# EECS3311

Software Design

## BEHAVIORAL DESIGN PATTERNS

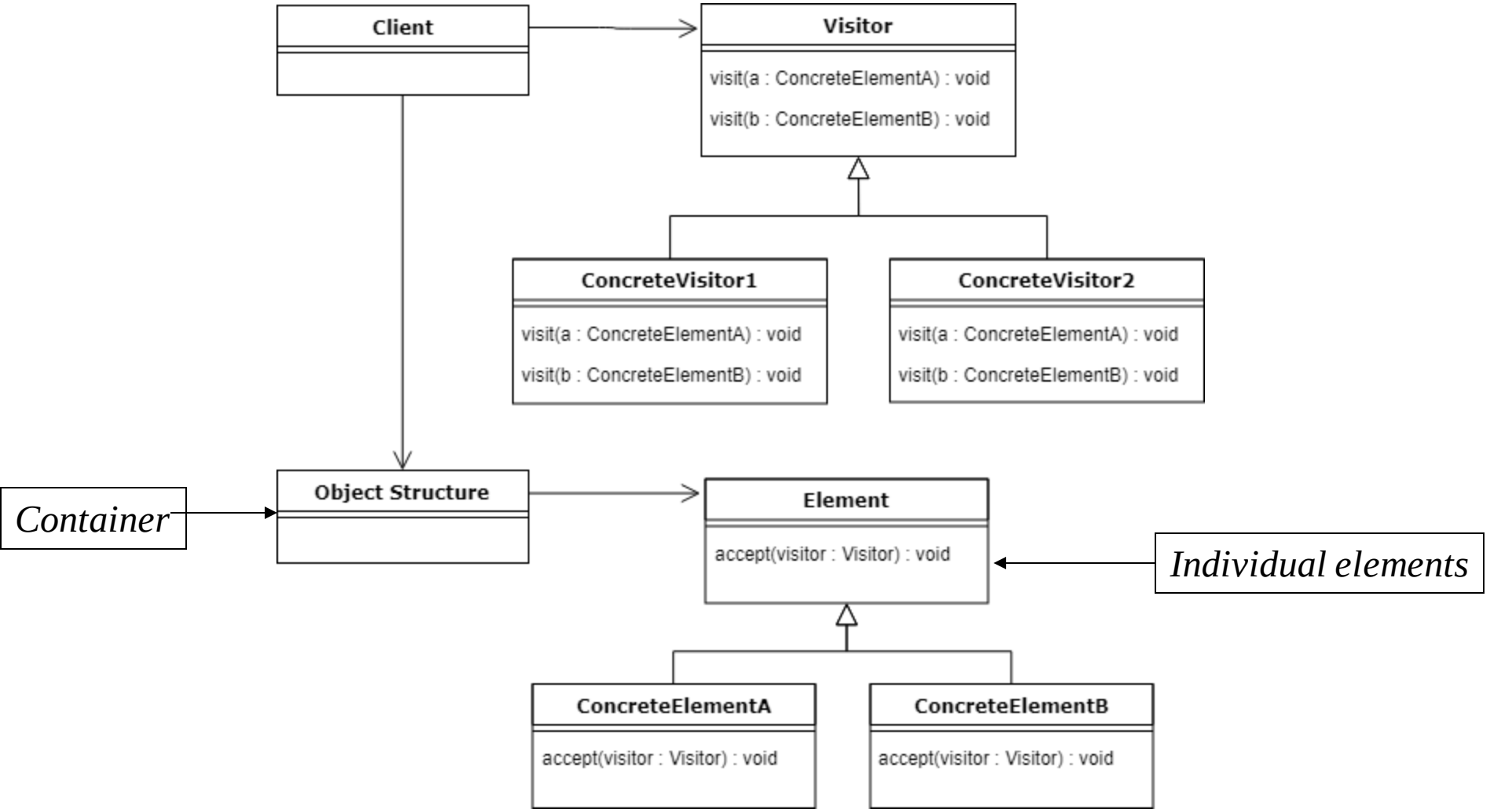
# Classification of GoF Design Pattern

| Creational | Structural | Behavioral |
|---|---|---|
| Factory Method | Adapter | Interpreter |
| Abstract Factory | Bridge | Template Method |
| Builder | Composite | Visitor |
| Prototype | Decorator | Chain of Responsibility |
| Singleton | Flyweight | Command |
| | Facade | Iterator |
| | Proxy | Mediator |
| | | Memento |
| | | State |
| | | Strategy |
| | | Observer |

# Visitor Design Pattern

- Intent
  - Centralize operations on an object structure so that they can vary independently but still behave polymorphically
- Applicability
  - When classes in a collection define many unrelated operations
  - Class relationships of objects in the structure rarely change, but the operations on them change often
  - Apply different operations depending the type of object we visit in a collection

# Visitor – Class Diagram



# Visitor Design Pattern - Example

```
// "Visitor"
public abstract class Visitor {
    public abstract void visit(JSONElement json);
    public abstract void visit(XMLElement xml);
}
// "Element Visitor 1"
public class ElementVisitor1 extends Visitor {
    @Override
    public void visit(JSONElement json) {
        System.out.println("Processing JSON element" +
         " with " + this.getClass().getSimpleName());
    }
    @Override
    public void visit(XMLElement xml) {
        System.out.println("Processing XML element" +
         " with "+ this.getClass().getSimpleName());
    }
}
```

```
//"Element Visitor 2"
public class ElementVisitor2 extends Visitor {
    @Override
    public void visit(JSONElement json) {
        System.out.println("Processing JSON element"
     + " with " + this.getClass().getSimpleName());
    }
    @Override
    public void visit(XMLElement xml) {
        System.out.println("Processing XML element"
      + "with " + this.getClass().getSimpleName());
    }
}
```

# Visitor Design Pattern – Example (2)

```
// "Element"
public abstract class Element {
   public abstract void accept(Visitor visitor);
}

// "JSON Element"
public class JSONElement extends Element {
   @Override
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
}

//"XML Element"
public class XMLElement extends Element {
   @Override
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
}
```

```
// "Document"
public class Document extends Element{
   List<Element> elements = new ArrayList<>();

   public void addElement(Element element) {
      elements.add(element);
   }

   @Override
   public void accept(Visitor visitor) {
      for (Element element : elements)
         element.accept(visitor);
      }
   }
}
```

# Visitor Design Pattern – Example (3)

```
// Client code
public class MainApp {
   public static void main(String[] args) {
      // Create two different visitors
      Visitor v1 = new ElementVisitor1();
      Visitor v2 = new ElementVisitor2();

      // Create document with 3 elements
      Document d = new Document();
      d.addElement(new JSONElement());
      d.addElement(new JSONElement());
      d.addElement(new XMLElement());

      // Visit document with visitors
      d.accept(v1);
      d.accept(v2);
   }
}
```

Output:

***Processing JSON element with ElementVisitor1***
***Processing JSON element with ElementVisitor1***
***Processing XML element with ElementVisitor1***
***Processing JSON element with ElementVisitor2***
***Processing JSON element with ElementVisitor2***
***Processing XML element with ElementVisitor2***

# Visitor Design Pattern – Alternative Example

```
// "Document"
public class Document extends Element{
   List<Element> elements = new ArrayList<>();

   public void addElement(Element element) {
      elements.add(element);
   }

   @Override
   public void accept() {
      for (Element element : elements) {
         Visitor visitor = VisitorFactory.create(element);
         visitor.visit(element);
      }
   }
}
```

