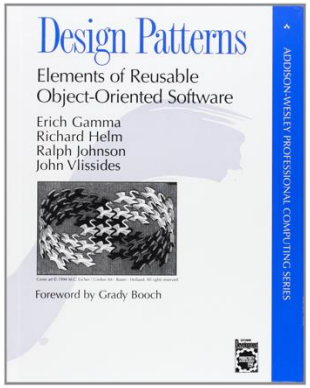
Design Patterns – Behaviour

UA.DETI.PDS
José Luis Oliveira

Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.
- Effective Java, 2nd Ed., Joshua Bloch



- *Design Patterns Explained Simply* *(sourcemaking.com)*
- https://refactoring.guru/design-patterns

Behavioral patterns

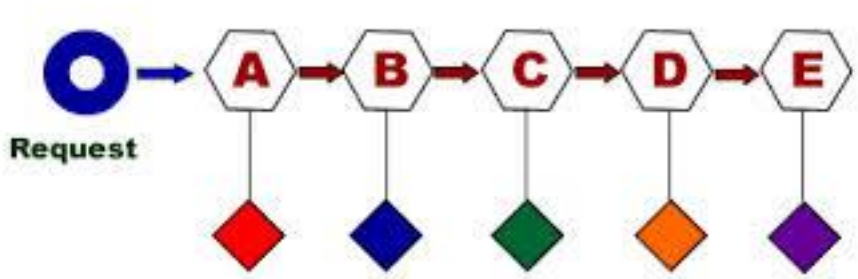
- They identify common communication patterns between objects and realize these patterns.

Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

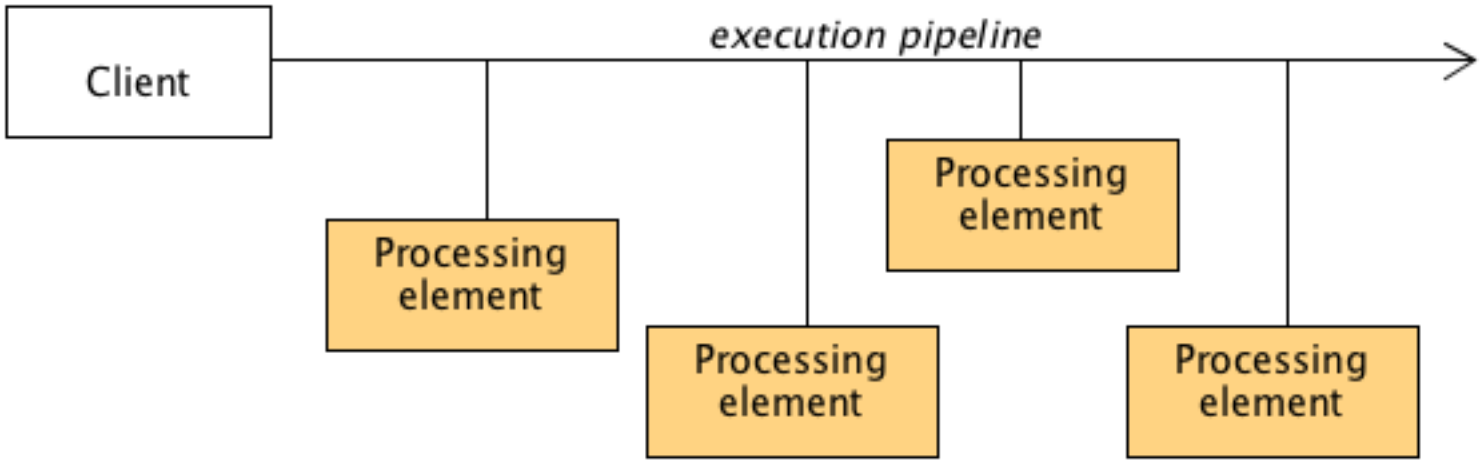
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor



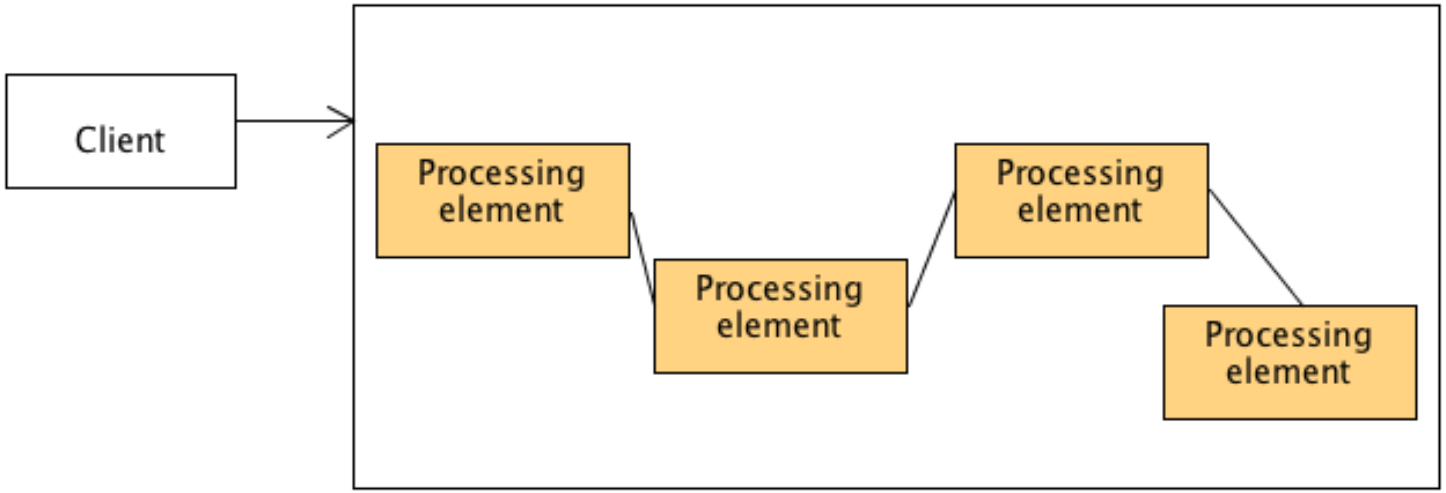
The problem

- There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled.
  - Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

