# Design Patterns

## *Part 1*

Factory Method
Abstract Factory
Singleton
Adapter
Composite
State
Strategy
Template Method

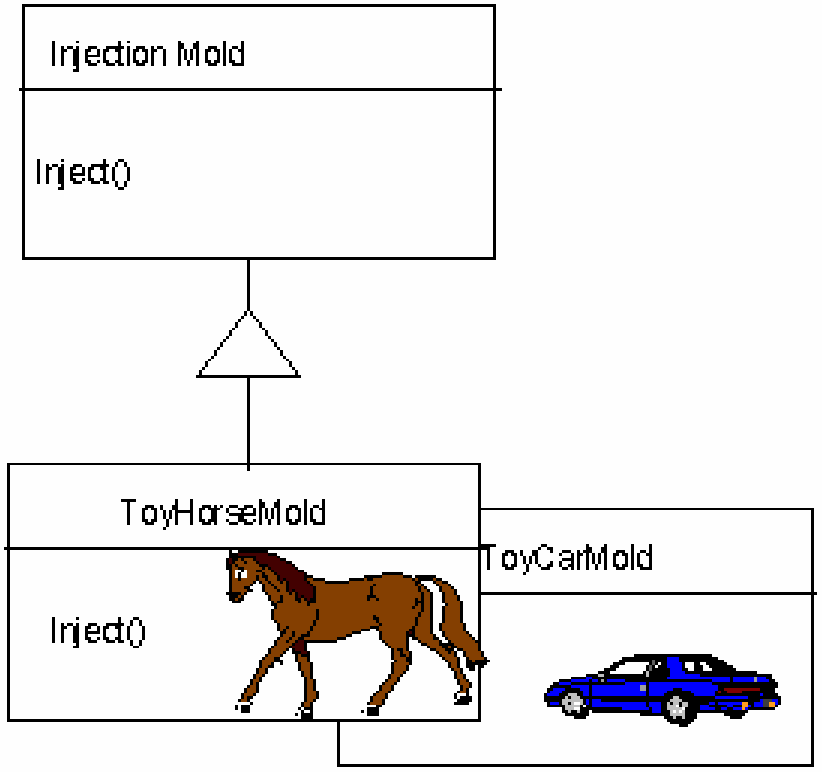
*version 3*

# Factory Method Pattern

# 1. Non-Software Example

## *Factory Method*

In injection molding, manufacturers process plastic molding powder and inject the plastic into molds of desired shapes. Like the *Factory Method*, the subclasses (in this case the molds) determine which classes to instantiate. In the example, the ToyHorseMold class is being instantiated.



# Non-Software Example (cont.)

- The Injection Mold corresponds to the *Product*, as it defines the interface of the objects created by the factory.
- A specific mold (ToyHorseMold or ToyCarMold) corresponds to the *Concrete Product*, as it implements the *Product* interface.
- The Toy Company corresponds to the *Creator*, since it may use the factory to create product objects.
- The Division of the Toy Company that manufactures a specific type of toy (horse or car) corresponds to the *Concrete Creator*.
- Creating objects with an Injection Mold is much more flexible than using equipment that only creates toy horses.

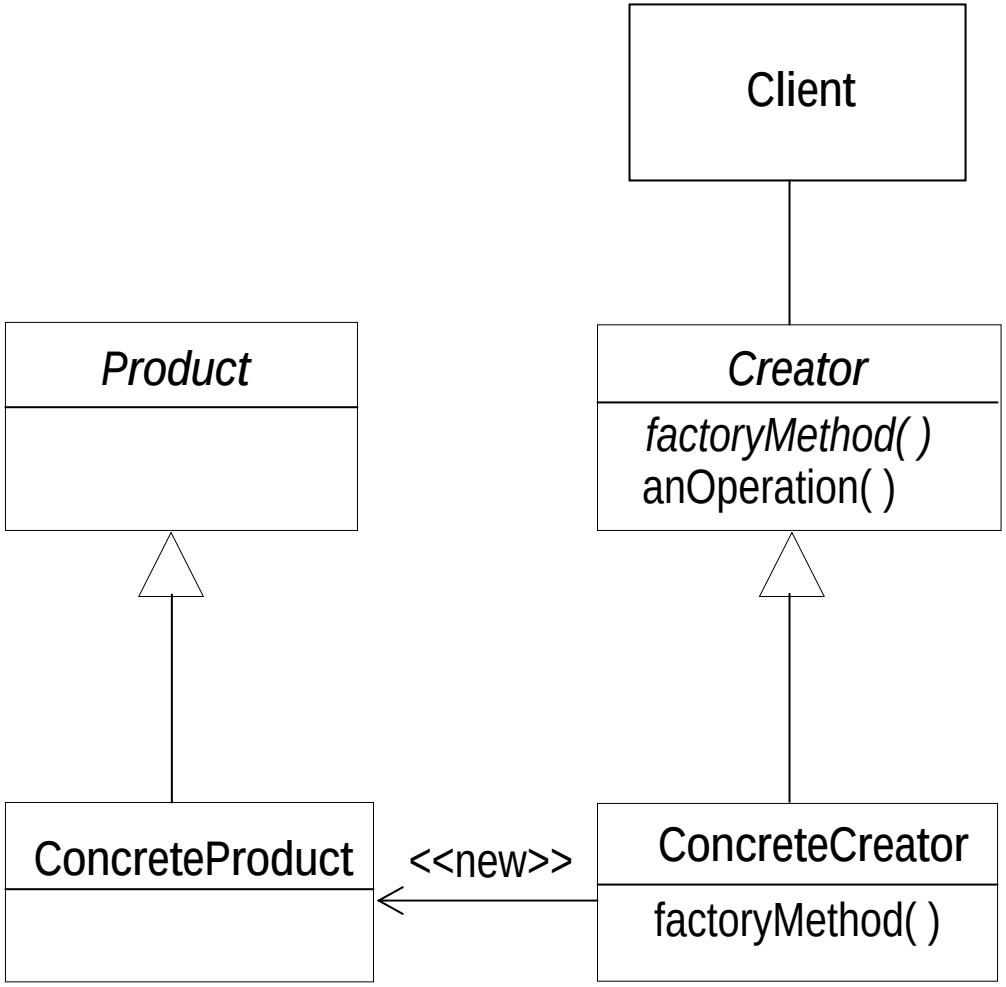
# 2. Problem

- A class needs to instantiate a derivation of another class, but doesn’t know which one.
- When a constructor alone is inadequate for creating an object, we need a method that returns the object.

# 3. Solution

- *"**Factory Method Pattern** defines an interface for creating an object, but let subclasses decide which class to instantiate. **Factory Method** lets a class defer instantiation to subclasses."* (GoF)

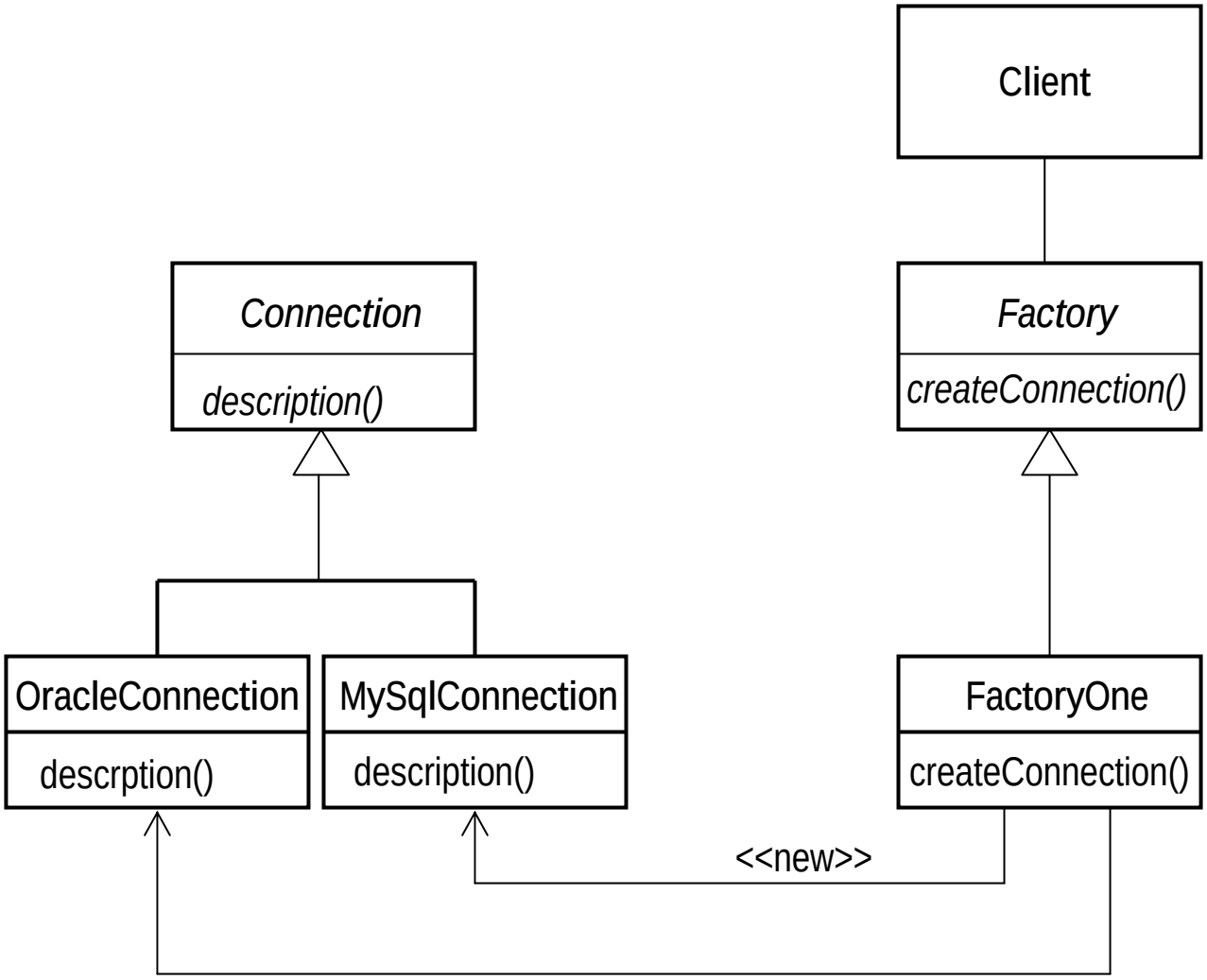
# 4. Class Diagram



# 5. Example

- During runtime, a client wants to create a database connection among various databases.
- New types of connections will be added in the future.

# Class Diagram



# Client

```
public class client {
    public static void main(String[] args) {
        Factory factory;
        factory = new FactoryOne();
        Connection connection = factory.createConnection(Oracle); //Factory method
        System.out.println("You are connecting with " +  connecton.description());
    }
}
```

# Factory and FactoryOne

```
public abstract class Factory {
    public Factory() {
        protected abstract connection createConnection(String type); //Factory method
    }
}

public class FactoryOne extends Factory {
    publice connection createConnection(String type) { //Factory method
        if (type.equals(“Oracle”))
            return new OracleConnection();
        else
            return new MySqlConnection();
    }
}
```

# Connection, OracleConnection, and MySqlConnection

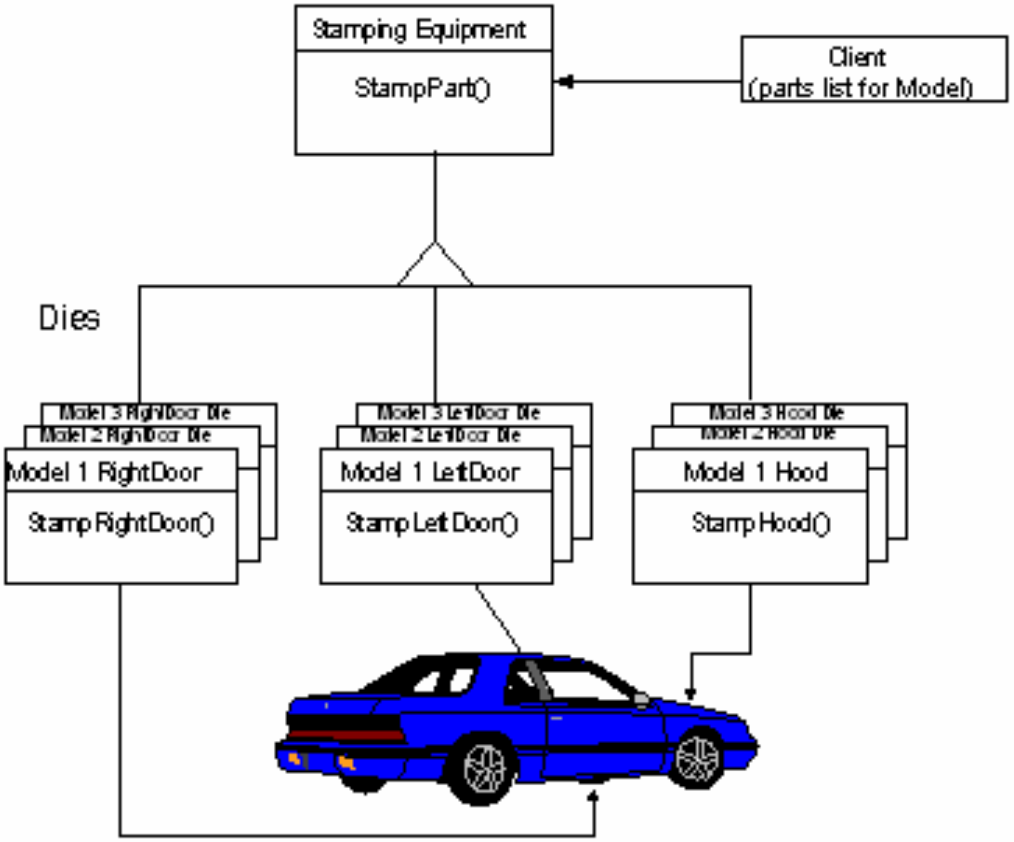
```
public abstract class Connection {
    public abstract Connection() {
    }

    public String description() {
        return “Generic”;
    }
)
```

```
public class OracleConnection extends Connection {
    public OrcleConnection() {
    }

    public String description {
        return “Oracle”;
    }
}

public class MySqlConnection extends Connection {
    public MySqlConnection() {
    }

    public String description {
        return “MySQL”;
    }
}
```

# Abstract Factory Pattern

# 1. Non-Software Example

## *Abstract Factory*




Sheet metal stamping equipment is an example of an *Abstract Factory* for creating auto body parts. Using rollers to change the dies, the concrete class can be changed. The possible concrete classes are hoods, trunks, roofs, left and right front fenders, etc. The master parts list ensures that classes will be compatible. Note that an *Abstract Factory* is a collection of *Factory Methods*.