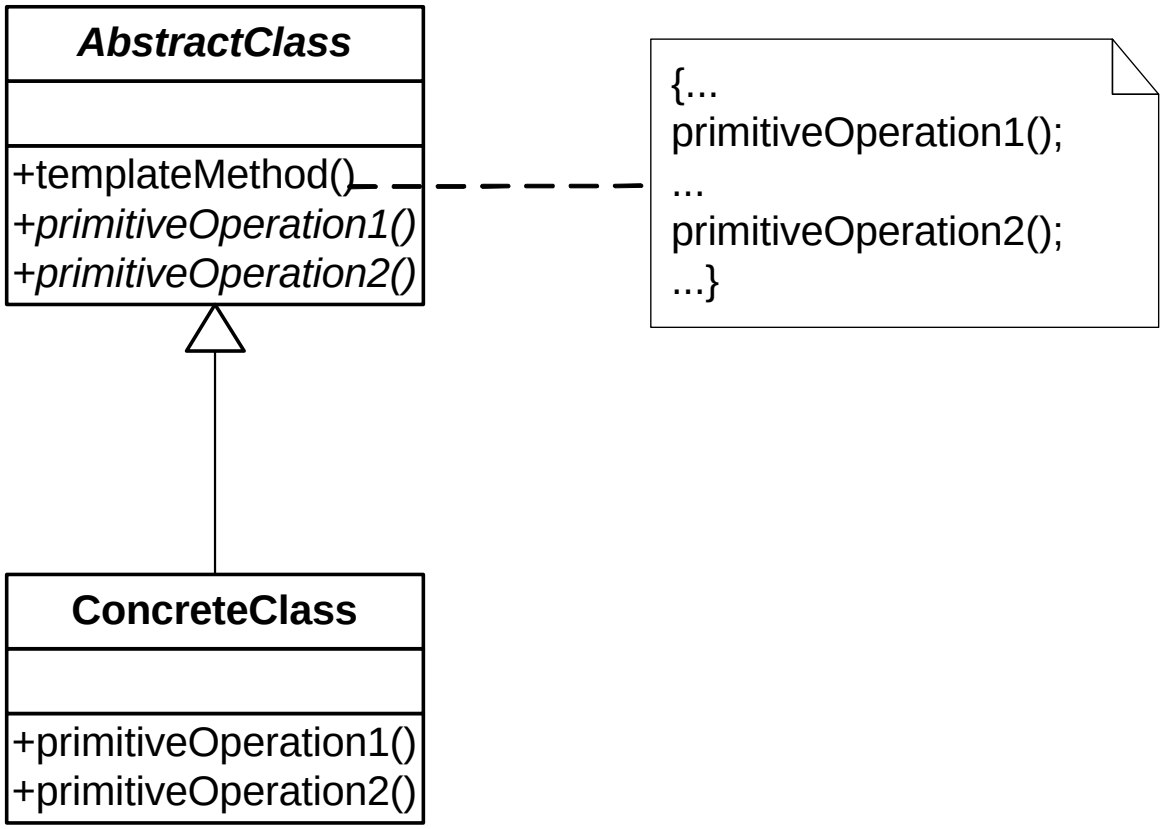
Definition of the class representing the container changes

Different visitor is created based on individual element

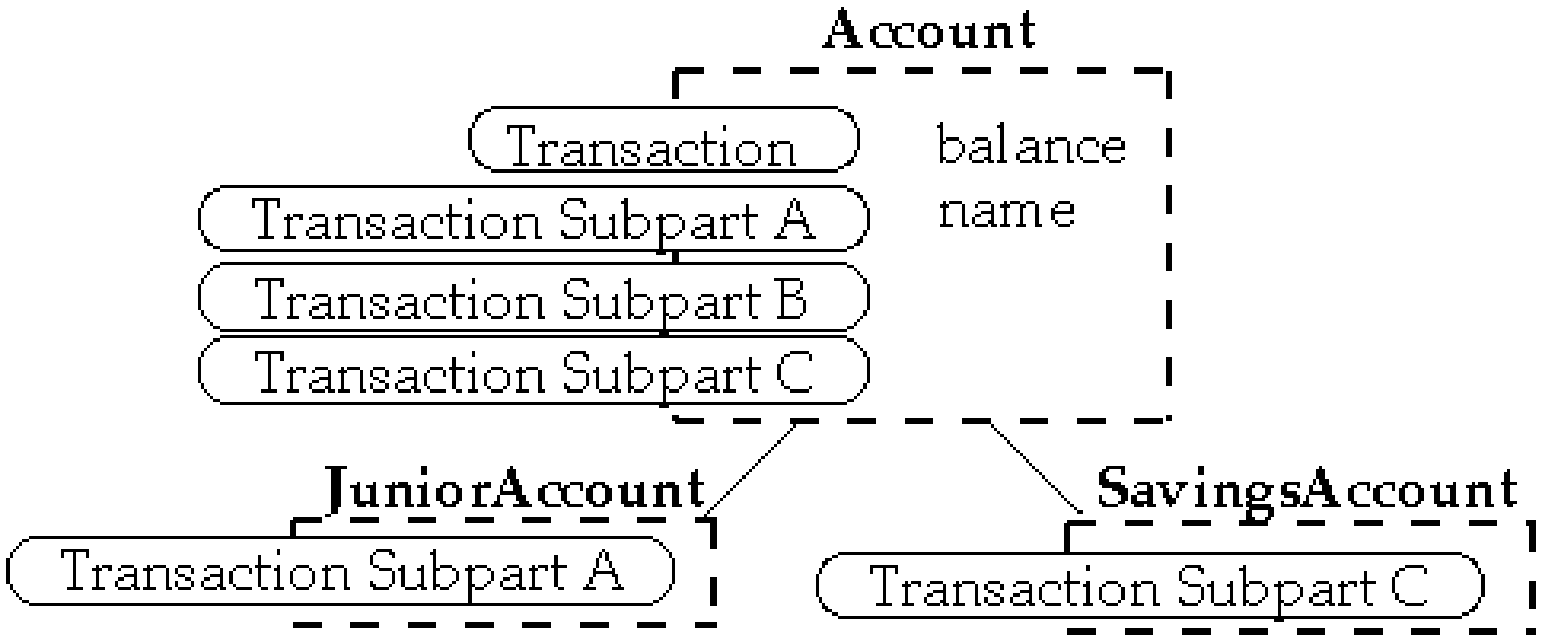
# Template Method Design Pattern

- **Intent**
  - Definition of the general structure (steps) of an algorithm with the ability to change the logic of some steps in specific embodiments or cases
- **Application**
  - When the basic steps of an algorithm can be implemented in a class and the subclasses can implement variations of specific sub-steps
  - When we want to bring together the basic steps of an algorithm in a class to promote reuse of code

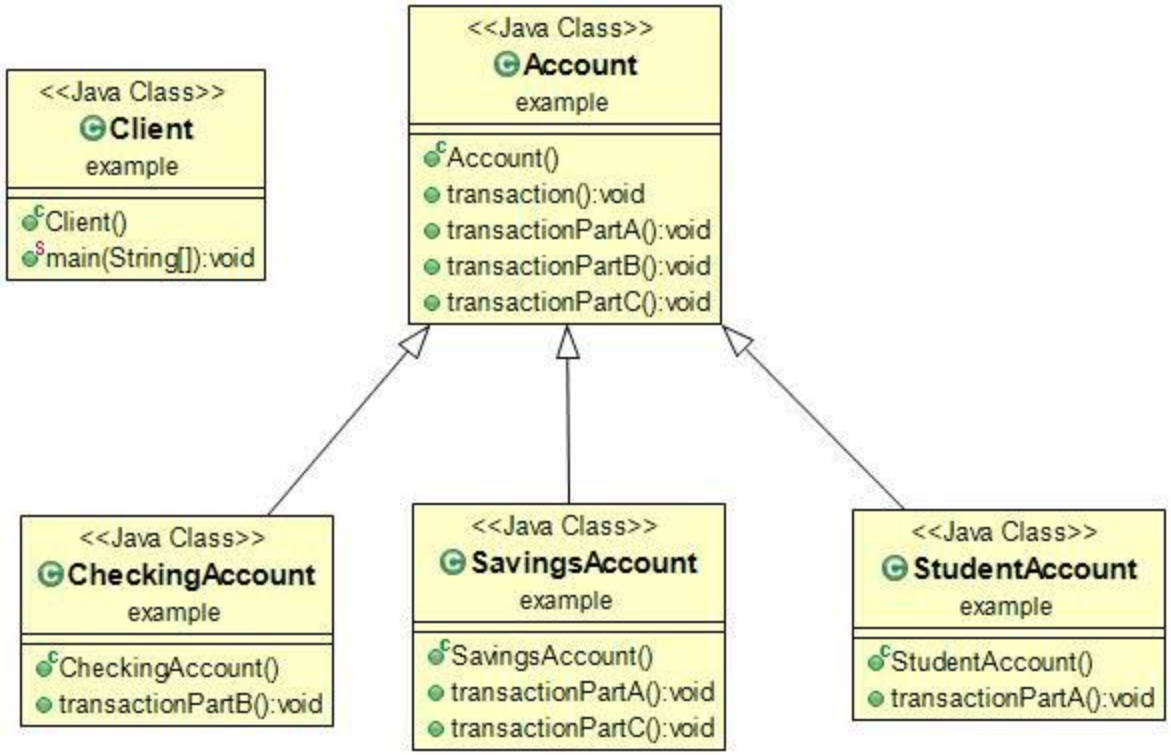
# Template Method Design Pattern – Class Diagram