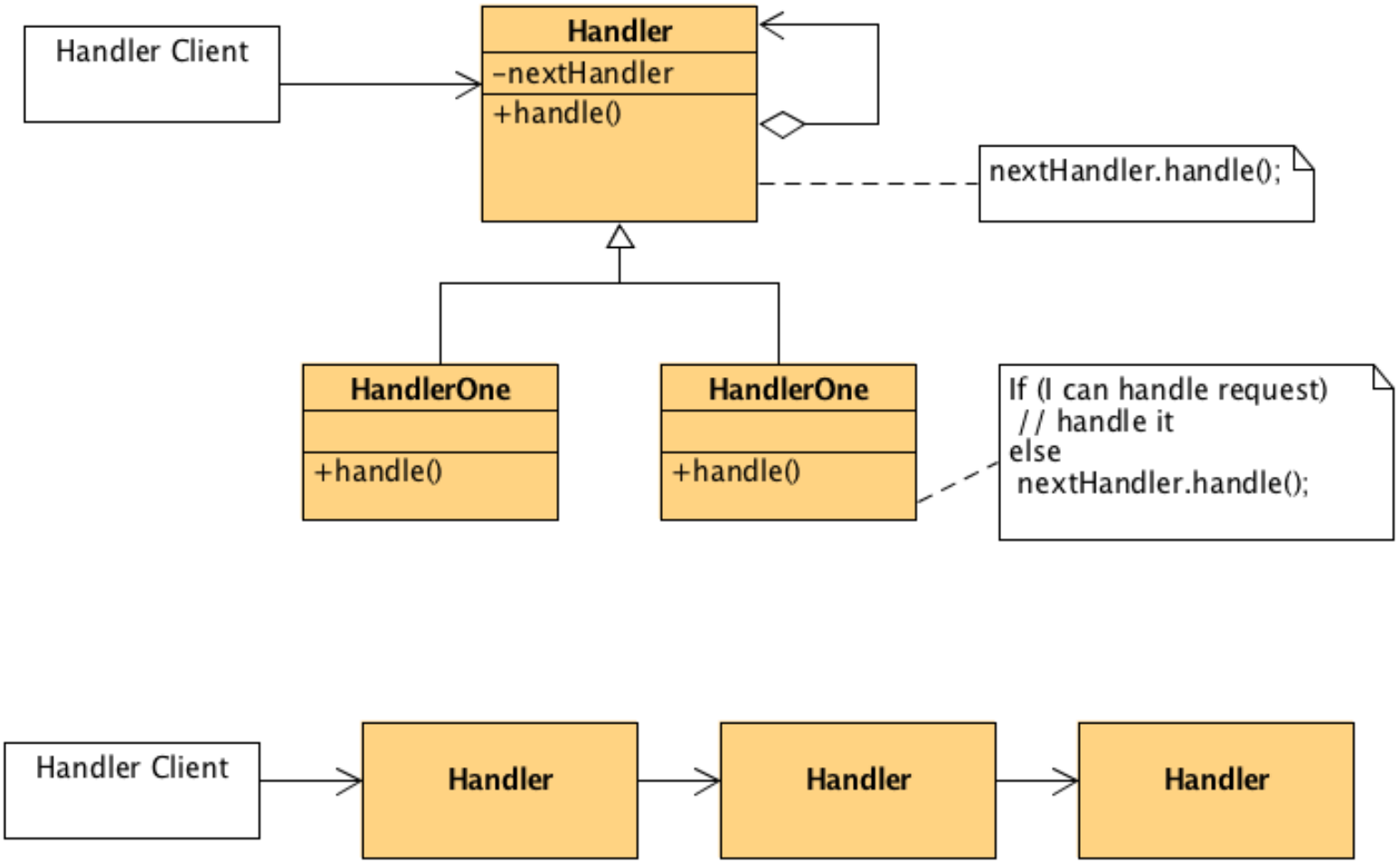


Intent

- Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.



Structure


Handler Client
Handler
-nextHandler
+handle()
nextHandler.handle();
HandlerOne
+handle()
HandlerOne
+handle()
If (I can handle request)
// handle it
else
nextHandler.handle();
Handler Client
Handler
Handler
Handler

Example

```
abstract class Parser {
	private Parser successor = null;

	public void parse(String fileName) {
		if (successor != null)
			successor.parse(fileName);
		else
			System.out.println("No parser for the file: " + fileName);
	}

	protected boolean canHandleFile(String fileName, String format) {
		return (fileName == null) || (fileName.endsWith(format));
	}

	public Parser setSuccessor(Parser successor) {
		this.successor = successor;
		return this;
	}
}
```

Example

```
class JsonParser extends Parser {
	@Override
	public void parse(String fileName) {
		if (canHandleFile(fileName, ".json"))
			System.out.println("A JSON parser for: " + fileName);
		else
			super.parse(fileName);
	}
}

class CsvParser extends Parser {
	@Override
	public void parse(String fileName) {
		if (canHandleFile(fileName, ".csv"))
			System.out.println("A CSV parser for: " + fileName);
		else
			super.parse(fileName);
	}
}
```

Example

```
class TextParser extends Parser {

	@Override
	public void parse(String fileName) {
		if (canHandleFile(fileName, ".txt")) {
			System.out.println("A text parser for: " + fileName);
		} else {
			super.parse(fileName);
		}

	}

}
```

Example

```
public class ChainOfResponsibilityDemo {

    public static void main(String[] args) {
        List<String> fileList = new ArrayList<>();
        fileList.add("someFile.txt");
        fileList.add("otherFile.json");
        fileList.add("csvFile.csv");
        fileList.add("somethingelse.doc");
        Parser textParser =
                new CsvParser().setSuccessor(
                        new TextParser().setSuccessor(
                                new JsonParser()));
        for (String fileName : fileList) {
            textParser.parse(fileName);
        }
    }
}
```

```
A text parser for: someFile.txt
A JSON parser for: otherFile.json
A CSV parser for: csvFile.csv
No parser for the file: somethingelse.doc
```

Check list

- The base class maintains a "next" pointer.
- Each derived class implements its contribution for handling the request.
- If the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer.
- The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
- The client "launches and leaves" each request with the root of the chain.
- Recursive delegation produces the illusion of magic.