Creational

# Non-Software Example (cont.)

- The Master Parts List corresponds to the *Client*, which groups the parts into a family of parts.
- The Stamping Equipment corresponds to the *Abstract Factory*, as it is an interface for operations that create *Abstract Product* objects.
- The dies correspond to the *Concrete Factories*, as they create *Concrete Products*.
- Each part category (Hood, Door, etc.) corresponds to the *Abstract Product*.
- Specific parts (i.e., driver side door for 1998 Nihonsei Sedan) corresponds to the *Concrete Products*.
- Changing dies to make new product families (Hoods to Doors) is easy.

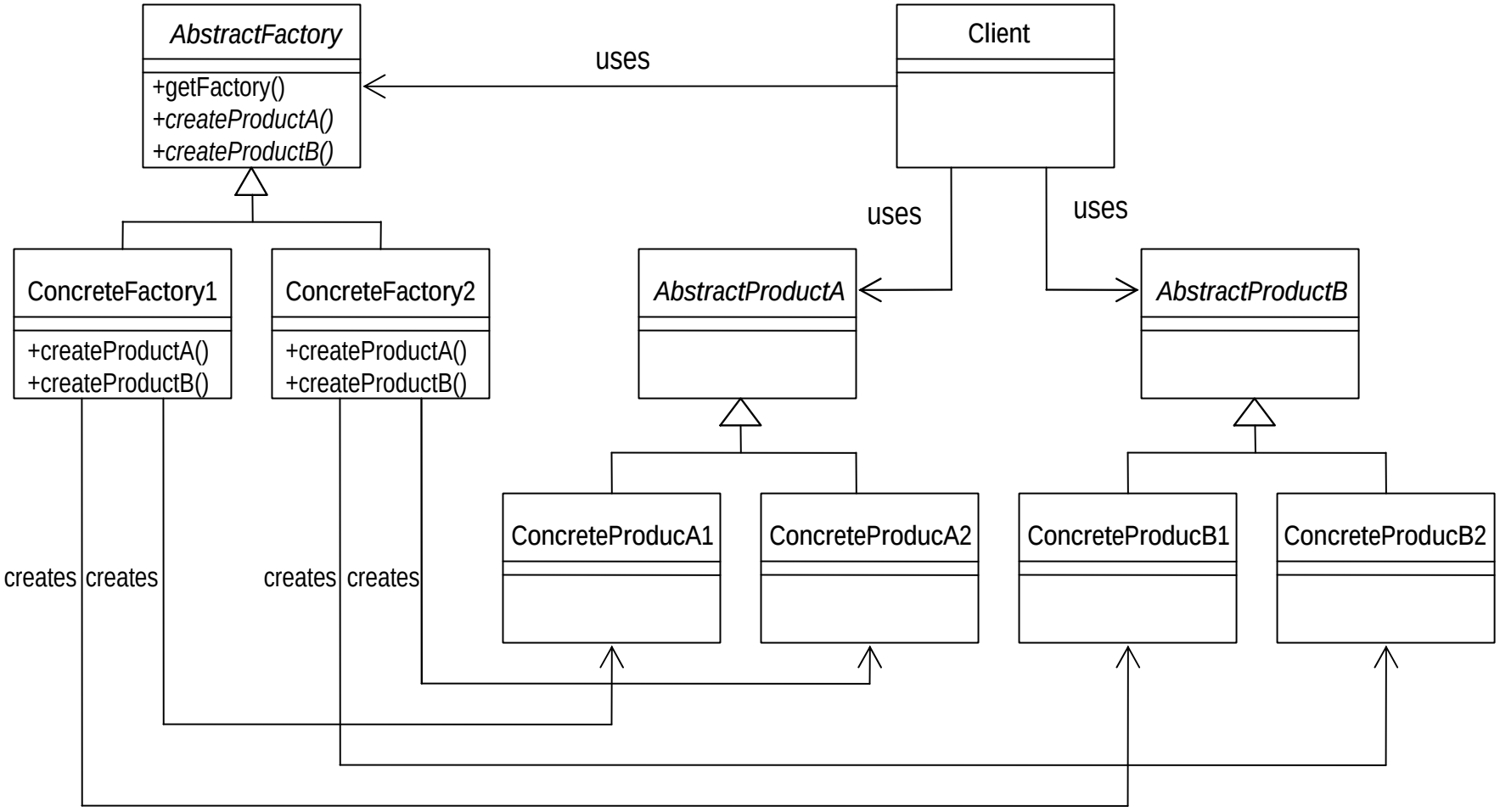
# 2. Problem

- We want to build a system that works with multiple ***families*** of products.

# 3. Solution

- *"**Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes."* (GoF)
- For a given *set* of related *abstract classes*, ***Abstract Factory*** provides a way to create instances of these abstract classes from a *matched set* of *concrete subclasses*.

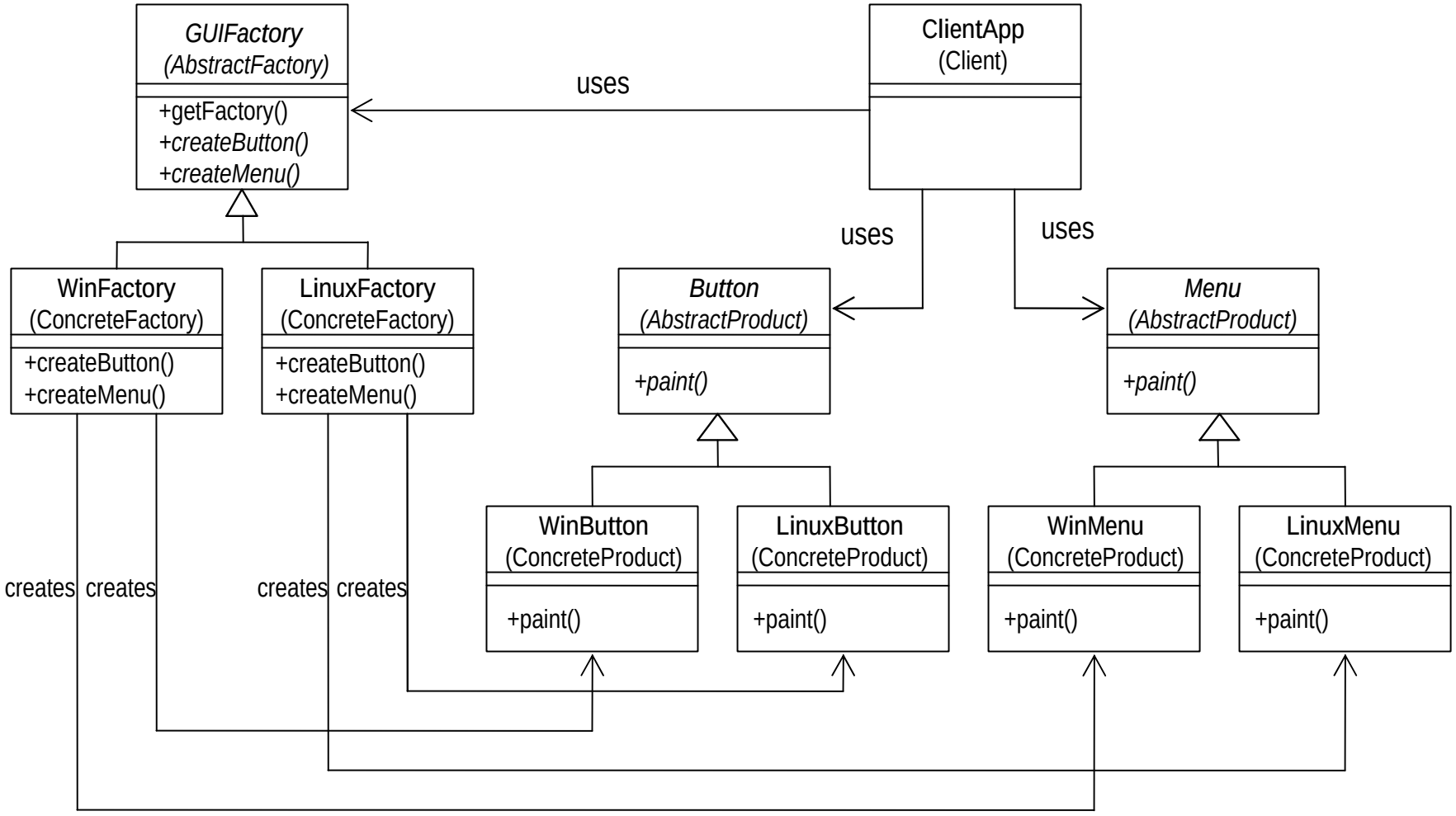
# 4. Class Diagram



# 5. Example

- *Client* application can be run on multiple operating systems.
- *Client* application uses a *GUIFactory* to create different widgets for use.
- The widgets that are created are based on the operating system that the application is running on.
- *Client* application has no knowledge of the actual widget type used (i.e., *LinuxButton* or *WindowsButton*).
- *Client* application uses abstract classes as the interface to the created widgets (i.e., uses through interface of an abstract *Button*).

# Class Diagram



# ClientApp

```
class ClientApp {
    public static void main(String[] args) {
        GUIFactory aFactory = GUIFactory.getFactory();
        Button aButton = aFactory.createButton();
        aButton.caption = "Play";
        aButton.paint();
    }
}
```

# GUI Factory

```
abstract class GUIFactory {
   public static GUIFactory getFactory() {
      int sys = readFromConfigFile("OS_TYPE");
      if (sys == 0)
         return (new WinFactory());
      else
         return (new LinuxFactory());
   }

   public abstract Button createButton();
   public abstract Menu createMenu();
}
```

# WinFactory and LinuxFactory

```
class WinFactory extends GUIFactory {
    public Button createButton() {
        return(new WinButton());
    }
    public Menu createMenu() {
        return(new WinMenu());
    }
}

class LinuxFactory entends GUIFactory {
    public Button createButton() {
        return(new LinuxButton());
    }
    public Menu createMenu() {
        return(new LinuxMenu());
    }
}
```

# Button and Menu

```
abstract class Button {
   public String caption;
   public abstract void paint();
}

abstract class Menu {
   public String caption;
   public abstract void paint();
}
```

# WinButton, LinuxButton, WinMenu, and LinuxMenu

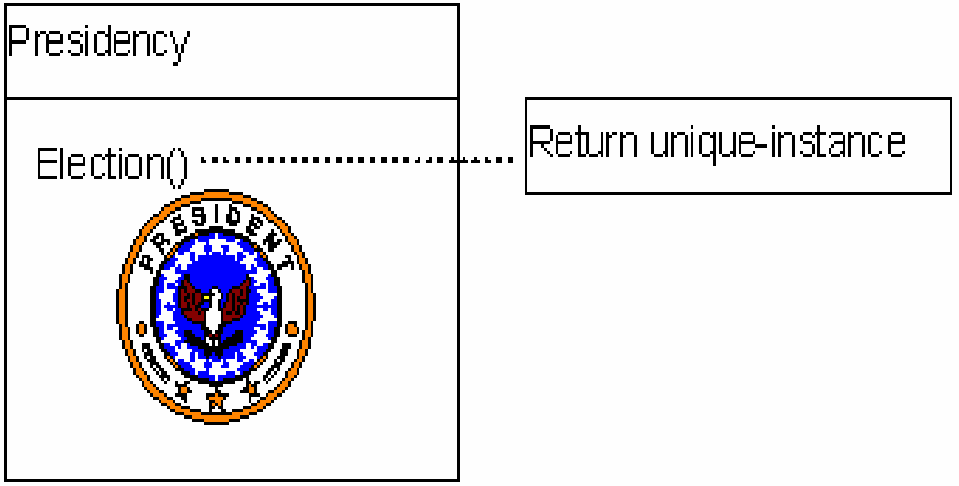
```
class WinButton extends Button {
    public void paint() {
        System.out.println("I'm a WinButton: “ + caption);
    }
}
class LinuxButton extends Button {
    public void paint() {
        System.out.println("I'm a LinuxButton: “ + caption);
    }
}
class WinMenu extends Menu {
    public void paint() {
        System.out.println("I'm a WinMenu: “ + caption);
    }
}
class LinuxMenu extends Menu {
    public void paint() {
        System.out.println("I'm a LinuxMenu: “ + caption);
    }
}
```

# Singleton Pattern

# 1. Non-Software Example

## *Singleton*

The office of the Presidency of the United States is an example of a *Singleton*, since there can be at most one active president at any given time. Regardless of who holds the office, the title "The President of the United States" is a global point of reference to the individual.