# Template Method – Example (1)



# Template Method - Example



# Template Method Design Pattern - Example

```
public class Account {
    public void transaction() {
     System.out.println("Starting Transaction
         in transaction class ");
     this.transactionPartA();
     this.transactionPartB();
     this.transactionPartC();
   }

    public void transactionPartA() {
     System.out.println("Doing PartA in
           transaction class ");
   }

   public void transactionPartB() {
     System.out.println("Doing PartB in
       transaction class ");
   }

   public void transactionPartC() {
    System.out.println("Doing PartC in
     transaction class ");
   }
}
```

```
public class CheckingAccount extends Account {

   public void transactionPartB() {
    System.out.println("Doing PartB in
       CheckingAccount class ");
  }
}
```

```
public class SavingsAccount extends Account {

public void transactionPartA() {
    System.out.println("Doing PartA in
      SavingsAccount class ");
   }

   public void transactionPartC() {
    System.out.println("Doing PartC in
      SavingsAccount class ");
   }
}
```

# Template Method Design Pattern - Example

```
public class Client {

  public static void main(String[] args) {
    System.out.println("Let's have a generic
      account");
    Account anAccount = new Account();
    anAccount.transaction();
    System.out.println("---------------");

    System.out.println("Let's have a checking
      account");
    anAccount = new CheckingAccount();
    anAccount.transaction();

    System.out.println("---------------");

    System.out.println("Let's have a savings
      account");

    anAccount = new SavingsAccount();
    anAccount.transaction();
  }
}
```

Output

```
Let's have a generic account
Starting Transaction in transaction class
Doing PartA in transaction class
Doing PartB in transaction class
Doing PartC in transaction class
---------------
Let's have a checking account
Starting Transaction in transaction class
Doing PartA in transaction class
Doing PartB in CheckingAccount class
Doing PartC in transaction class
---------------
Let's have a savings account
Starting Transaction in transaction class
Doing PartA in SavingsAccount class
Doing PartB in transaction class
Doing PartC in SavingsAccount class
```

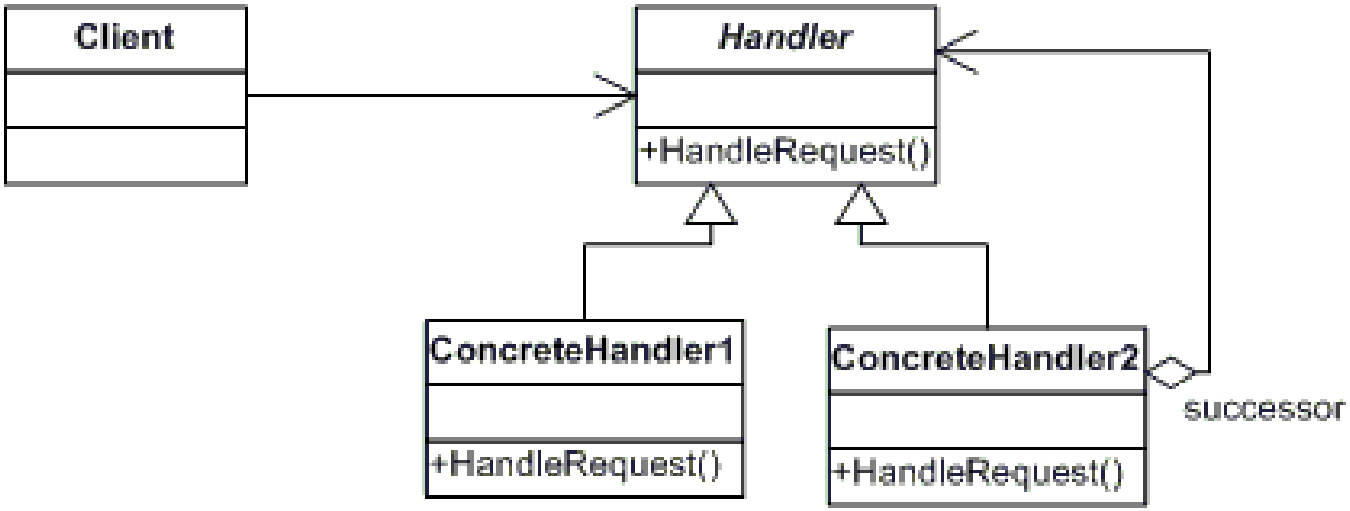
# Template Method - Comments

- Consequences
  - \+ Leads to a revesal of control (“Hollywood principle”: don't call us – we'll call you)
  - \+ Code reuse
  - – Use of subclasses for specializing the behaviour of the sub-steps of the main algorithm/process)
- Implementation
  - – Virtual vs. non-virtual template method
  - – Number of sub-steps
- Known Uses
  - – In most large Object Oriented applications, especially in OO frameworks

# Design Pattern Chain of Responsibility

- **Intent**
  - The Chain of Responsibility pattern is used to promote low coupling between the sender of a request and the objects that handle the request. The pattern enables one or more objects to handle the request
- **Application**
  - The objects that can handle the request are "chained" so that one passes the request to another when and as needed
  - The result is the request to be able to be managed by one or more objects

# Chain of Responsibility – Class Diagram



# Chain of Responsibility – Example

```
// "Handler"
public abstract class Handler {
   protected Handler successor;
   public void setSuccessor(Handler successor){
      this.successor = successor;
   }

   public abstract void handleRequest(int request);
}

// "Concrete Handler 1"
public class ConcreteHandler1 extends Handler {
   @Override
   public void handleRequest(int request) {
      if (request >= 0 && request < 10) {
         System.out
            .println(this.getClass().getSimpleName() +
                     " handled request " + request);
      } else if (successor != null) {
         successor.handleRequest(request);
      }
   }
}
```

```
// "Concrete Handler 2"
public class ConcreteHandler2 extends Handler {
   @Override
   public void handleRequest(int request) {
      if (request >= 10 && request < 20) {
         System.out.println(
               this.getClass().getSimpleName() +
               " handled request " + request);
      } else if (successor != null) {
         successor.handleRequest(request);
      }
   }
}

// "Concrete Handler 3"
public class ConcreteHandler3 extends Handler {
   @Override
   public void handleRequest(int request) {
      if (request >= 20 && request < 30) {
         System.out.println(
               this.getClass().getSimpleName() +
               " handled request " + request);
      } else if (successor != null) {
         successor.handleRequest(request);
      }
   }
}
```

# Chain of Responsibility – Client Code

```
// Client code
public class MainApp {
   public static void main(String[] args) {
      // Setup Chain of Responsibility
      Handler h1 = new ConcreteHandler1();
      Handler h2 = new ConcreteHandler2();
      Handler h3 = new ConcreteHandler3();
      h1.setSuccessor(h2);
      h2.setSuccessor(h3);

      // Generate and process request
      int[] requests = {2, 5, 14, 22, 18, 3, 27, 20};
      for (int request : requests) {
         h1.handleRequest(request);
      }
   }
}
```

Output:

**ConcreteHandler1 handled request 2**
**ConcreteHandler1 handled request 5**
**ConcreteHandler2 handled request 14**
**ConcreteHandler3 handled request 22**
**ConcreteHandler2 handled request 18**
**ConcreteHandler1 handled request 3**
**ConcreteHandler3 handled request 27**
**ConcreteHandler3 handled request 20**

# State design Pattern

- **Intent**
  - It allows for an object of a class to change its behavior according to the state it is. This behavior is applied to all such objects which are in the same state. From an outside client it looks like the object has changed class!