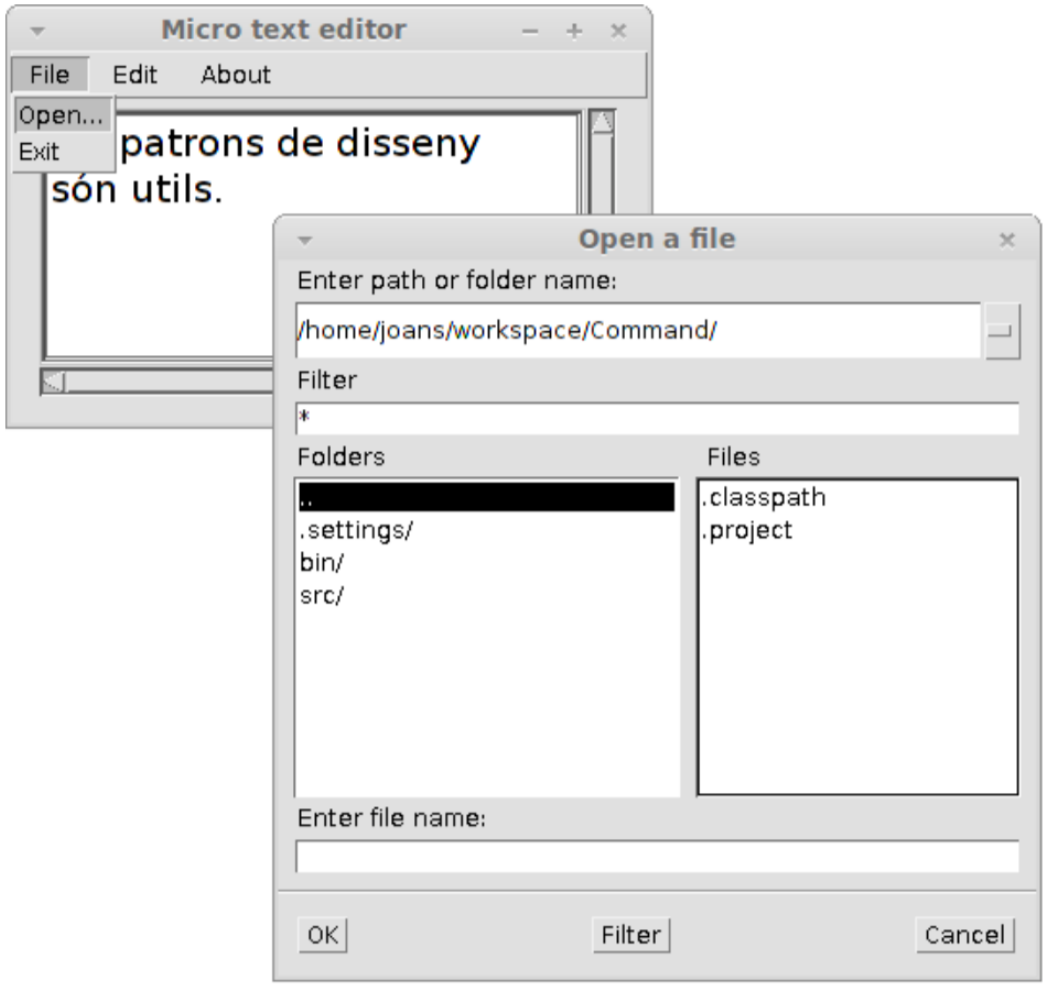
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Motivation

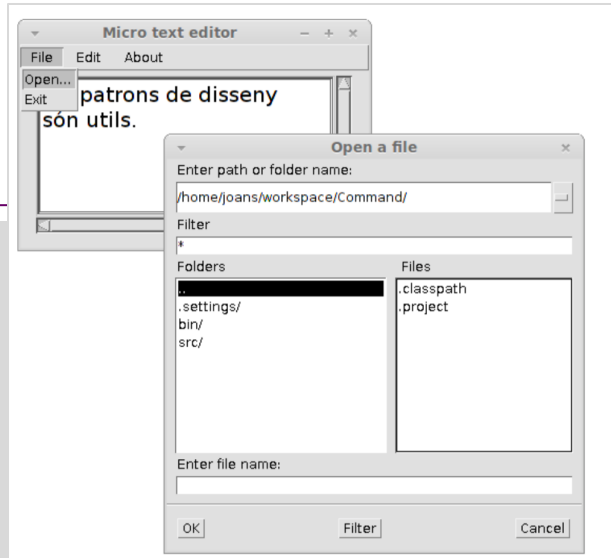
- **Intent**
  - **Encapsulate a request as an object**, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - Promote "invocation of a method on an object" to full object status
  - An object-oriented callback
- **Problem**
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Motivating example



Example

```
menuFile = new Menu("File", true);
menuItemOpen = new MenuItem("Open...");
menuItemExit = new MenuItem("Exit");
menuFile.add(menuItemOpen);
menuFile.add(menuItemExit);
```



- AWT is a Java toolkit to build user interfaces
  - Has classes **Frame** (window), **Panel** (container of other controls), **TextArea** (holds text), **Menu** with options **MenuItem**

- **MenuItem** objects must issue a request to some object in response to user input
  - But it can not know nothing about the operation being requested or the receiver of the request