# Non-Software Example (cont.)

- The Office of the Presidency of the United States corresponds to the *Singleton*.
- The office has an instance operator (the title of "President of the United States") which provides access to the person in the office.
- At any time, at most one unique instance of the president exists.
- The title of the office provides controlled access to a sole instance of the president.
- Since the office of the presidency encapsulates the president, there is strict control over how and when the president can be accessed.

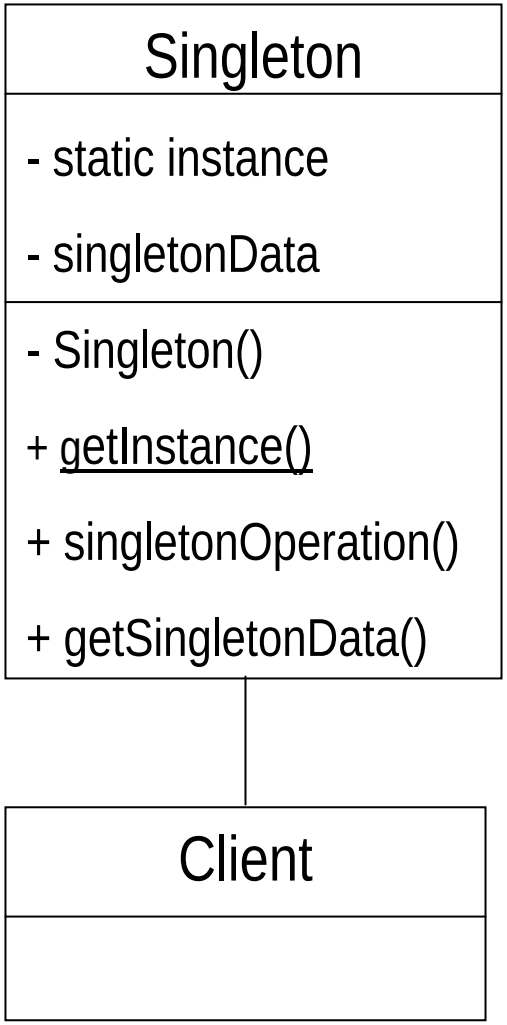
# 2. Problem

- We want to have only *one* of an object but there is no global object that controls the instantiation of this object.
- Several different client objects need to refer to the same thing and we want to ensure that we do not have more than one of them.
- Objects need a *global* and *single* points of access.

# 3. Solution

- *"**Singleton Pattern** ensures a class only has one instance, and provides a global point of access to it."* (GoF)
- Define a *static method* of the class that returns the *Singleton*.

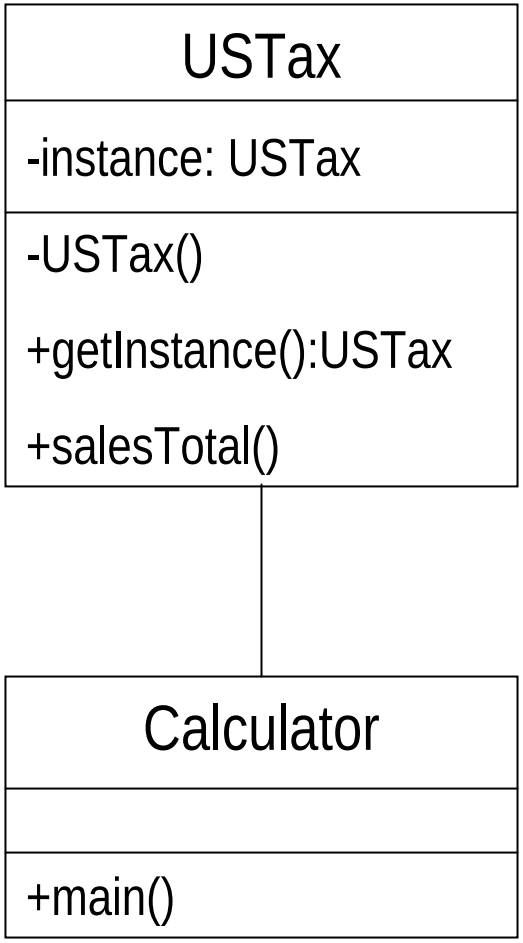
# 4. Class Diagram



# 5. Example

- *USTax* is the ***Singleton*** and there can be only one instance of  tax on any given transaction at one point.
- *Calculator* is the ***Client*** that uses the instance of *USTax* to get the total of the sale.
- *USTax* has an instance operator which provides access to the applicable tax on any particular sales transaction.
- *USTax* is encapsulated, hence there is a strict control over its access.

# Class Diagram



# Calculator and USTax

```
class calculator {
    public static void main(String args[ ]) {
       UsTax tax = USTax.getInstance();
       tax.salesTotal();

       }
   }
}
```

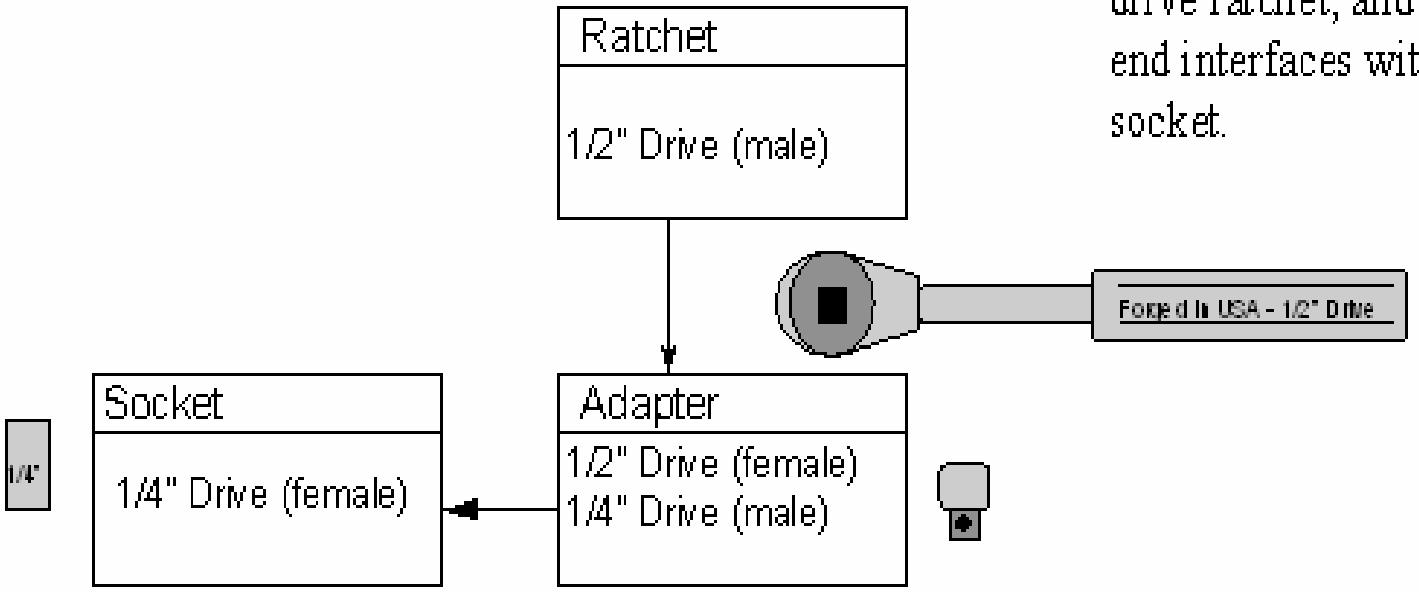
```
class USTax {
    private static USTax instance;
    private USTax();

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

    public float salesTotal() {
        ...
    }
}
```

# Adapter Pattern

# 1. Non-Software Example

## *Adapter*

A 1/2" drive ratchet will not ordinarily work with a 1/4" drive socket. Using an *Adapter*, the female end interfaces with the 1/2" drive ratchet, and the male end interfaces with the 1/4" socket.



# Non-Software Example (cont.)

- The ratchet corresponds to the *Target*, as it is the domain specific interface that the client uses.
- The socket corresponds to the *Adaptee*, since it contains an interface (1/4" drive) that needs adapting.
- The socket adapter corresponds to the *Adapter*, as it adapts the interface of the *Adaptee* to that of the *Target*.
- The *Adaptee* is adapted to the *Target* by committing to a *Concrete Adapter* object.

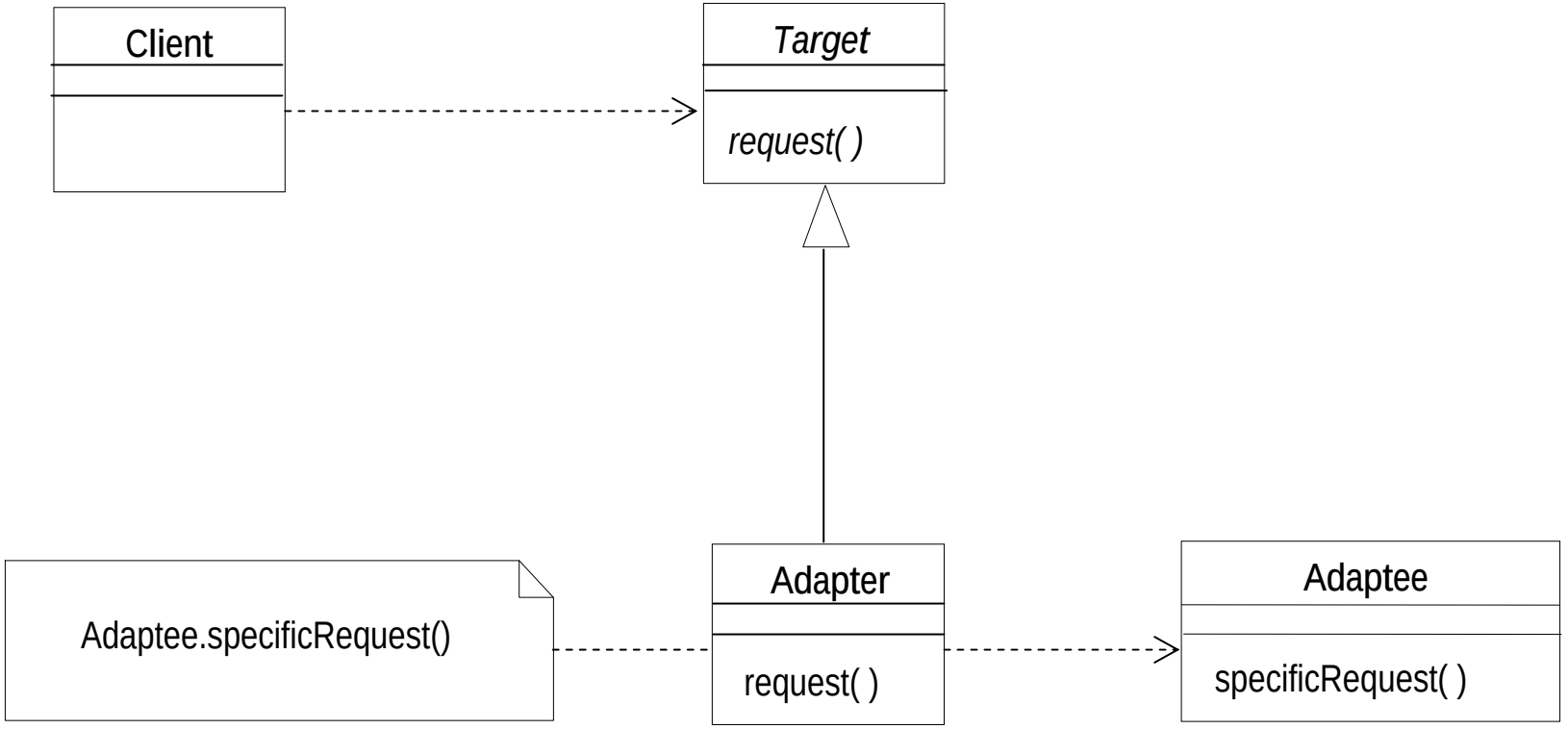
# 2. Problem

- A system has the right data and behavior but the wrong interface.
- We have to make something a derivative of an abstract class we are defining or already have.
- We need to make use of incompatible classes
- We need to reuse existing legacy components.