- **Applicability**
  - The State design pattern is useful when we would like the object to be able to denote the different behaviors objects may have when they are in different states (e.g. a robot will move slowly when responds to the command (i.e. method) *move* if its battery state is *low* in order to conserve energy, and move fast when the battery state is *high*)

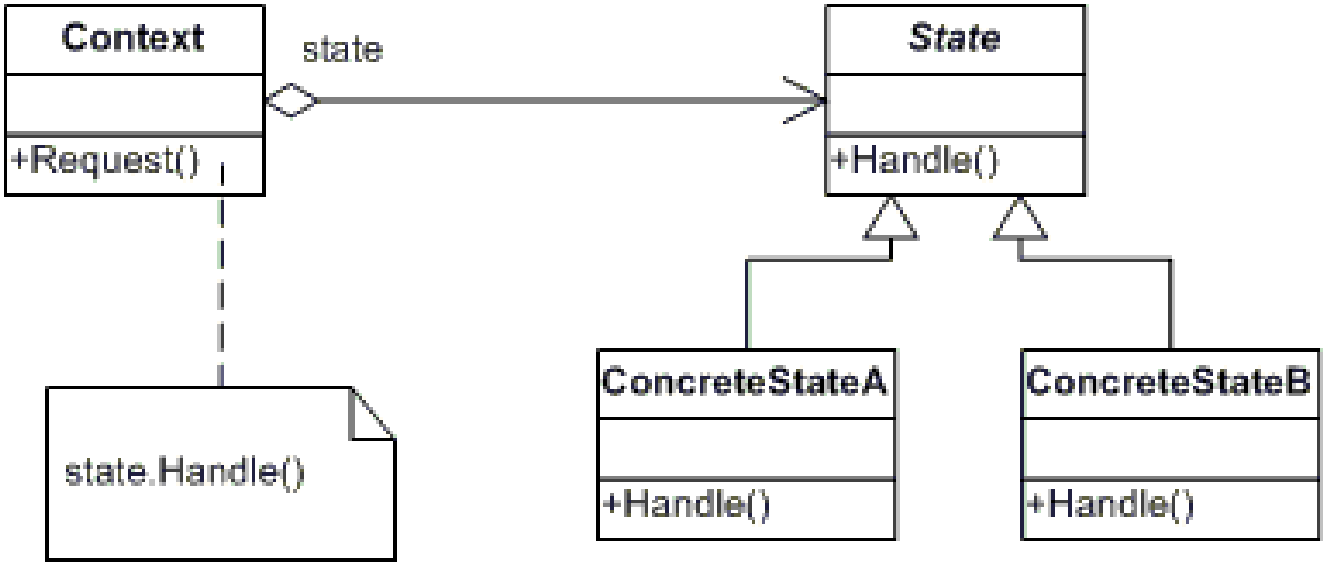
  - The design benefit of the State Design Pattern is that the logic which is related to an object's state is concentrated in the classes that denote the corresponding state

# State Design Pattern – Structural Elements

The structural elements (i.e classes) of the State Design Pattern are:

- **Context**
  - Defines an interface the client code accesses. Such an interface is the method request (see next page)
  - It associates with an entity that refers to a specific state. The entity is an implementation (e.g. ConcreteStateA, ConcreteStateB) of a class that inherits from the class State.
- **State**
  - Defines the interface that encapsulates the behavior for a specific state of a context. This interface is the method Handle
- **Concrete State**
  - It is the subclass of State which implements using polymorphism the behavior that is associated with a given Context.

# State Design Pattern – Class Diagram



# State Design Pattern - Example

```
// "State"
public abstract class State {
   public abstract void handle(Context context);
}

// "Concrete State A"
public class ConcreteStateA extends State {
   @Override
   public void handle(Context context) {
      context.setState(new ConcreteStateB());
   }
}

// "Concrete State B"
public class ConcreteStateB extends State {
   @Override
   public void handle(Context context) {
      context.setState(new ConcreteStateA());
   }
}
```

```
// "Context"
public class Context {
   private State state;

   public Context(State state) {
      this.state = state;
   }

   public void setState(State state) {
      System.out.println("State: " +
           state.getClass().getSimpleName());
      this.state = state;
   }

   public State getState() {
      return state;
   }

   public void request() {
      state.handle(this);
   }
}
```

# State Design Pattern – Client Code Example

```
// Client code
public class MainApp {
    public static void main(String[] args) {
        // Setup context in a state
        Context c = new Context(new ConcreteStateA());
        // Issue requests, which toggles state
        c.request();
        c.request();
        c.request();
        c.request();
    }
}
```

Output:

**State: ConcreteStateA**
**State: ConcreteStateB**
**State: ConcreteStateA**
**State: ConcreteStateB**
**State: ConcreteStateA**

# Memento Design Pattern

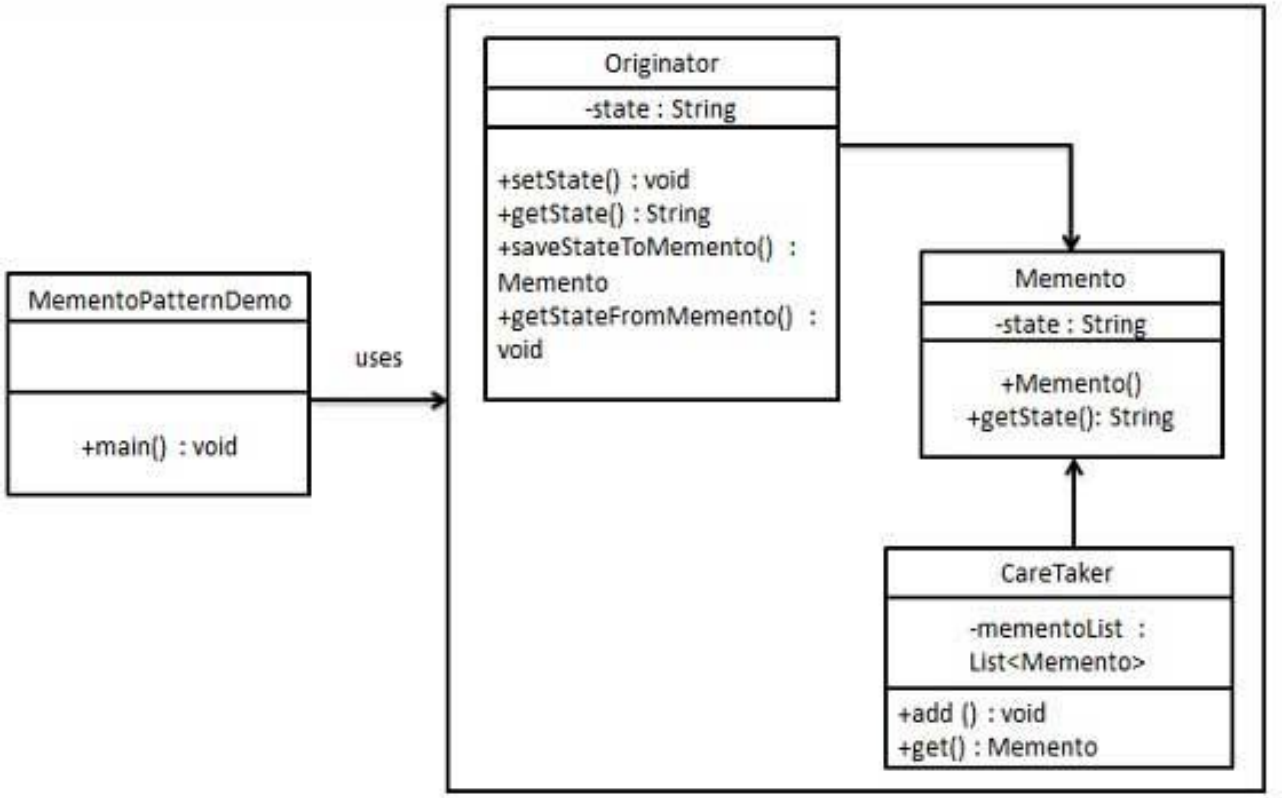
- **Intent**
  - It allows for the active state of an object to be archived, so that if need be, it can be later restored.
  - The object state may change in between, but the object may revert back to any its archived states
- **Applicability**
  - The Memento design pattern is useful when we have objects that change their state dynamically as the program runs, but we need to revert back to a prior state given the business logic of the application, or when we want to revert back to a “safe” or “default” state