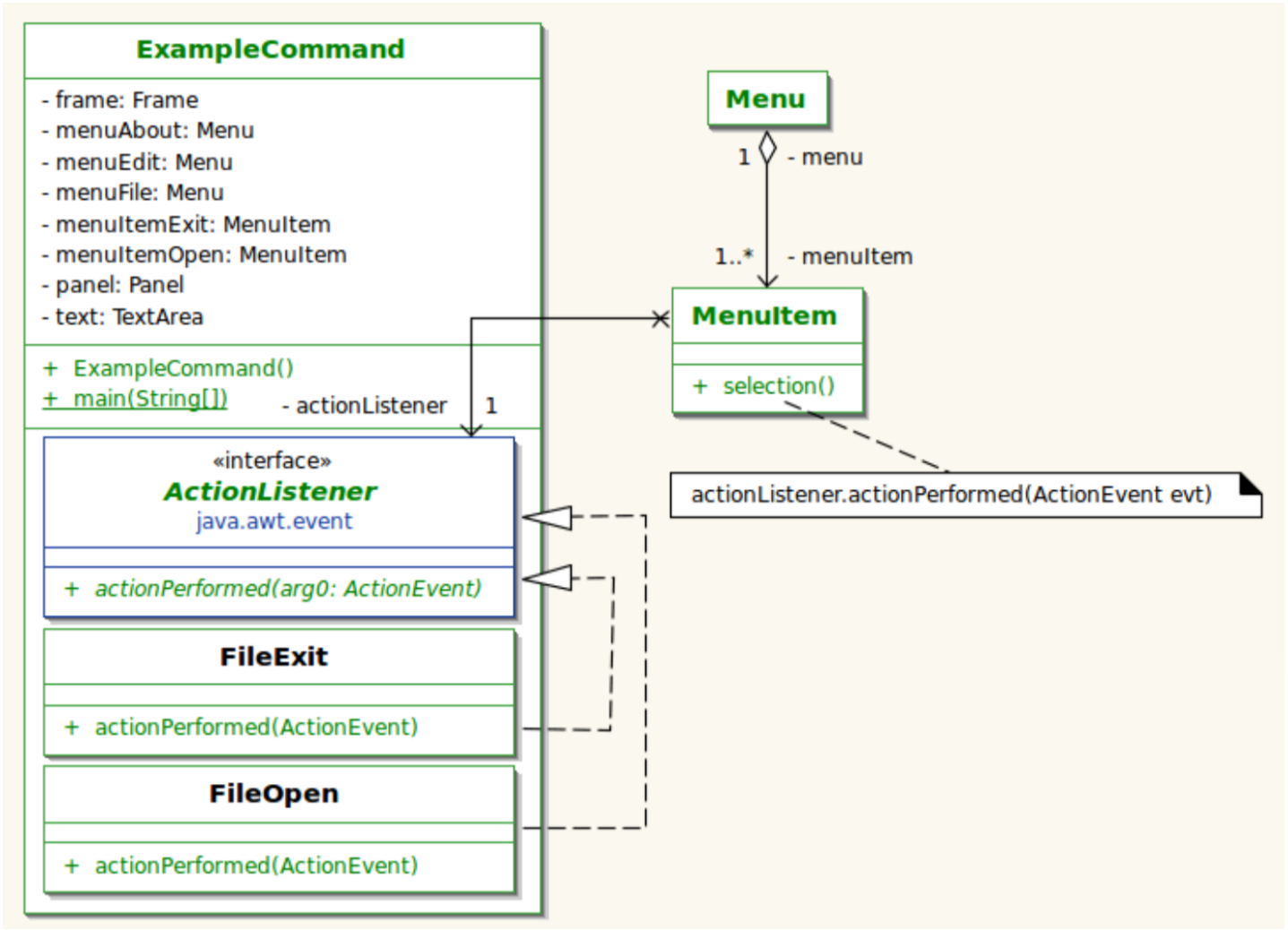
Solution

```
menuFile = new Menu("File", true);
menuItemOpen = new MenuItem("Open...");
menuItemExit = new MenuItem("Exit");
menuFile.add(menuItemOpen);
menuFile.add(menuItemExit);
menuItemOpen.addActionListener(new FileOpen());
menuItemExit.addActionListener(new FileExit());
//...
class FileOpen implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      FileDialog fDlg = new FileDialog(frame,"Open", FileDialog.LOAD);
      fDlg.show();
   }
}
class FileExit implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.exit(0);
   }
}
```

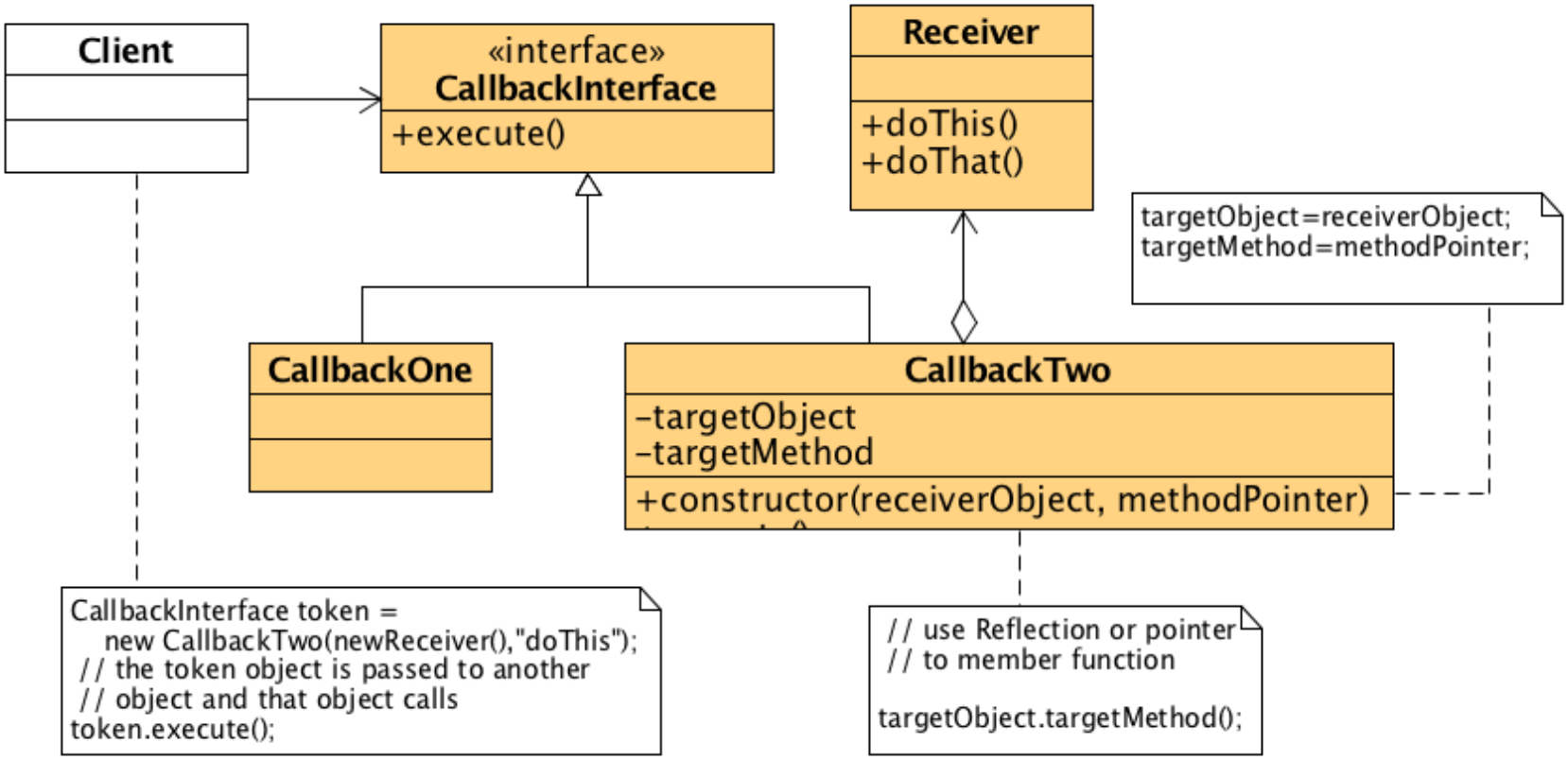
Solution

- The key is the **ActionListener** interface:
  - it declares the method *actionPerformed(ActionEvent evt)*
- Each concrete *ActionListener* class implements *actionPerformed* to answer user input
- *MenuItem* class knows it must call *actionPerformed* from its *ActionListener* when the menu item is selected
- **ActionListener interface is the Command**