# 3. Solution

- *"**Adapter Pattern** converts the interface of a class into another interface clients expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."* (GoF)
- The ***Adapter*** provides a *wrapper* with the desired *interface*.

# 4. Class Diagram

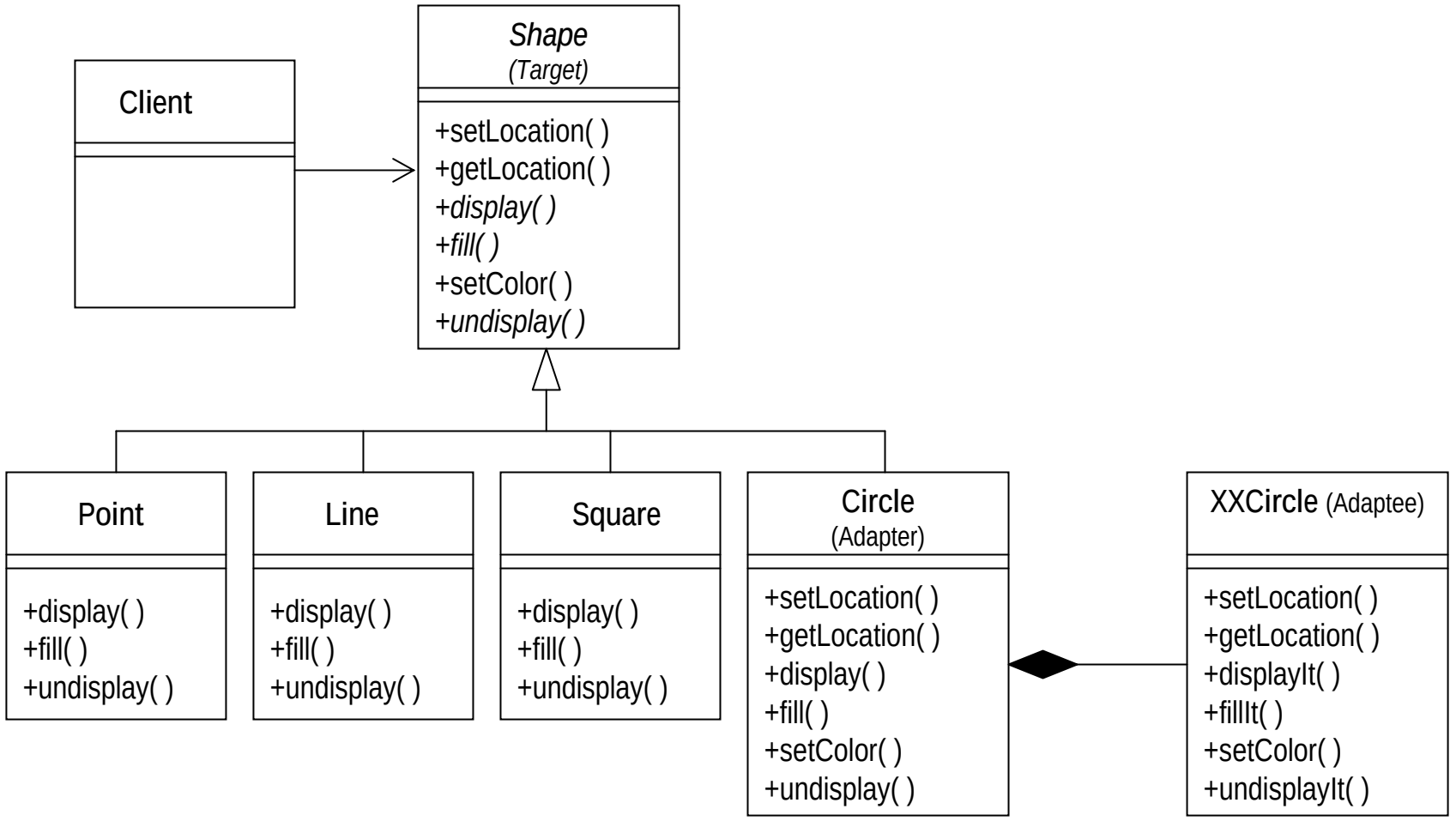


# 5. Example

*Client* objects want to display, fill, and undisplay different *shapes* with two sets of requirements

- Original requirements
  - Create classes for points, lines, and squares that have various behavior – display, fill, and undisplay.
  - The client objects should not have to know whether they actually have a point, a line, or a square. They just want to know that they have one of these shapes.
- New requirements
  - Create classes for circles, a new kind of shape.
  - Adapted an existing class, *XXCircle,* which has different method names and parameter lists.

# Class Diagram



# Circle

```
class Circle extends Shape {
    private XXCircle pxc;
    …
    public Circle( ) {
        pxc = new XXCircle( );
    }

    void public display( ) {
        pxc.displayIt( );
    }
}
```

# Composite Pattern

# 1. Non-Software Example

*Composite* is often exhibited in recipes. A recipe consists of a list of ingredients which may be atomic elements (such as milk and parsley) or composite elements (such as a roux).

White Roux

1 tablespoon flour
1 tablespoon butter

To make a roux, melt the butter in a saucepan over medium heat. When the butter starts to troth, stir in flour, combining well.

Continue cooking the roux over heat until it turns pale brown and has a nutty fragrance.

Parsley Sauce

1 portion of white roux
1 cup milk
2 tablespoon of parsley

To make sauce, over heat, add a little of the milk to the roux, stirring until you have a smooth liquid. Stir over medium heat for 2 minutes. Then stir in parsley and cook for another 30 seconds. Remove from heat, and leave sauce standing to thicken.

# Non-Software Example (cont.)

- Any item in a recipe is a *Component*.
- There are two types of components: *Leaves* and C*omposites*.
- The simple elements, such as milk, correspond to the *Leaf* objects.
- Elements, such as the white roux which are themselves composed of *Leaf* elements, are *Composites*.
- The chef corresponds to the C*lient*.
- Recipes can be written by combining primitive and composite objects.
- When a chef combines elements of a recipe, *Composite* elements are added the same way that simple elements are.
- It is easy to add new kinds of elements, as is evidenced by the frequency in which recipes are combined.

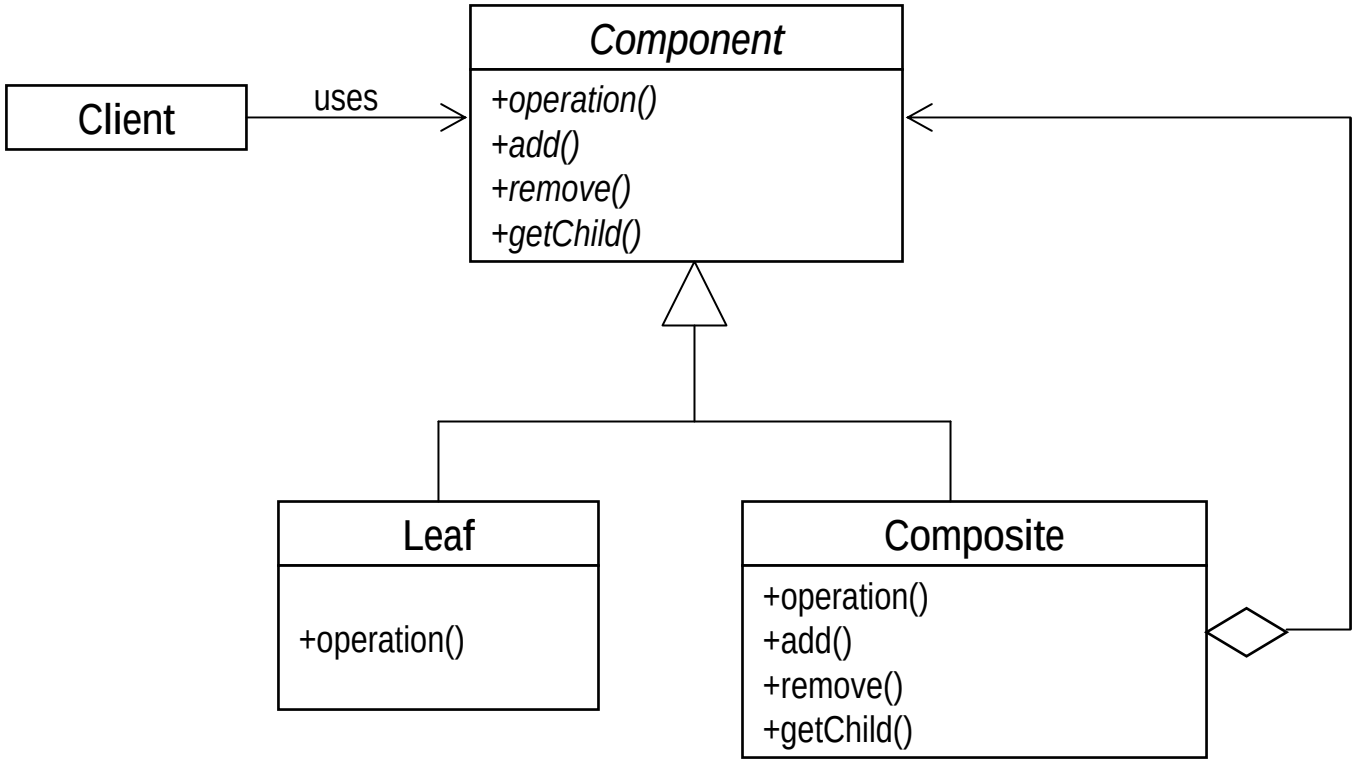
# 2. Problems

- A hierarchy of variable width and depth with *leaves* and *composites* must be treated uniformly through a common interface.
- Users should be able to ignore the difference between compositions of objects and individual objects and treat them identically.
- Having to query the "type" of each object before attempting to process it is not desirable.

# 3. Solution

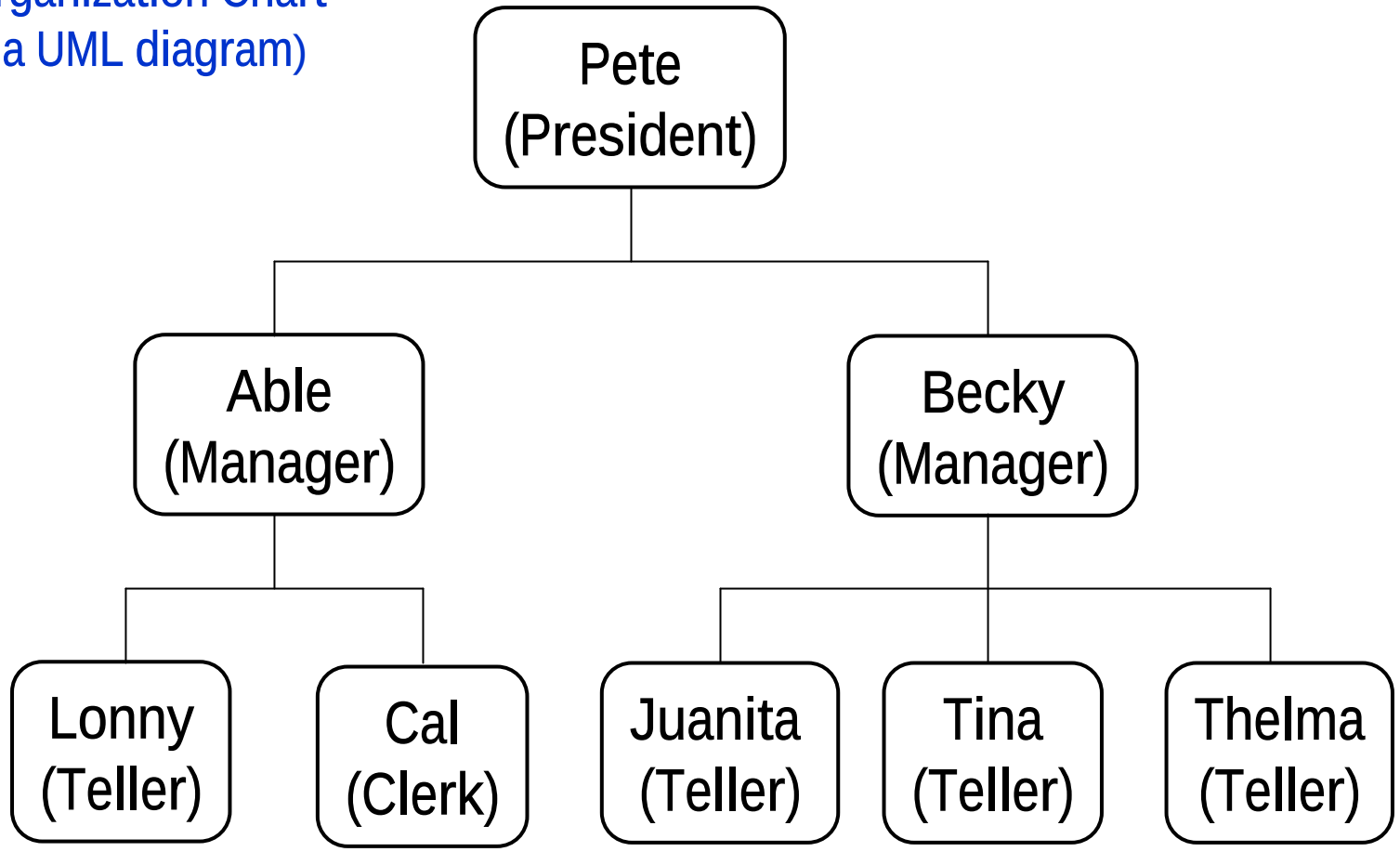
- *"**Composite Pattern** composes objects into tree structures to represent part-whole hierarchies. **Composite** lets clients treat individual objects and compositions of objects uniformly."* (GoF)
- An abstract class that represents *both* primitives and their containers.
- The *Component* interface specifies the services that are shared among *Leaf* and *Composite classes*.
- *Composite* uses recursive composition.