# Memento Design Pattern – Structural Elements

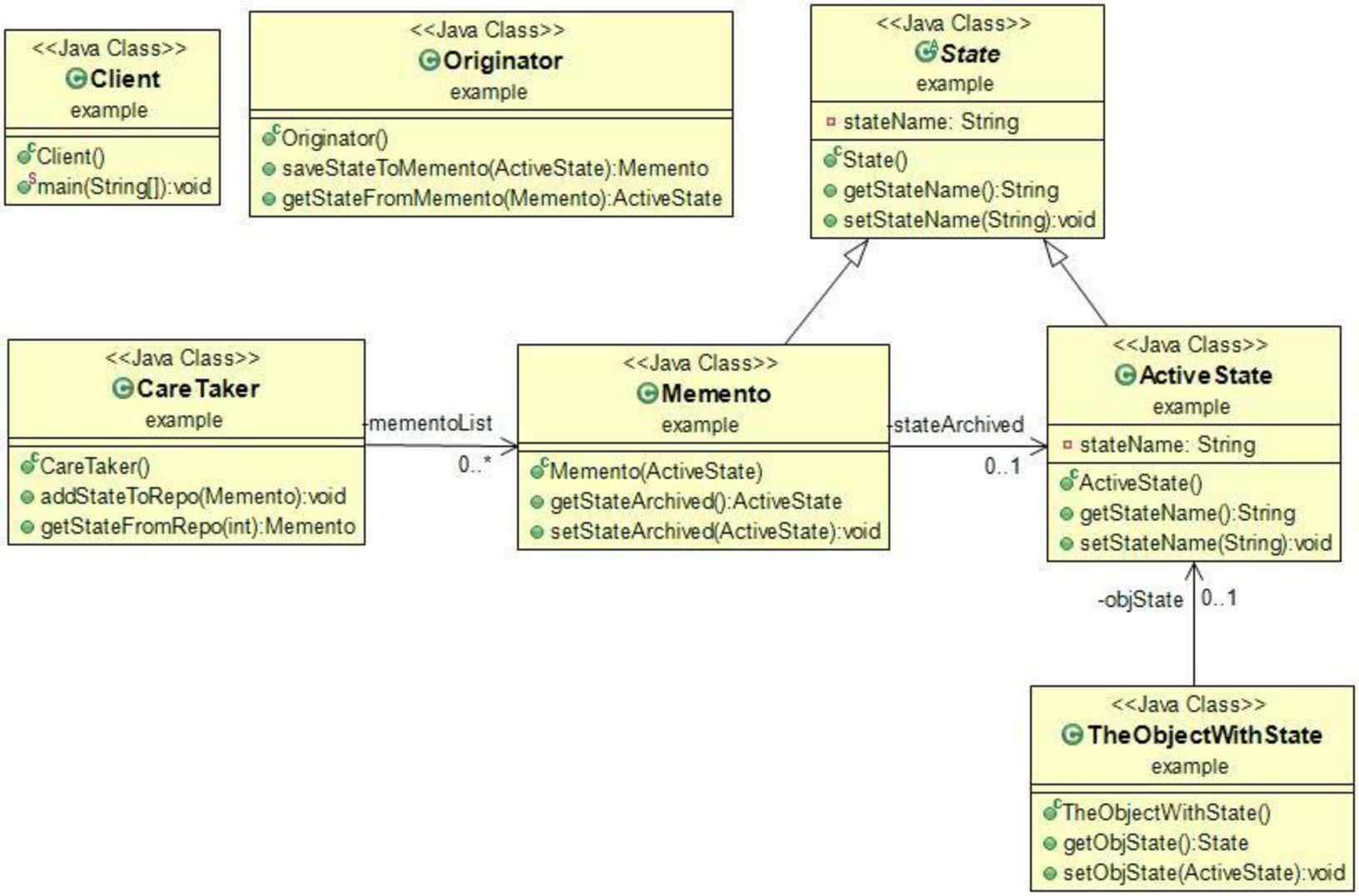
The structural elements (i.e classes) of the Memento Design Pattern are:

- Memento
  - contains the state of an object to be restored.
- Originator
  - creates and stores states in Memento objects
- Caretaker
  - is responsible to restore object state from Memento.

# Memento Class Diagram



# Memento Example



# Memento Design Pattern - Example

```
public class ActiveState extends State {
  private String stateName;
}
```

```
public class TheObjectWithState {
  private ActiveState objState;
}
```

```
public class Memento extends State {
  private ActiveState stateArchived;

  public Memento(ActiveState stateToBeArchived) {
    super();
    this.stateArchived = stateToBeArchived;
  }

  public ActiveState getStateArchived() {
    return stateArchived;
  }

  public void setStateArchived(ActiveState
                stateArchived) {
    this.stateArchived = stateArchived;
  }
}
```

```
public class Originator {

   public Memento saveStateToMemento(ActiveState
                                      state) {
      Memento aMemento = new Memento(state);
      return aMemento;
   }

    public ActiveState getStateFromMemento(Memento
                           memento) {
      return memento.getStateArchived();

    }
}
```

```
public class CareTaker {
   private List<Memento> mementoList =
            new ArrayList<Memento>();

   public void addStateToRepo(Memento memento){
       mementoList.add(memento);
   }

   public Memento getStateFromRepo(int index){
      return mementoList.get(index);
   }
}
```

Output

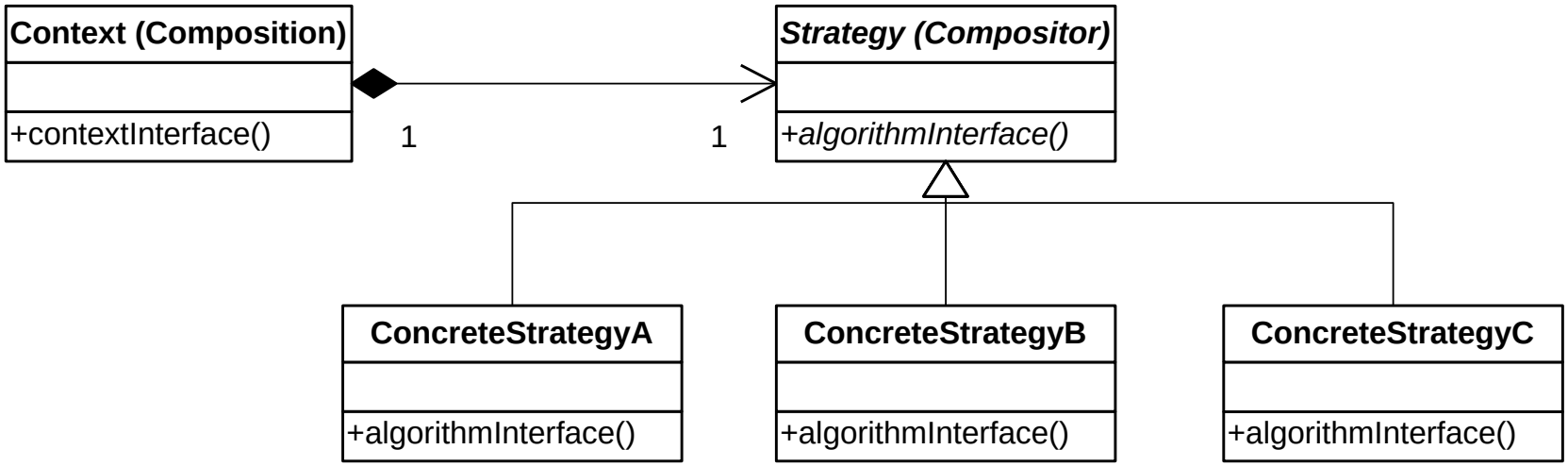
```
The state of the ObjectWithState is State 1
The state of the ObjectWithState is State 2
Restoring the state of the ObjectWithState
The state now is State 1
```

```
public class Client {
   public static void main(String[] args) {
     TheObjectWithState anObj = new ….
     ActiveState activeState = new ActiveState();
     activeState.setStateName("State 1");
     anObj.setObjState(activeState);
     System.out.println("The state of the
        ObjectWithState is " +
        anObj.getObjState().getStateName());
     CareTaker aCareTaker = new CareTaker();
     Originator anOriginator = new Originator();
     Memento aMemento =
       anOriginator.saveStateToMemento(activeState);
     aCareTaker.addStateToRepo(aMemento);
     anOriginator.saveStateToMemento(activeState);
    // Now change the state of the object
     activeState = new ActiveState();
     activeState.setStateName("State 2");
     anObj.setObjState(activeState);
     System.out.println("The state of the
     ObjectWithState is " + ……. );
     System.out.println("Restoring the state ….);
    aMemento = aCareTaker.getStateFromRepo(0);
    anObj.setObjState(aMemento.getStateArchived());
    System.out.println("The state now is " +
        anObj.getObjState().getStateName());
   }
}
```

