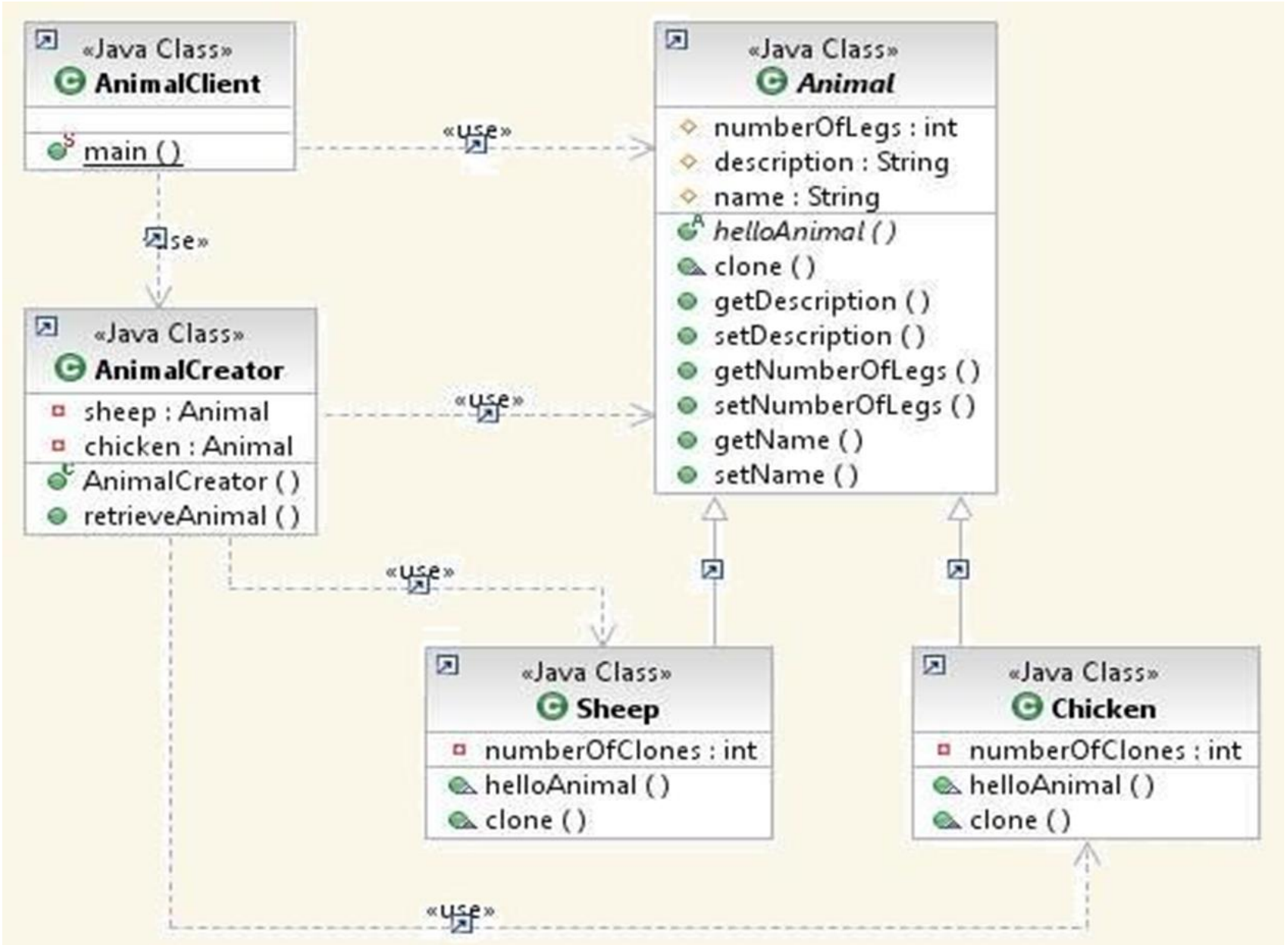
Participants:

- **Prototype**
  - declares an interface for cloning itself.
- **ConcretePrototype**
  - implements an operation for cloning itself.
- **Client**
  - creates a new object by asking a prototype to clone itself.



Example

Animal farm



Prototype Pattern Example code

```
public abstract class Animal implements Cloneable {
        protected int numberOfLegs = 0;
        protected String description = "";
        protected String name = "";

        public abstract String helloAnimal();

        public Animal clone() {
                Animal clonedAnimal = null;
                clonedAnimal = (Animal) super.clone();
                clonedAnimal.setName(name);
                return clonedAnimal;
        } // method clone

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
} // class Animal
```

```
public class Chicken extends Animal {
    private int numberOfClones = 0;

    public String helloAnimal() {
        StringBuffer chickenTalk = new StringBuffer();
        chickenTalk.append("Cluck cluck World. I am ");
        chickenTalk.append(name);
        return chickenTalk.toString();
     } // helloAnimal

     public Chicken clone() {
        Chicken clonedChicken = (Chicken) super.clone();
        String chickenName = clonedChicken.getName();
        numberOfClones++;
        clonedChicken.setName(chickenName + numberOfClones);
        return clonedChicken;
     } // method clone
}
```

```
public class Sheep extends Animal {
    private int numberOfClones = 0;

    public String helloAnimal() {
        StringBuffer sheepTalk = new StringBuffer();
        sheepTalk.append("Meeeeeee World. I am ");
        sheepTalk.append(name);
        return sheepTalk.toString();
    } // helloAnimal

    public Sheep clone() {
        Sheep clonedSheep = (Sheep) super.clone();
        String sheepName = clonedSheep.getName();
        numberOfClones++;
        clonedSheep.setName(sheepName + numberOfClones);
        return clonedSheep;
     } // method clone
}
```

Prototype Pattern Example code

```
public class AnimalCreator {
        private Animal sheep = new Sheep();
        private Animal chicken = new Chicken();

        public AnimalCreator() {
                sheep.setName("Sheep");
                chicken.setName("Chicken");
        } // no-arg constructor

        public Animal retrieveAnimal(String kindOfAnimal) {
                if ("Chicken".equals(kindOfAnimal)) {
                        return (Animal) chicken.clone();
                }
                else if ("Sheep".equals(kindOfAnimal)) {
                        return (Animal) sheep.clone();
                } // if
                return null;
        } // method retrieveAnimal
} // class AnimalCreator
```

Prototype Pattern Example code

```
public class AnimalClient {
    public static void main(String[] args) {
        AnimalCreator animalCreator = new AnimalCreator();
        Animal[] animalFarm = new Animal[8];

        animalFarm[0] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[1] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[2] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[3] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[4] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[5] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[6] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[7] = animalCreator.retrieveAnimal("Sheep");

        for (int i= 0; i<=7; i++) {
                System.out.println(animalFarm[i].helloAnimal());
        } // for
    } // main method
} // class AnimalClient
```

```
Cluck cluck world. I am Chicken1.
Cluck cluck world. I am Chicken2.
Cluck cluck world. I am Chicken3.
Cluck cluck world. I am Chicken4.
Meeeeeee world. I am Sheep1.
Meeeeeee world. I am Sheep2.
Meeeeeee world. I am Sheep3.
Meeeeeee world. I am Sheep4.
```

Prototype Pattern

When to Use

- When product creation should be decoupled from system behavior
- When to avoid subclasses of an object creator in the client application
- When creating an instance of a class is time-consuming or complex in some way.

Consequences of Prototype Pattern

- Hides the concrete product classes from the client
- Adding/removing of prototypes at run-time
- Allows specifying new objects by varying values or structure
- Reducing the need for sub-classing