# 4. Class Diagram



# 5. Example

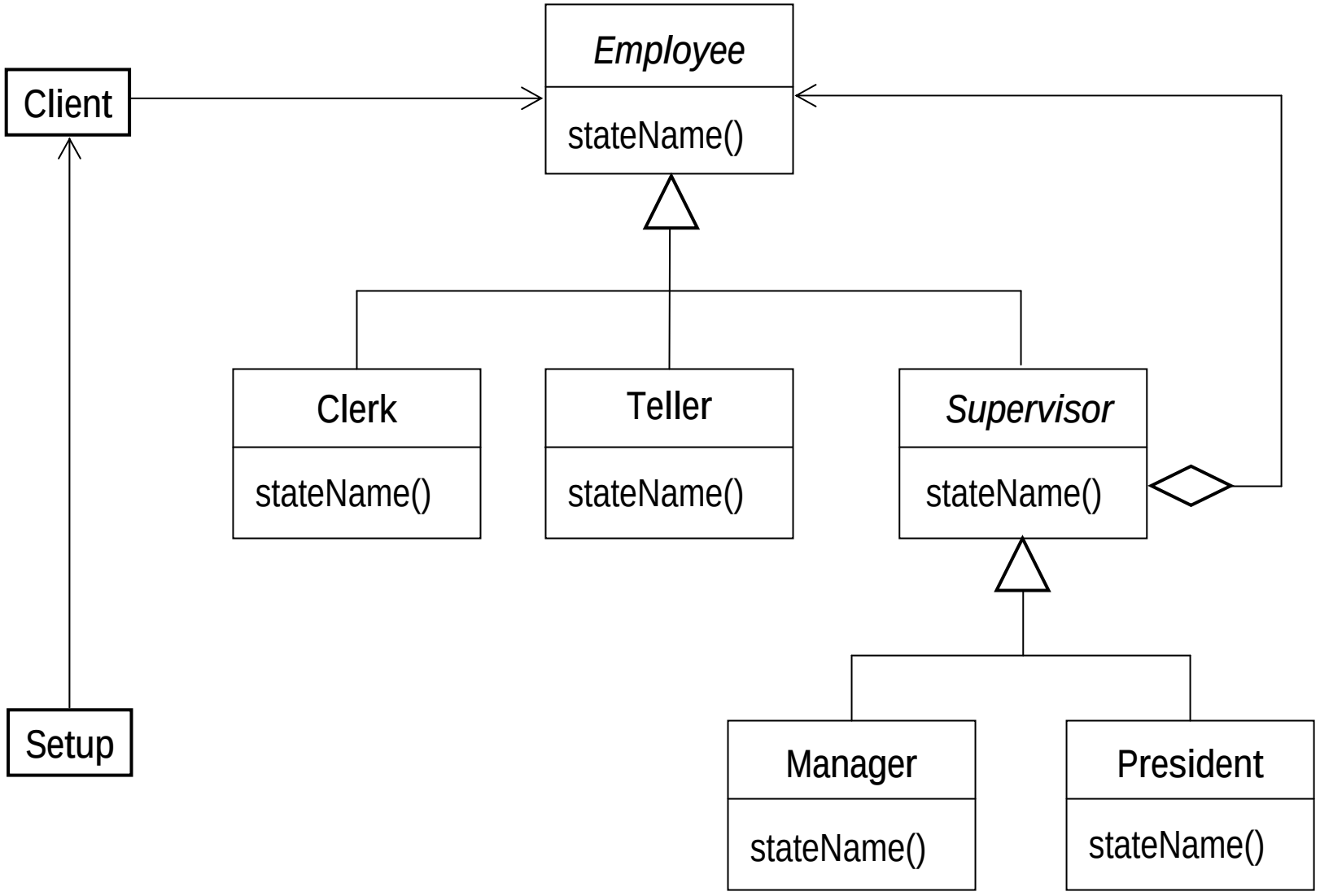
An Organization Chart
(not a UML diagram)



# Example (cont.)

- Organization charts are in the form of trees.
- We want to be able to write the client code once, but run it with varying configurations of the organization chart.
- We want to add and remove employees at runtime and execute operations on all of them.

# Class Diagram



# Setup

```
// Sets up the org chart and initiates execution
class Setup {

  public static void main( String args[] )  {

  // Make manager Able's organization
    Teller lonny = new Teller( "Lonny" );
    Clerk cal = new Clerk( "Cal" );
    Manager able = new Manager( "Able" );
    able.add( lonny);
    able.add( cal);

  // Make manager Becky's organization
  Teller juanita = new Teller( "Juanita" );
  Teller tina = new Teller( "Tina" );
  Teller thelma = new Teller( "Thelma" );
  Manager becky = new Manager( "Becky" );
  becky.add( juanita);
  becky.add( tina );
  becky.add( thelma);

  // Create the president's direct reports
  President pete = President.getPresident( "Pete" );
  pete.add( able );
  pete.add( becky );

  // Initiate client
  Client.employee = pete;
  Client.doClientTasks();
   }
}
```

# Client and Employee

```
class Client {
    // This class relates to a specific Employee
    public static Employee employee;
    …
    public static void doClientTasks() {
    // Do work with this employee
     ...
    employee.stateName();
    }
}
```

```
abstract class Employee {
    String name = "not assigned yet";
    String title = "not assigned yet";

    public void stateName() {
        System.out.println( title + " " + name );
    }
}
```

# Supervisor

```
abstract class Supervisor extends Employee {
    protected Vector directReports = new Vector();

    public void stateName() {
        super.stateName();    // print name of this employee first
        if( directReports.size() > 0 )   // be sure there are elements
            for( int i = 0; i < directReports.size(); ++i )
             ( (Employee)directReports.elementAt( i ) ).stateName();
    }

    public void add( Employee anEmployee ) {
        this.directReports.addElement( anEmployee );
    }
}
```

# Manager

```
class Manager extends Supervisor  {
    public Manager( String aName ) {
        this();
        name = aName;
    }

    public Manager() {
        super();
        title = "Manager";
    }

    public void stateName() {
        // do processing special to manager naming
        ...
        super.stateName();
    }
}
```

# President

```
class President extends Supervisor {
    private static President president = new President();
    private President( String aName ) {
        this();
        name = aName;
    }

     private President( ) {
        super();
        title = "President";
    }

    public void stateName()  {
        // Do processing special to presidential naming
        ...
        super.stateName();
    }

    public static President getPresident( String aName ) {
        president.name = aName;
        return President.president;
    }
}
```

# Clerk and Teller

```
class Clerk extends Employee {
    public Clerk( String aName ) {
        this();
        name = aName;
    }

    public void stateName() {
        super.stateName();
    }

    public Clerk() {
        title = "Clerk";
    }
}
```

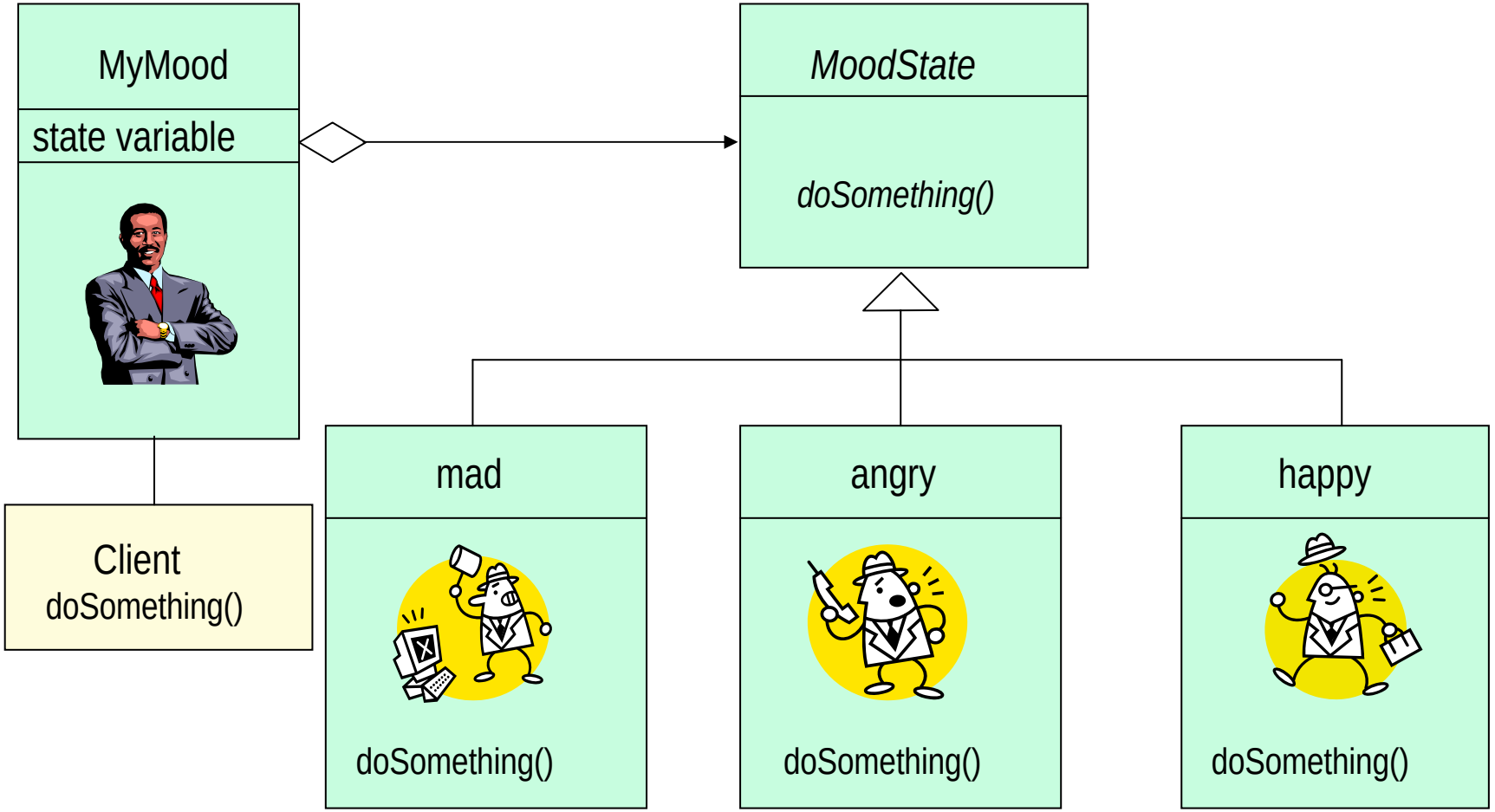
```
class Teller extends Employee {
    public Teller( String aName ) {
        this();
        name = aName;
    }

    public void stateName() {
        super.stateName();
    }

    public Teller() {
        title = "Teller";
    }
}
```

# Output

```
President Pete
Manager Able
Teller Lonny
Clerk Cal
Manager Becky
Teller Juanita
Teller Tina
Teller Thelma
```

# State Pattern

# 1. Non-Software Example



# Non-Software Example (cont.)

- The object of type MyMood can be in one of the several ***states***: mad, angry, and happy which is indicated by the "current state" kept in MyMood.
- It has different behaviour in each ***state***.
- MyMood contains a pointer to the current ***state*** and uses the behaviour of the corresponding object of type MoodState.