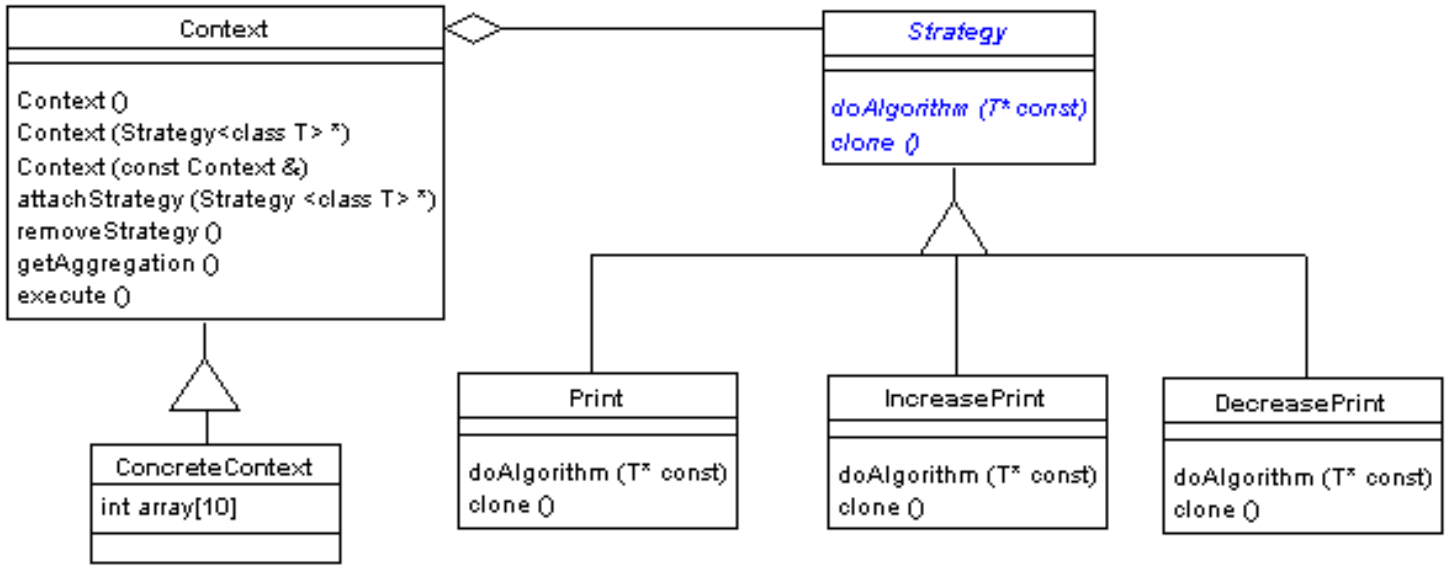
# Strategy Design Pattern

- Intent
  - Design a family of algorithms which can be called through a well defined common interface, so that the client programs and the algorithms that can be called can change independently
- Applicability
  - When an object should associate with one or more algorithms to perform a task, and the algorithms can be encapsulated, and a common interface can be used to call all these different algorithms

# Strategy Design Pattern – Class Diagram



# Strategy Design Pattern – Class Diagram



# Strategy – Example (1)

```
// "Context"
public class Context {
   private Strategy strategy;
   private Integer[] array = new Integer[]
           {5, 4, 9, 1, 8, 3, 2, 7, 0, 6};

   public Context(Strategy strategy) {
      this.strategy = strategy;
   }

   public Integer[] getArray() {
      return array;
   }

   public void setStrategy(Strategy strategy) {
      this.strategy = strategy;
   }

   public void execute() {
      strategy.doAlgorithm(this);
   }
}
```

```
// "Strategy"
public abstract class Strategy {
   public abstract void doAlgorithm(Context context);
}

// "Simple Print Strategy"
public class SimplePrintStrategy extends Strategy {
   @Override
   public void doAlgorithm(Context context) {
      // Get array to print
      Integer[] array = context.getArray();
      System.out.println("Simple Print: " + array);
   }
}

// "Increase Print Strategy"
public class IncreasePrintStrategy extends Strategy {
   @Override
   public void doAlgorithm(Context context) {
      // Get array to print
      Integer[] array = context.getArray();
      // Sort array in ascending order
      Arrays.sort(array);
      System.out.println("Increase Print: " + array);
   }
}
```

# Strategy – Example (2)

```
// "Decrease Print Strategy"
public class DecreasePrintStrategy extends Strategy {
   @Override
   public void doAlgorithm(Context context) {
      // Get array to print
      Integer[] array = context.getArray();
      // Sort array in descending order
      Arrays.sort(array, Collections.reverseOrder());
      System.out.println("Decrease Print: " + array);
   }
}
```

# Strategy – Client Code Example (3)

```
// Client code
public class MainApp {
   public static void main(String[] args) {
      // Create a context with simple printing strategy
      Context context = new Context(new SimplePrintStrategy());
      context.execute();
      System.out.println("**********************");

      // Change the strategy to increasing
      context.setStrategy(new IncreasePrintStrategy());
      context.execute();
      System.out.println("**********************");

      // Change the strategy to decreasing
      context.setStrategy(new DecreasePrintStrategy());
      context.execute();
      System.out.println("**********************");
   }
}
```

Output:

**Simple Print: {5, 4, 9, 1, 8, 3, 2, 7, 0, 6}**
**********
**Increase Print: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}**
**********
**Decrease Print: {9, 8, 7, 6, 5, 4, 3, 2, 1, 0}**
**********

# Strategy - Comments

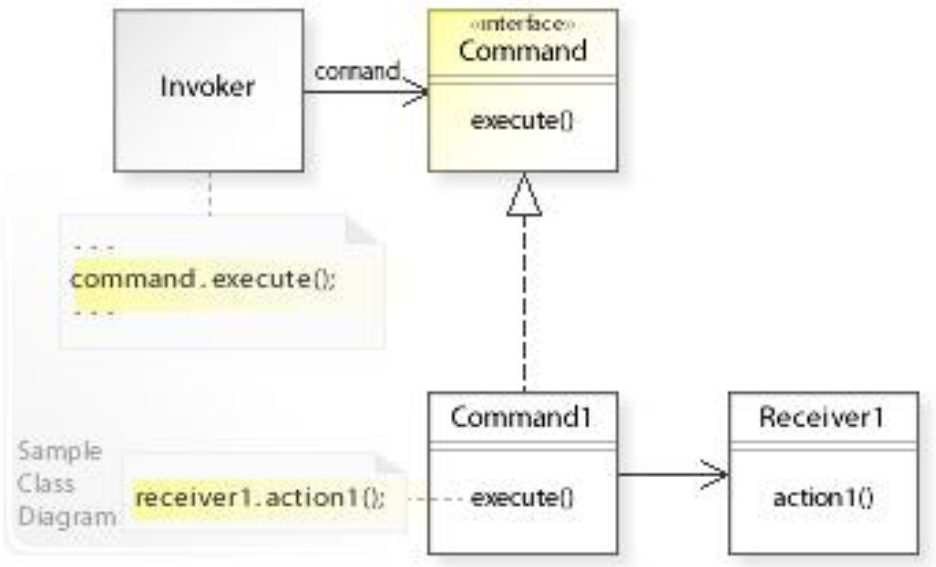
- **Consequences**
  - \+ Flexibility, code reuse
  - \+ We can dynamically at run time change the algorithms that can be called from the client code
  - – Cost to associate algorithms to objects
  - – Need for common interface for all algorithms used (Inflexible strategy interface)
- **Implementation**
  - – Need to pass data between the algorithm (strategy) and the corresponding object (context)