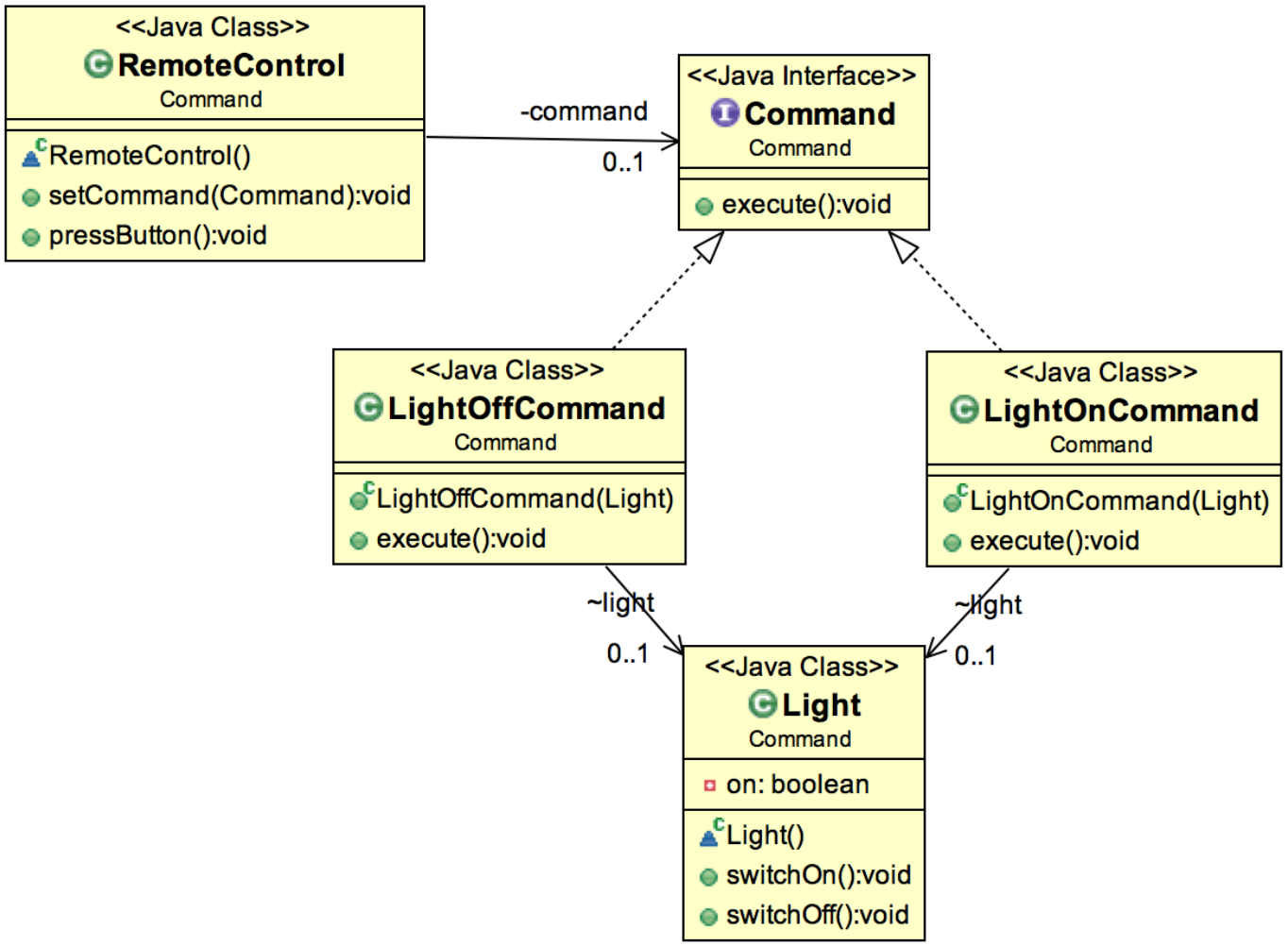
Solution


ExampleCommand
- frame: Frame
- menuAbout: Menu
- menuEdit: Menu
- menuFile: Menu
- menuItemExit: MenuItem
- menuItemOpen: MenuItem
- panel: Panel
- text: TextArea
+ ExampleCommand()
+ main(String[])
- actionListener
1
«interface»
ActionListener
java.awt.event
+ actionPerformed(arg0: ActionEvent)
FileExit
+ actionPerformed(ActionEvent)
FileOpen
+ actionPerformed(ActionEvent)
Menu
1
- menu
1..*
- menuItem
MenuItem
+ selection()
actionListener.actionPerformed(ActionEvent evt)

Structure

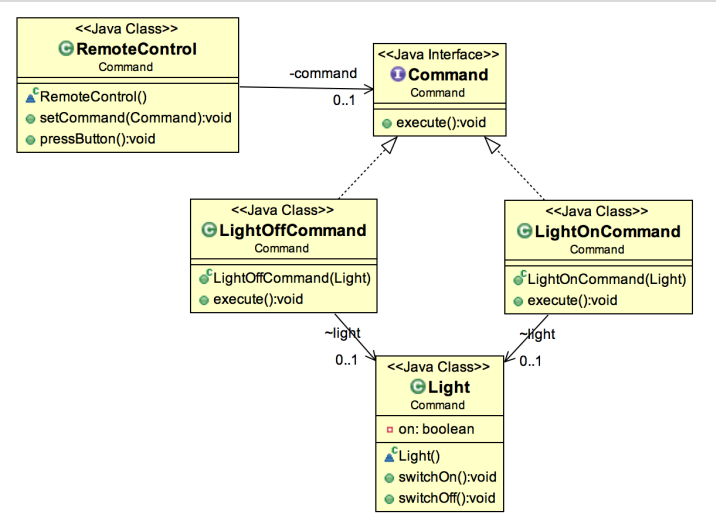

Client
«interface»
CallbackInterface
+execute()
Receiver
+doThis()
+doThat()
targetObject=receiverObject;
targetMethod=methodPointer;
CallbackOne
CallbackTwo
-targetObject
-targetMethod
+constructor(receiverObject, methodPointer)
CallbackInterface token =
new CallbackTwo(newReceiver(),"doThis");
// the token object is passed to another
// object and that object calls
token.execute();
// use Reflection or pointer
// to member function
targetObject.targetMethod();