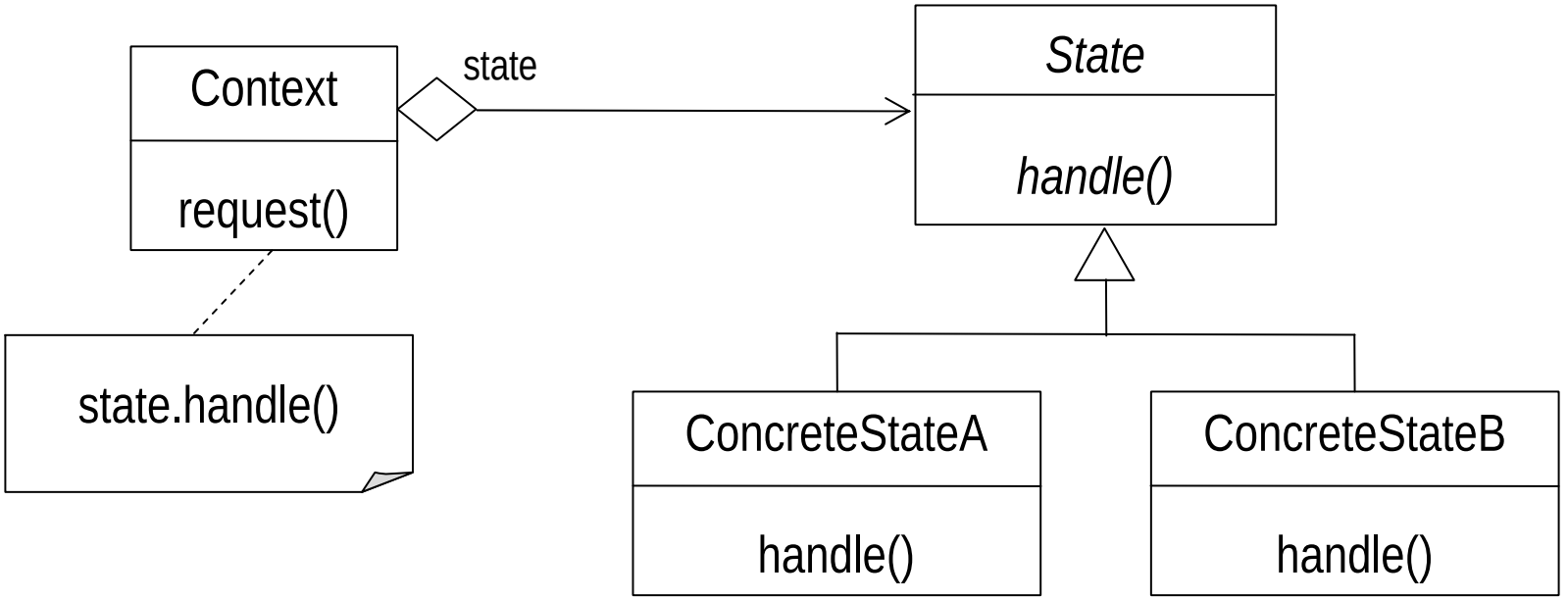
# 2. Problem

- An object's behavior depends on its *State*.
- An object must change its behavior at run-time depending on the *State*.
- Operations have large, multipart conditional statements that depend on the object's *State*.

# 3. Solution

- *"State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class."* (GoF)
- Define a Context class to present a single interface to the outside world.
- Aggregate a State object and delegate behavior to it.
- Represent 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.

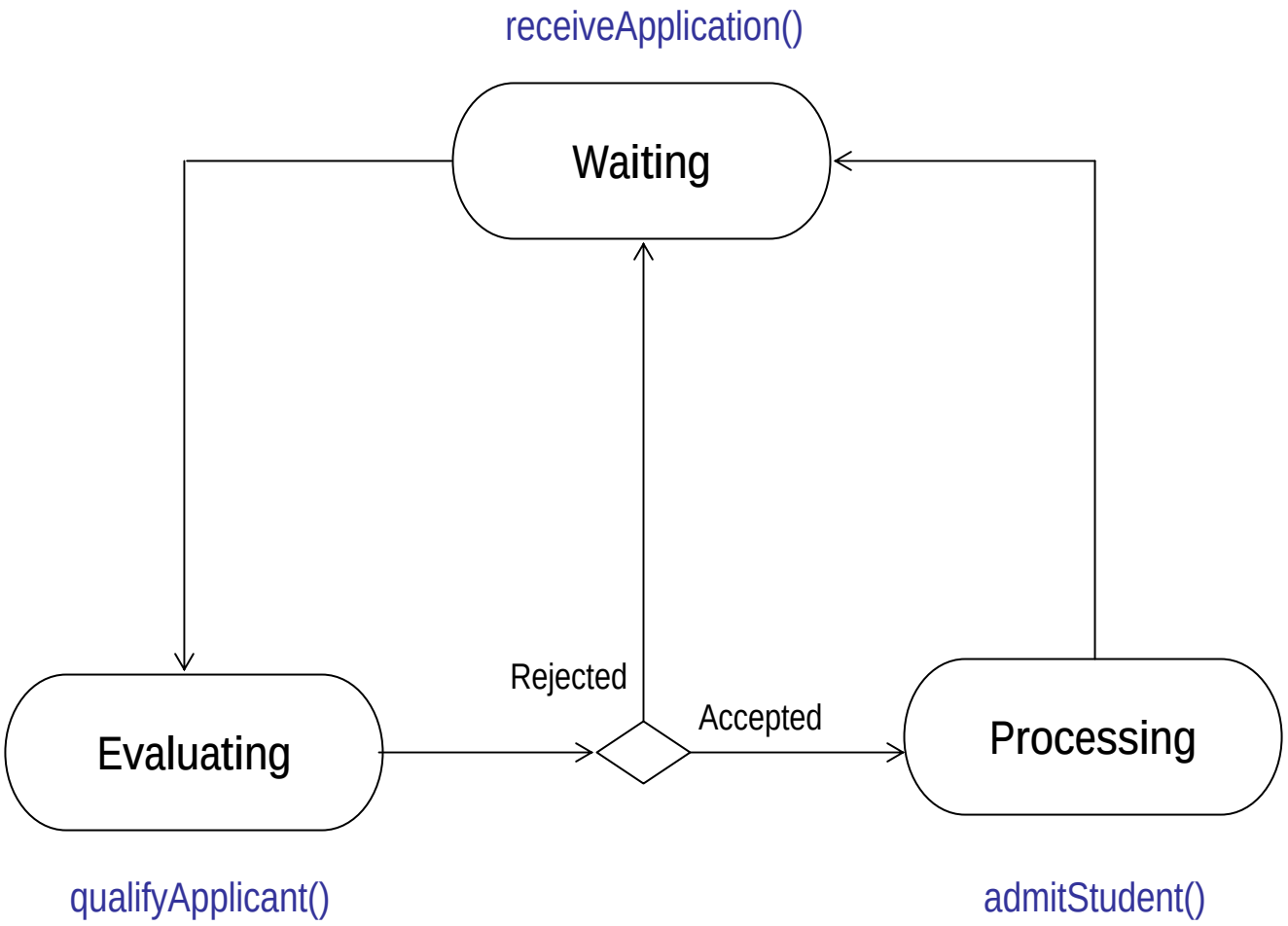
# 4. Class Diagram



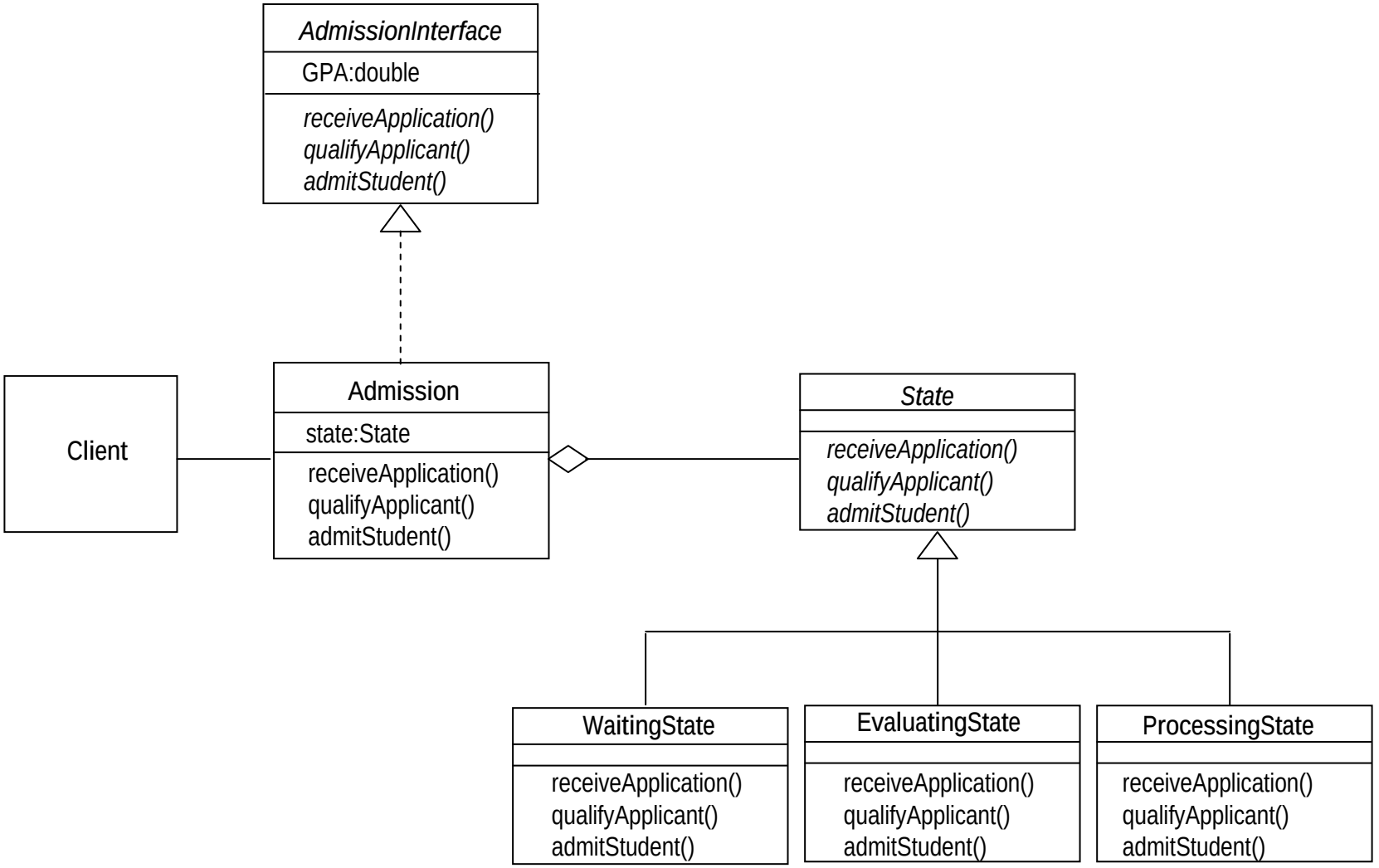
# 5. Example

- A university admission sytem has three methods, receiveApplication(), qualifyApplicant(), and admitStudent(),
- This system is in one of three states – waiting, evaluating, and processing.
- The behavior of each method changes depending on the state of the object.