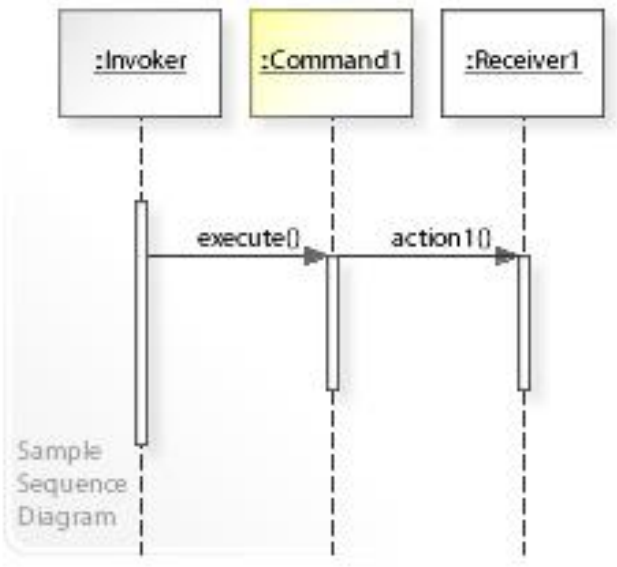
# Command Design Pattern

- The Command pattern is a behavioral pattern.
- The intent is that the request is wrapped under an object as command and passed to invoker object.
- The Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

https://www.tutorialspoint.com/design_pattern/command_pattern.htm

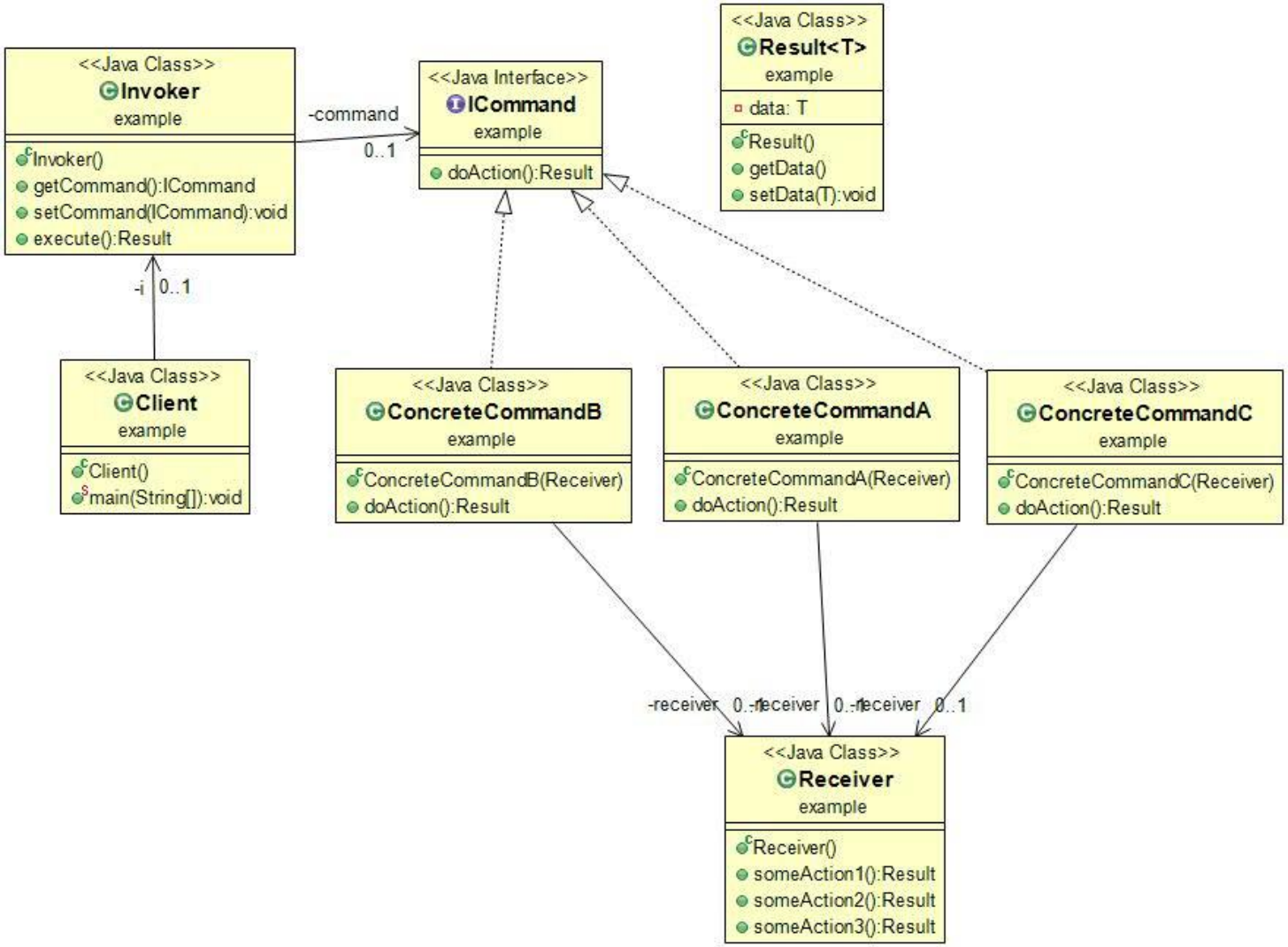
# Command Design Pattern – Class Diagram




https://en.wikipedia.org/wiki/Command_pattern

# Command Design Pattern - Example



# Command Design Pattern – Example

```
public interface ICommand {
    public Result doAction();
};
```

```
public class ConcreteCommandA implements
             ICommand {
  private Receiver receiver;

  public ConcreteCommandA(Receiver receiver){
     super();
     this.receiver = receiver;
  }
  public Result doAction() {
    return receiver.someAction1();
  }
}
```

```
public class Result<T> {

// Here we encapsulate the returned results
   private T data;
}
```

```
public class ConcreteCommandB implements
             ICommand {
  private Receiver receiver;

  public ConcreteCommandA(Receiver receiver){
     super();
     this.receiver = receiver;
  }
  public Result doAction() {
    return receiver.someAction2();
  }
}
```

```
public class ConcreteCommandC implements
             ICommand {
  private Receiver receiver;

  public ConcreteCommandA(Receiver receiver){
     super();
     this.receiver = receiver;
  }
  public Result doAction() {
    return receiver.someAction3();
  }
}
```

# Command Design Pattern – Example

```
public class Invoker {
  private ICommand command;

  public ICommand getCommand() {
    return command;
}
public void setCommand(ICommand command) {
    this.command = command;
}
public Result execute() {
        return command.doAction();
    }
}
```

```
public class Receiver {
  public Result someAction1() {
    System.out.println("Some Action 1 was
       invoked");
    Result result = new Result<Boolean>();
    result.setData(true);
    return result;
  }

  public Result someAction2() {
    System.out.println("Some Action 2 was
      invoked");
    Result result = new Result<String>();
    result.setData("A String Result");
    return result;
  }

  public Result someAction3() {
    System.out.println("Some Action 3 was
      invoked");
    Result result = new Result<Float>();
    result.setData(10.0f);
    return result;
  }
}
```

# Command Design Pattern – Example

```
public class Client {

  private static Invoker i = new Invoker();

  public static void main(String[] args) {

    Receiver r = new Receiver();
   i.setCommand(new ConcreteCommandA(r));
   System.out.println("The result is " +
   i.execute().getData().toString());

   i.setCommand(new ConcreteCommandB(r));
   System.out.println("The result is " +
   i.execute().getData().toString());

   i.setCommand(new ConcreteCommandC(r));
   System.out.println("The result is " +
   i.execute().getData().toString());
   }
}
```

Output

```
Some Action 1 was invoked
The result is true
Some Action 2 was invoked
The result is A String Result
Some Action 3 was invoked
The result is 10.0
```

# Observer Design Pattern

- **Intent**
  - Define a one-to-many dependency between objects so that when one object (i.e. the *subject*) changes state, all its dependents (i.e. *observers*) are notified and updated (or perform an operation) automatically.

- **Applicability**
  - When an abstraction has two aspects, one dependent on the other.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should notify other objects without making assumptions about who these objects are.