Example


<<Java Class>>
RemoteControl
Command
RemoteControl()
setCommand(Command):void
pressButton():void
-command
0..1
<<Java Interface>>
Command
Command
execute():void
<<Java Class>>
LightOffCommand
Command
LightOffCommand(Light)
execute():void
<<Java Class>>
LightOnCommand
Command
LightOnCommand(Light)
execute():void
~light
0..1
~light
0..1
<<Java Class>>
Light
Command
on: boolean
Light()
switchOn():void
switchOff():void

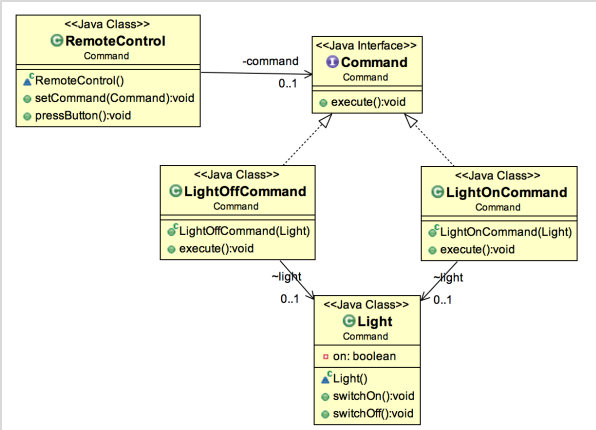
Example – the solution

```
// Receiver
class Light {
        private boolean on;
        public void switchOn()  { on = true; }
        public void switchOff() { on = false; }
}


// Invoker
class RemoteControl {
        private Command command;
        public void setCommand(Command command) {
                this.command = command;
        }
        public void pressButton() {
                command.execute();
        }
}
```



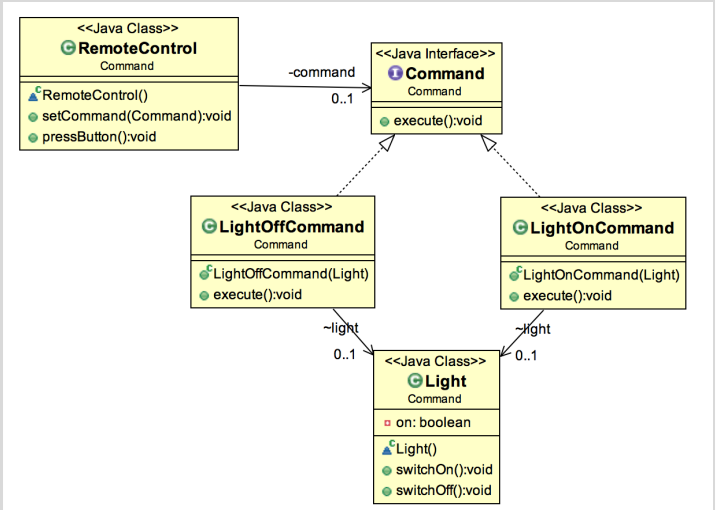
Example – the solution

```
//Command
interface Command {
        public void execute();
}

// Concrete Command
class LightOnCommand implements Command {
        // reference to the light
        Light light;
        public LightOnCommand(Light light)                { this.light = light;    }
        public void execute() { light.switchOn(); }
}

// Concrete Command
class LightOffCommand implements Command {
        // reference to the light
        Light light;
        public LightOffCommand(Light light)               { this.light = light;    }
        public void execute() { light.switchOff();        }
}
```



Example – the client

```
public class Client {
  public static void main(String[] args)  {
     RemoteControl control = new RemoteControl();

     Light light = new Light();

     Command lightsOn = new LightOnCommand(light);
     Command lightsOff = new LightOffCommand(light);

     //switch on
     control.setCommand(lightsOn);
     control.pressButton();

     //switch off
     control.setCommand(lightsOff);
     control.pressButton();

  }
}
```



Example 2 – reflection

```
public class CommandReflect {
  private int state;
  public CommandReflect( int in ) {   state = in;   }
  public int addOne( Integer one ) {   return state + one;   }
  public int addTwo( Integer one, Integer two ) {   return state + one + two;   }
  static public class Command {
    private Object   receiver;            // the "encapsulated" object
    private Method   action;              // the "pre-registered" request
    private Object[] args;                // the "pre-registered" arg list
    public Command( Object obj, String methodName, Object[] arguments ) {
      receiver = obj;
      args = arguments;
      Class<?> cls = obj.getClass();          // get the object's "Class"
      Class<?>[] argTypes = new Class[args.length];
      for (int i=0; i < args.length; i++)   // get the "Class" for each
        argTypes[i] = args[i].getClass();  //    supplied argument
            try {     action = cls.getMethod( methodName, argTypes );     }
      catch( NoSuchMethodException e ) { System.out.println( e ); }
    }
    public Object execute() {
      try {    return action.invoke( receiver, args );    }
      catch( IllegalAccessException e    ) { System.out.println( e ); }
      catch( InvocationTargetException e ) { System.out.println( e ); }
      return null;
  } }
```

Example 2 – reflection

```
public static void main( String[] args ) {
  CommandReflect[] objs = { new CommandReflect(1), new CommandReflect(2) };
  Command[] cmds = {
    new Command( objs[0], "addOne", new Integer[] { 3 } ),
    new Command( objs[1], "addTwo", new Integer[] { 4, 5 } ) };
  System.out.print( "\nReflection results:  " );
  for (Command cmd: cmds)
     System.out.print( cmd.execute() + " " );
  System.out.println();
     }
```

```
Reflection results:  4 11
```

Example 2 – using functional interface

```
public class CommandFactory {
    private Map<String, Command> commands;
    private CommandFactory()
        { commands = new HashMap<>(); }
    public void addCommand(String name, Command command)
        { commands.put(name, command); }
    public void executeCommand(String name)
        { if (commands.containsKey(name)) commands.get(name).execute(); }
    public String listCommands() {
        return "Enabled commands: "
            + commands.keySet().stream().collect(Collectors.joining(", "));
    }
    public static CommandFactory init() {  /* Factory pattern */
        final CommandFactory cf = new CommandFactory();
        // commands are added here using lambdas.
        // It is also possible to dynamically add commands without editing the code.
        cf.addCommand("Light on", () -> System.out.println("Light turned on"));
        cf.addCommand("Light off", () -> System.out.println("Light turned off"));
        cf.addCommand("Sound on", () -> System.out.println("Sound turned on"));
        cf.addCommand("Sound off", () -> System.out.println("Sound turned off"));
        return cf;
    }
}
```

```
@FunctionalInterface
public interface Command {
    public void execute();
}
```

Example 2 – using functional interface

```
public class Main {
	public static void main(final String[] arguments) {

		CommandFactory cf = CommandFactory.init();
		System.out.println(cf.listCommands());
		cf.executeCommand("Light on");
		cf.executeCommand("Sound on");
		cf.executeCommand("Light off");
	}
}
```

```
Enabled commands: Light on, Sound on, Light off, Sound off
Light turned on
Sound turned on
Light turned off
```

When to use?