# State Diagram



# Class Diagram



# Client

```
public class TestAdmission {
    Admission admission;
    public static void main(String[] args) {
        admission = new Admission();
        admission.receiveApplication();
        admission.qualifyApplicant(3.5);
        admission.admitStudent();
    }
}
```

# AdmissionInterface

```
public interface AdmissionInterface {
    public static final double GPA = 3.0;

    public void receiveApplication();
    public void qualifyApplicant(double g);
    public void admitStudent();

    pubic State getWaitingState() { }
    pubic State getEvaluatingState() { }
    pubic State getProcessingState() { }

    public void setState(State s);
}
```

# Admission

```
public class Admission implements AdmissionInterface {
   State waitingState;
   State evaluatingState;
   State processingState;
   State state;

   public Admission() {
      waitingState = new waitingState(this);
      evaluatingState = new evaluatingState(this);
      processingState = new processingState(this);
      state = waitingState;
   }

   public void receiveApplication() {
      System.out.println(state.receiveApplication())
   }

   public void qualifyApplicant(double g) {
      System.out.println(state.qualifyApplicant(g))
   }

   public void admitStudent() {
      System.out.println(state.admitStudent())
   }
```

```
   pubic State getWaitingState() {
      return waitingState;
   }

   pubic State getEvaluatingState() {
      return evaluatingState;
   }

   pubic State getProcessingState() {
      return processingState;
   }

   pubic void setState(State s) {
      state = s;
   }

}
```

# State

```
public interface State {
    public String receiveApplication();
    public String qualifyApplicant(double g);
    public String admitStudent();
}
```

# WaitingState

```
public class WaitingState implements State {
    AdmissionInterface admission;
    public WaitingState(AdmissionInterface a) {
        admission = a;
    }
    public String receiveApplication() {
        return “Received an application.”;
        admission.setState(admission.getEvaluatingState());
    }
    public String qualifyApplicant(double g) {
        return “Must receive an application first.”;
    }
    public String admitStudent() {
        return “Must receive an application first.”;
    }
}
```

# EvaluatingState

```
public class EvaluatingState implements State {
    AdmissionInterface admission;
    public EvaluatingState(AdmissionInterface a) {
        admission = a;
    }
    public String receiveApplication() {
        return “Already received an application.”;
    }
    public String qualifyApplicant(double g) {
        if (g >= GPA) {
            admission.setState(admission.getProcessingState());
            return “The application is evaluated and will be finalized soon.”;
        } else {
            admission.setState(admission.getWaitingState());
            return “The applicant is denied.”;
        }
    }
    public String admitStudent() {
        return “Must be evaluated first.”;
    }
}
```

# ProcessingState

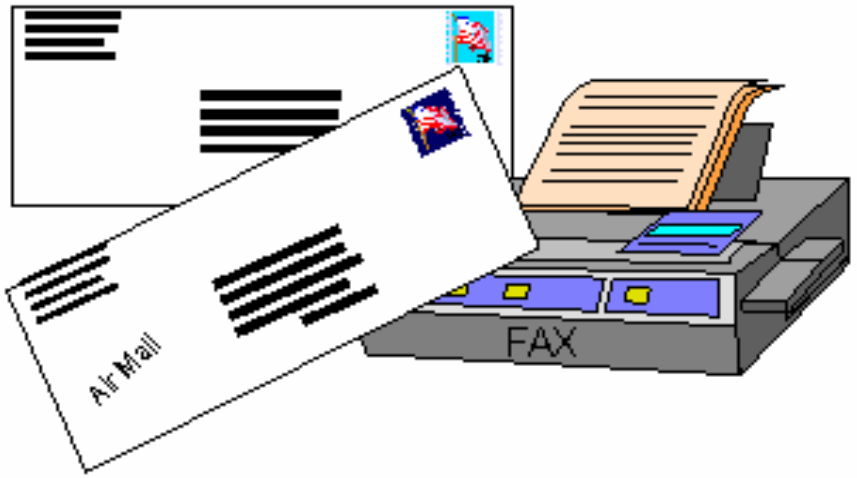
```
public class ProcessingState implements State {
    AdmissionInterface admission;
    public ProcessingState (AdmissionInterface a) {
        admission = a;
    }

    public String receiveApplication() {
        return “The appcation is finalized.”;
    }
    public String qualifyApplicant(double g) {
        return “The appcation is finalized.”;
    }
    public String admitStudent() {
        return “The application is finalized and an admission letter is in the mail.”;
        admission.setState(admission.getWaitingState());
    }
}
```

# Strategy Pattern

# 1. Non-Software Example

## *Strategy*

Faxing, overnight mail, air mail, and surface mail all get a document from one place to another, but in different ways.




Behavioral

# Non-Software Example (cont.)

- Text-based communication corresponds to the *Strategy*.
- Specific forms of text base communication, such as fax, mail, etc. correspond to the *ConcreteStrategies*.
- The context in which textual information is conveyed corresponds to the *Context*.
- Strategies combine families of related algorithms. There are several ways to send textual information.
- *Strategies* allow the algorithm to vary independently of the context. For example, if textual information must be delivered to the next office, placing one copy on your colleague's chair is a valid algorithm, but e-mail could also be used. The proximity does not necessarily dictate the algorithm.
- *Strategies* offer a choice of implementations.

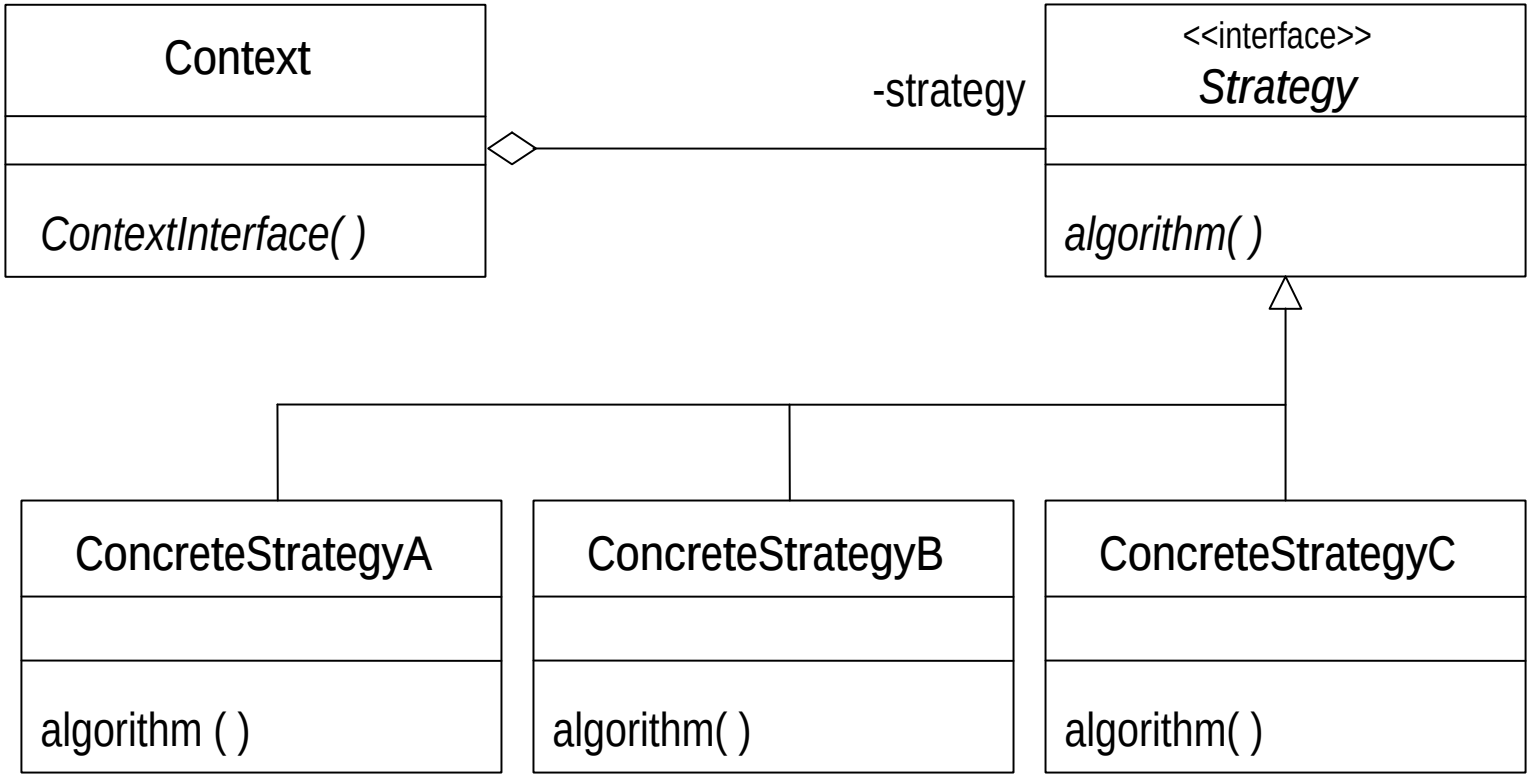
# 2. Problem

- The selection of an *algorithm* that needs to be applied depends upon the client making the request or the data being acted upon.
- We have volatile task-oriented code that we'd like to separate out of our application for easy maintenance.
- We want to avoid muddling how we handle task.
- We want to change the *algorithm* we use for a task at runtime.

# 3. Solution

- *"**Strategy Pattern** defines a family of algorithms, encapsulate each one, and make them interchangeable. **Strategy** lets the algorithm vary independently form clients that use it."* (GoF)
- It is task-oriented.
- It encapsulates behavior (algorithm, strategy).
- It allows for the *selection* to be made based upon *context*.
- It separates the *selection* of algorithm from the *implementation* of the algorithm.
- It simplifies *Client* objects by relieving them of any responsibility for selecting behavior or implementing alternate behaviors.
- It allows us to add a new algorithm easily without disturbing the application using the algorithm.

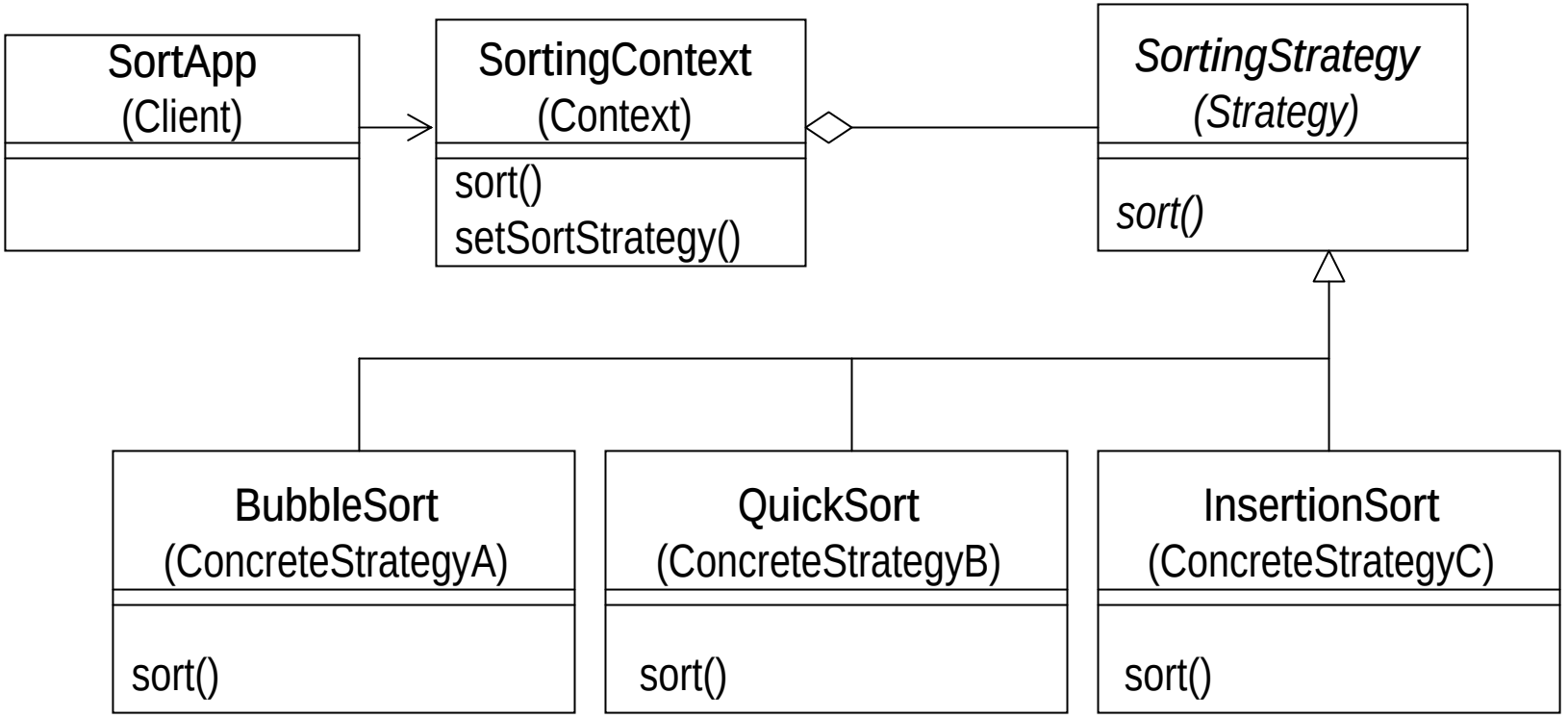
# 4. Class Diagram



# 5. Example

- A client wants to decide at run time what algorithm it should use to sort an array.
- Many different sort algorithms are already available.
- Choice of a sorting method is based on
  - the size of data
  - performance of the method
- To decouple the policy of selecting a sort method from the interface, we encapsulate the sorting method implementations with a *Strategy* pattern.
- Encapsulate different sort algorithms using the *Strategy* pattern.