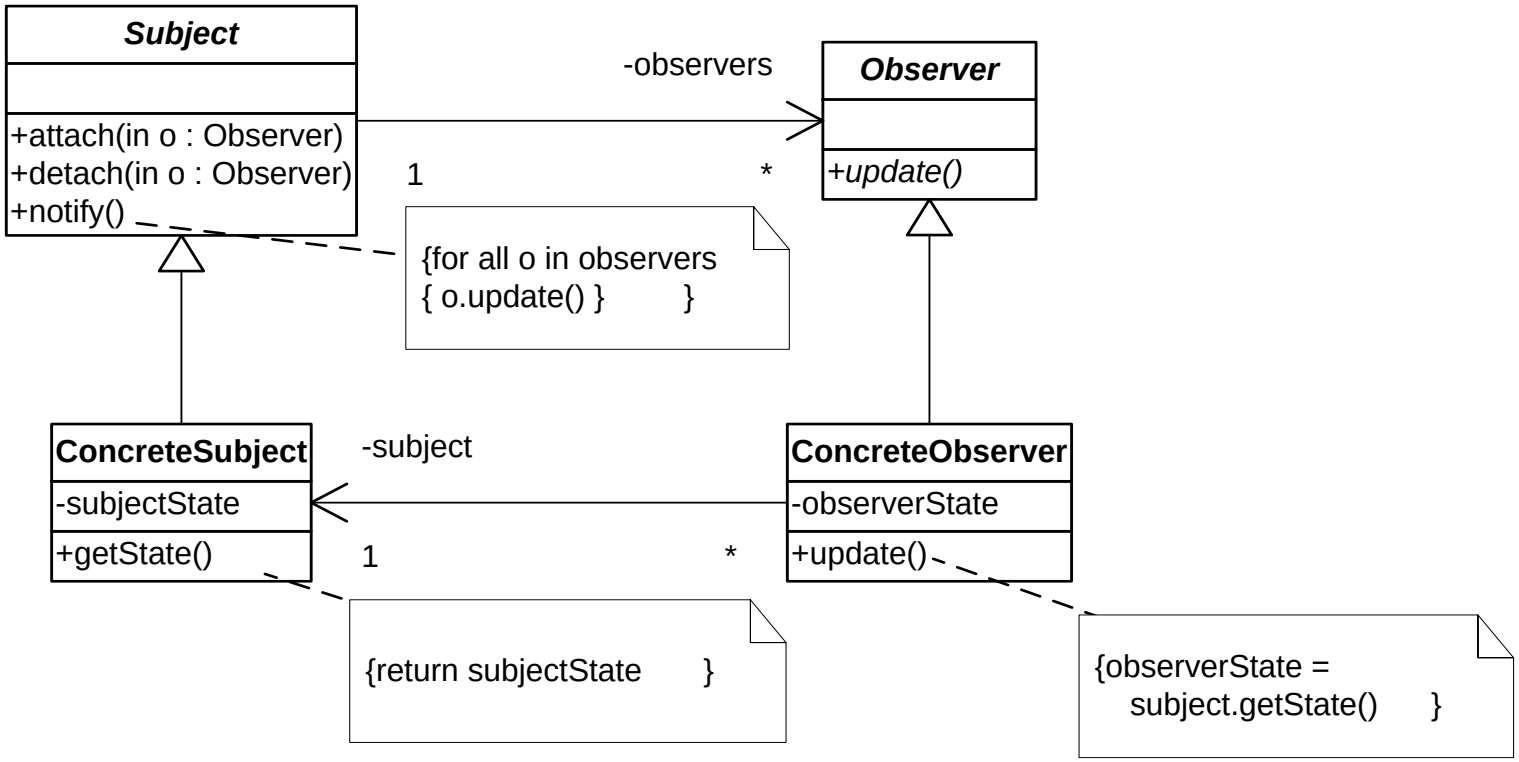
# Observer Design Pattern (Cont'd)

- **Consequences**
  - \+ Modularity: *subject* and *observers* may vary independently
  - \+ Extensibility: can define and add any number of *observers*
  - \+ Customizability: different *observers* provide different views of subject
  - – Unexpected updates: *observers* don't know about each other
  - – Update overhead: might need hints
- **Implementation**
  - – Subject-observer mapping
  - – Dangling references
  - – Avoiding observer-specific update protocols: the push and pull models
  - – Registering modifications of interest explicitly

# Observer Design Pattern (Cont'd)

- **Known uses**
  - Smalltalk model-view-controller (MVC)
  - Interviews (subjects and views)
  - Andrew (data objects and views)
- **Benefits**
  - Design reuse
  - Uniform design vocabulary
  - Enhance understanding, restructuring
  - Basis for automation

# Observer – Class Diagram



# Observer - Example (1)

```
// "Subject"
public abstract class Subject {
  private List<Observer> observers =
                      new ArrayList<>();

  public void attach(Observer observer) {
    observers.add(observer);
  }

  public void detach(Observer observer) {
    observers.remove(observer);
  }

  public void notifyObservers() {
    for (Observer observer : observers)
      observer.update(this);
  }
}

// "Observer"
interface Observer {
  public void update(Subject subject);
}
```

```
// " Clock Timer"
public class ClockTimer extends Subject {
  private int hour, minute, second;

  public int getHour() {
    return hour;
  }


  public int getMinute() {
    return minute;
  }


  public int getSecond() {
    return second;
  }

  public void tick() {
    // Update internal time-keeping state
    // ...

    notifyObservers();
  }
}
```

# Observer – Example (2)

```java
// "Digital Clock"
public class DigitalClock implements Observer {
  private ClockTimer subject;

  public DigitalClock(ClockTimer subject) {
    this.subject = subject;
    subject.attach(this);
  }

  @Override
  public void update(Subject changedSubject) {
    if (changedSubject.equals(subject))
      draw();
  }

  private void draw() {
    int hour = subject.getHour();
    int minute = subject.getMinute();
    int second = subject.getSecond();

    // Digital draw operation
    System.out.println("Drawing digital clock!");
  }
}
```

```java
// "Analog Clock"
public class AnalogClock implements Observer {
  private ClockTimer subject;

  public AnalogClock(ClockTimer subject) {
    this.subject = subject;
    subject.attach(this);
  }

  @Override
  public void update(Subject changedSubject) {
    if (changedSubject.equals(subject))
      draw();
  }

  private void draw() {
    int hour = subject.getHour();
    int minute = subject.getMinute();
    int second = subject.getSecond();

    // Analog draw operation
    System.out.println("Drawing analog clock!");
  }
}
```

# Observer – Client Code

```
// Client code
public class MainApp {
  public static void main(String[] args) {
    // Creating timer
    ClockTimer timer = new ClockTimer();

    // Initializing timer's observers
    AnalogClock analogClock = new AnalogClock(timer);
    DigitalClock digitalClock = new DigitalClock(timer);

    // Changing the time in timer
    timer.tick();
  }
}
```

Output:

**Drawing digital clock!**
**Drawing analog clock!**

# Your Turn

- What is the difference between the State pattern and the Strategy pattern? In which situations each one is best to be used for?
- What is the difference between the State pattern and the Adapter pattern? In which situations each one is best to be used for?
- What is the difference between the Adapter pattern and the Strategy pattern? In which situations each one is best to be used for?
- Give two examples for using the Chain of Responsibility design pattern. Explain the use of the pattern in your examples.
- What is the difference between the Chain of Responsibility pattern and the Strategy pattern? In which situations each one is best to be used for?
- Give two examples for using the Visitor design pattern. Explain the use of the pattern in your examples.

# Your Turn

- Check-out the content of the following sites:
  - https://refactoring.guru/design-patterns (check all patterns applicable)
  - https://www.javatpoint.com/design-patterns-in-java (check all patterns applicable)
  - https://howtodoinjava.com/design-patterns/
  - https://en.wikipedia.org/wiki/Adapter_pattern
  - https://en.wikipedia.org/wiki/State_pattern
  - https://en.wikipedia.org/wiki/Strategy_pattern
  - https://en.wikipedia.org/wiki/Visitor_pattern
  - https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern
  - https://en.wikipedia.org/wiki/Observer_pattern
  - https://stackoverflow.com/questions/15563005/observer-pattern-vs-mvc