- We need callback functionality
- Requests need to be handled at variant times or in variant orders
- The invoker should be decoupled from the object handling the invocation.

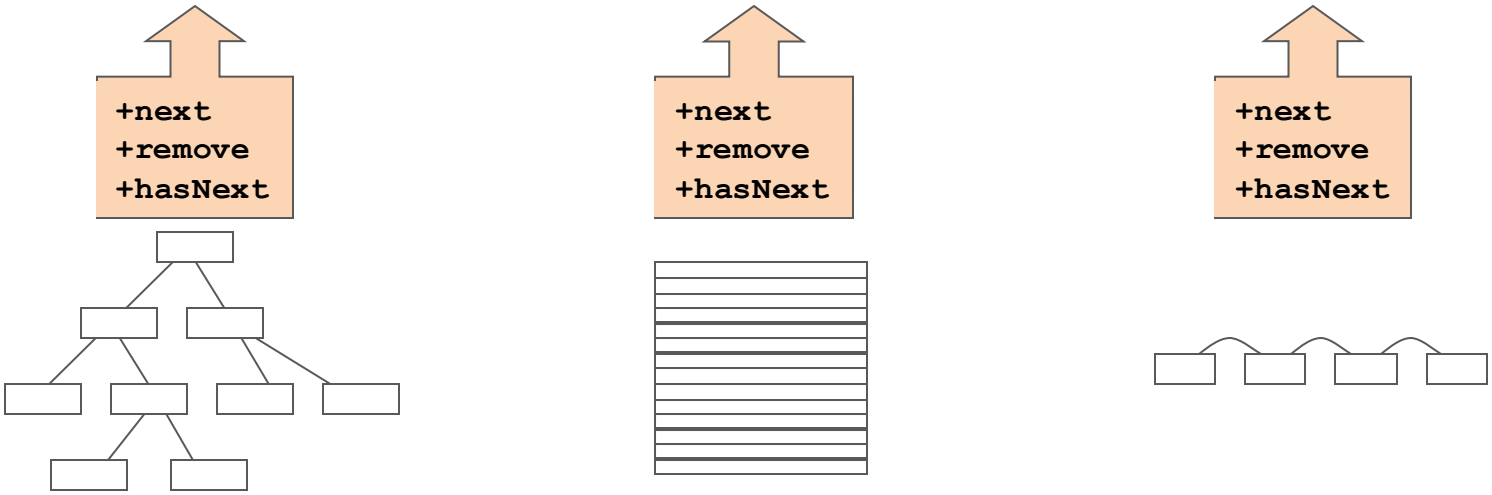
- Well know application:
  - Swing ActionListeners
  - Java Reflection

Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Iterator

- **Problem**



- **Intent**
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Example – common code

```
// For a set or list
for (Iterator it=collection.iterator(); it.hasNext(); ) {
        Object element = it.next();
}
// For keys of a map
for (Iterator it=map.keySet().iterator(); it.hasNext(); ) {
        Object key = it.next();
}
// For values of a map
for (Iterator it=map.values().iterator(); it.hasNext(); ) {
        Object value = it.next();
}
// For both the keys and values of a map
for (Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
        Map.Entry entry = (Map.Entry)it.next();
        Object key = entry.getKey();
        Object value = entry.getValue();
}
```

Iterators in Java

- Iterators are widely used on Java Collections
- We can get an iterator (*java.util.Iterator*) from the *Collection.iterator()* method.

- java.util.Iterator interface

```
public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Iterators in Java

- To implement an iterator in an aggregate it is necessary to use delegation:
  1. Create an private iterator in the aggregate class and a public method (*iterator()*) to return an iterator.
  2. Create the private iterator method accessing data from the collection.

```
VectorGeneric<Pessoa> vp = new VectorGeneric<>();
for (int i=0; i<10; i++)
      vp.addElem(new Pessoa("Some One "+i, 1000+i, Data.today()));

Iterator<Pessoa> vec = vp.iterator();
while ( vec.hasNext() )
      System.out.println( vec.next() );
```

Example

```
public class VectorGeneric<T> {
	private T[] vec;
	private int nElem;
	private final static int ALLOC = 50;
	private int dimVec = ALLOC;

	@SuppressWarnings("unchecked")
	public VectorGeneric() {
		vec = (T[]) new Object[dimVec];
		nElem = 0;
	}