# Class Diagram



# SortApp

```
// client application class
public class SortApp {
    public static void main(String args[ ]) {
        int data[ ] = {3,6,4,6,7,8,5,6,7,5,3,3};
        SortingContext sc = new SortingContext();
        int sortedList[ ] = sc.sort(data);
    }
}
```

# SortingContext

```
// the context class
public class SortingContext {

  private SortingStrategy  ss;

  public int[ ] sort( int data[ ] ) {
    int size = data.length();
    ss = setSortStrategy(size);
    return ss.sort(data);
  }
```

```
  // choose the sort strategy depending on
  // data size; separate the selection of the
  // algorithm from the implementation

    public SortingStrategy setSortStrategy(int n) {
      if( n >0 && n < 30 )
        ss = new BubbleSort();

      if( n >= 30 && n < 100 )
        ss = new InsertionSort();

      if( n >= 100  )
        ss = new QuickSort();

      return ss;
    }
}
```

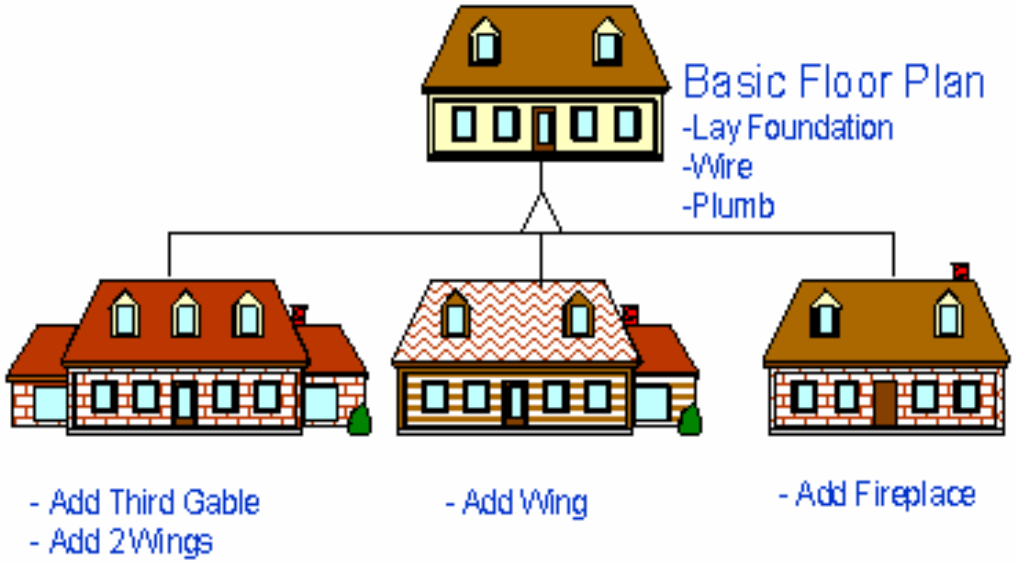
# SortingStrategy, BubbleSort, InsertionSort, QuickSort

```
public interface SortingStrategy {
    public int[ ] sort( int data[ ] );
}

public class BubbleSort implements SortingStrategy {
    public int[ ] sort( int data[ ]) { …  }
}

public class InsertionSort implements SortingStrategy {
    public int[ ] sort( int data[ ]) {  … }
}

public class QuickSort implements SortingStrategy {
    public int[ ] sort( int data[ ] ) {  … }
}
```

# Template Method Pattern

# 1. Non-Software Example

## *Template Method*




Variations added to Template Floor Plan

Subdivision developers often use a *Template Method* to produce a variety of home models from a limited number of floor plans. The basic floor plans are a skeleton, and the differentiation is deferred until later in the building process.

Behavioral

# Non-Software Example (cont.)

- The basic floor plan corresponds to the *AbstractClass*.
- The different elevations correspond to the *ConcreteClasses*.
- *Templates* factor out what is common, so that it can be reused. Multiple elevations can be built from a basic floor plan.
- The specific elevation does not need to be chosen until later in the process.

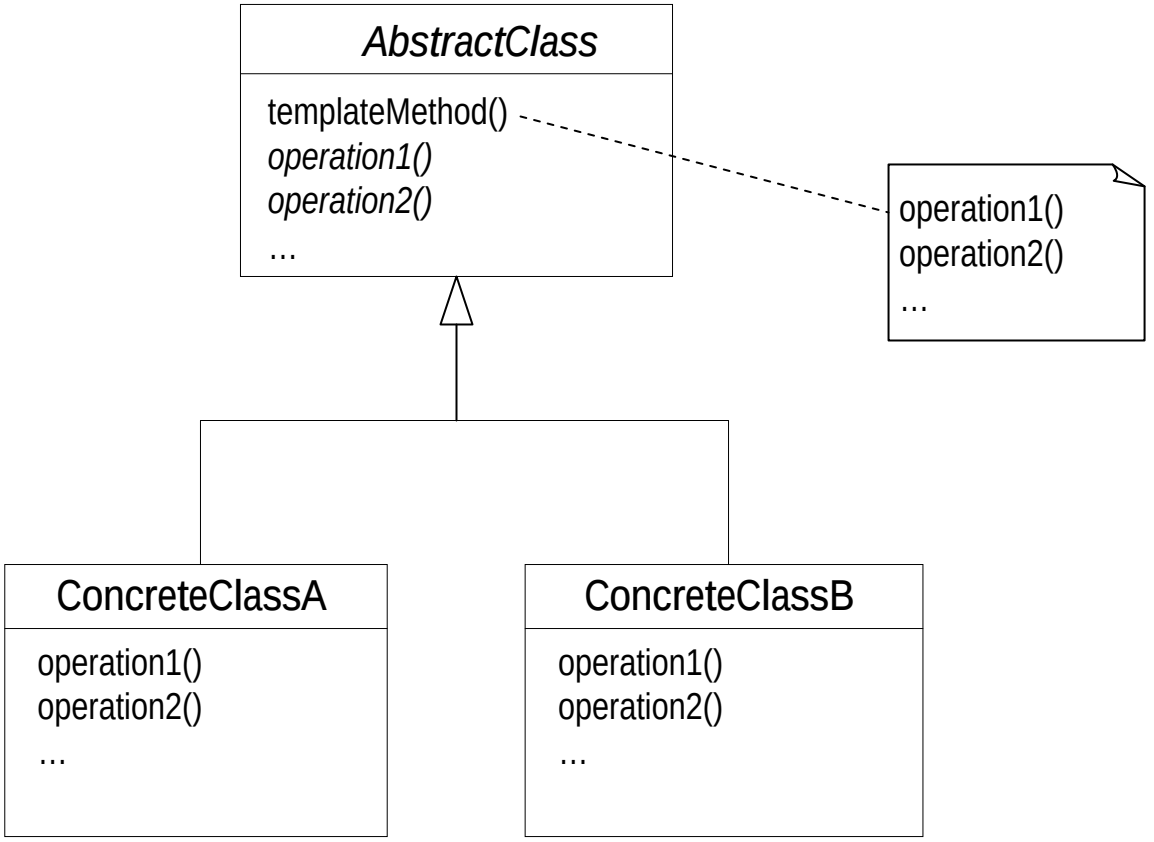
# 2. Problem

- There is a *procedure* or set of steps to follow that is consistent at one level of detail, but individual steps may have different implementation at a lower level of detail.
- Two different components have significant similarities but demonstrate no reuse of common interface or implementation.
- If a change that is common to both components becomes necessary, duplicate effort must be expended.

# 3. Solution

- *"**Template Method Pattern** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. **Template Method** lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."* (GoF)

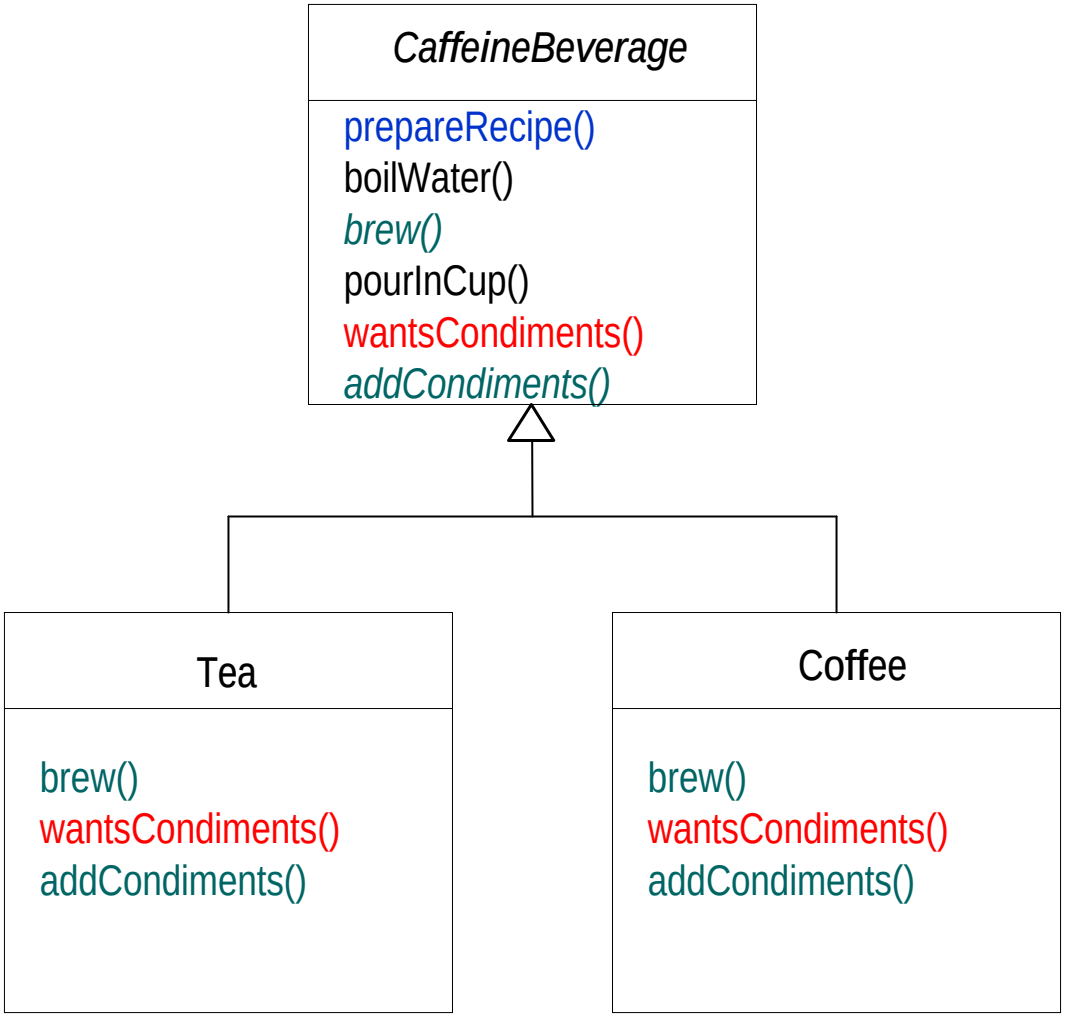
# 4. Class Diagram



# 5. Example

- Use *Template Method* to provide a framework for tea, coffee, and other beverages with caffeine.
- Maximize code reuse
- Keep the algorithm in one place and code changes only need to be made there
- Easy to add new caffeine beverages

# Class Diagram



# BeverageTest

```
// Client
public class BeverageTest {
    public static void main(String[] args) {

        Tea tea = new Tea();
        Coffee coffee = new Coffee();

        System.out.println( “\nMaking tea …” );
        tea.prepareRecipe();

        System.out.println( “\nMaking coffee …” );
        coffee.prepareRecipe();
    }
}
```

# CaffeineBeverage

```
public abstract class CaffeineBeverage {
    void prepareRecipe() {          // template method
        boilWater();
        brew();
        pourInCup();
        if (wantsCondiments()) { addCondimentes(); }
    }

    void boilWater() { System.out.println( “Boiling water” ) };
    abstract void brew();
    void pourInCup () { System.out.println( “Pouring into cup” ) };
    abstract void addCondiments();
    boolean wantsCondiments() { return true }; // a hook operation
}
```

# Tea

```
public class Tea extends CaffeinBeverage {
    public void brew() { System.out.println( “Steep the tea” ); }
    public void addCondiments() {  System.out.println( “Adding lemon” ); }

    public boolean wantsCondiments() {  // a hook operation
        String answer = getUserInput();
        if ( answer.toLowerCase().startsWith( “y” )) { return true; }
        else { return false; }
    }
    private String getUserInput() {
        String answer = null;
        System.out.println ( “Would you like lemon with your tea (y/n)? “);
        BufferedReader in = new BufferedReader ( new InputStreamReader( System.in ));
        // try-catch block
        return answer;
    }
}
```

# Coffee

```
public class Coffee extends CaffeinBeverage {
    public void brew() { System.out.println( “Dripping coffee through filter” ); }
    public void addCondiments() { System.out.println( “Adding sugar and milk” ); }

    public boolean wantsCondiments() {  // a hook operation
        String answer = getUserInput();
        if ( answer.toLowerCase().startsWith( “y” )) { return true; }
        else { return false; }
    {
    private String getUserInput() {
        String answer = null;
        System.out.println ( “Would you like milk and sugar with your coffee (y/n)? “);
        BufferedReader in = new BufferedReader ( new InputStreamReader( System.in ));
        // try-catch block
        return answer;
    }
}
```

# Output

```
%java BeverageTest

Making tea …
Boiling water
Steep the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding lemon

Making coffee …
Boiling water
Dripping coffee though filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```