	public boolean addElem(T elem) {
		if (elem == null)
			return false;
		ensureSpace();
		vec[nElem++] = elem;
		return true;
	}
```

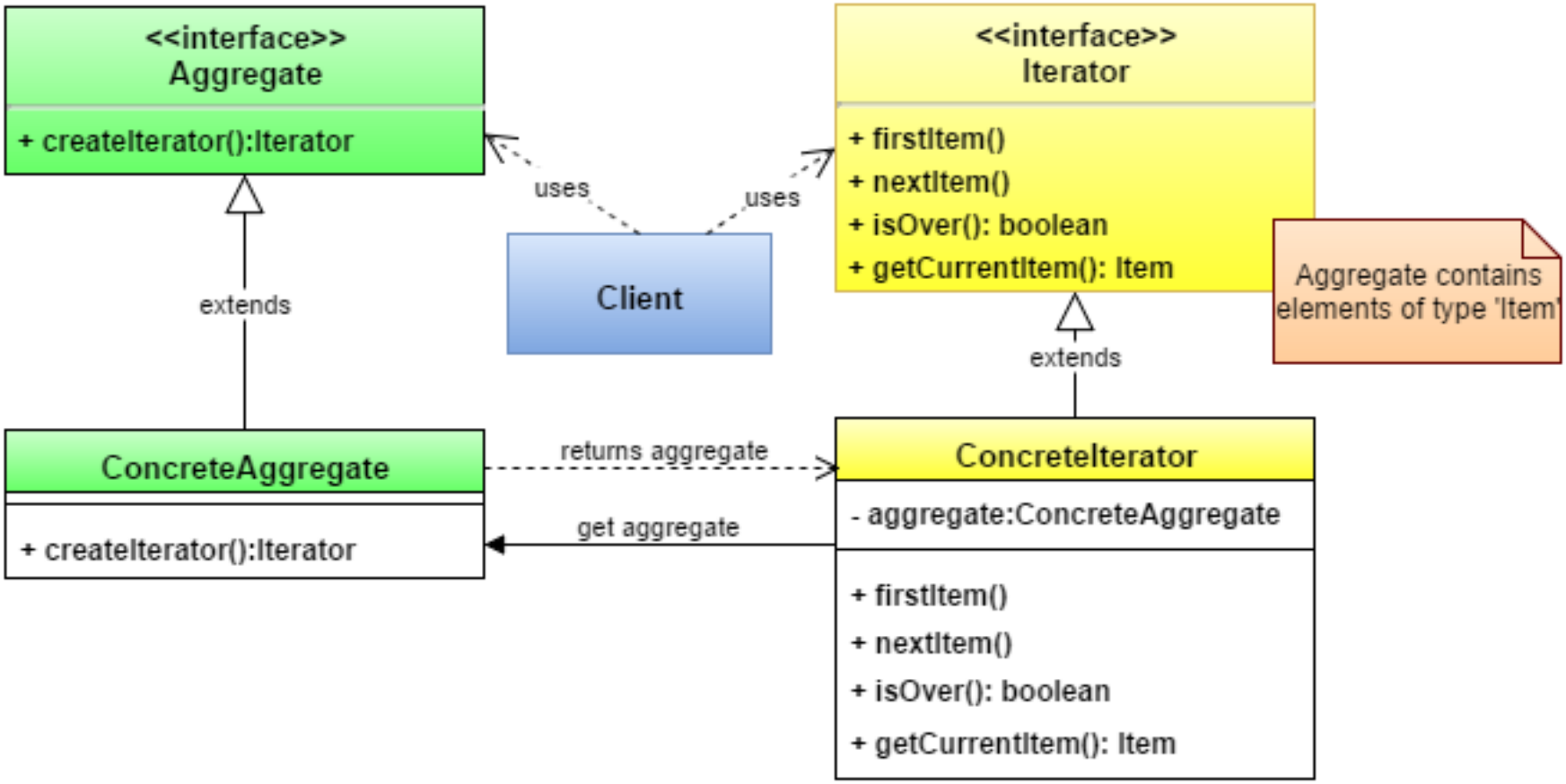
Example

```
private void ensureSpace() {
    if (nElem>=dimVec) {
        dimVec += ALLOC;
        @SuppressWarnings("unchecked")
        T[] newArray = (T[]) new Object[dimVec];
        System.arraycopy(vec, 0, newArray, 0, nElem );
        vec = newArray;
    }
}
public boolean removeElem(T elem) {
    for (int i = 0; i < nElem; i++) {
        if (vec[i].equals(elem)) {
            if (nElem-i-1 > 0) // not last element
                System.arraycopy(vec, i+1, vec, i, nElem-i-1 );
            vec[--nElem] = null; // libertar último objecto para o GC
            return true;
        }
    }
    return false;
}
```

Example

```
  public Iterator<T> iterator() {
        return (this).new VectorIterator<T>();
  }
  private class VectorIterator<K>  implements Iterator<K> {
        private int indice;
        VectorIterator() {
              indice = 0;
        }
        public boolean hasNext() {
              return (indice < nElem);
        }
        public K next() {
           if (hasNext())
              return (K)VectorGeneric.this.vec[indice++];
           throw new NoSuchElementException("only " + nElem + " elements");
        }
        public void remove() {  // default since Java 8
           throw new UnsupportedOperationException("Operacao nao suportada!");
}}    }
```

Structure

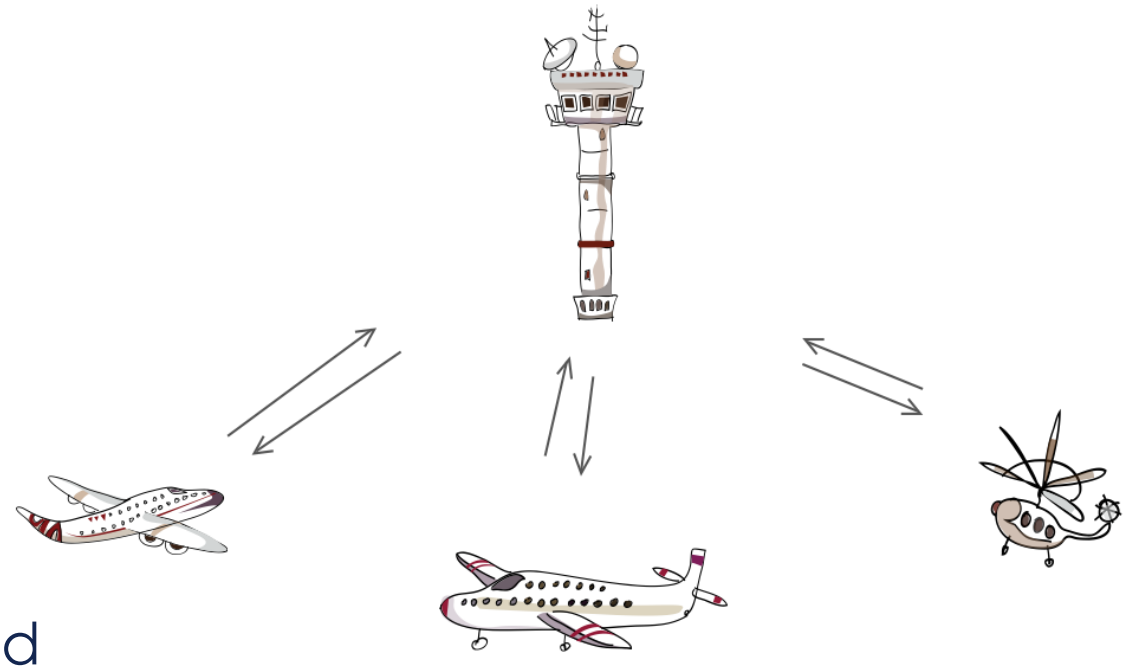

<<interface>>
Aggregate
+ createIterator():Iterator
<<interface>>
Iterator
+ firstItem()
+ nextItem()
+ isOver(): boolean
+ getCurrentItem(): Item
uses
uses
Client
extends
extends
Aggregate contains elements of type 'Item'
ConcreteAggregate
+ createIterator():Iterator
returns aggregate
get aggregate
ConcreteIterator
- aggregate:ConcreteAggregate
+ firstItem()
+ nextItem()
+ isOver(): boolean
+ getCurrentItem(): Item

Check list

- Add an *iterator()* method to the "collection" class and grant the "iterator" class privileged access.
- Design an "iterator" class that can encapsulate the traversal of the "collection" class.
- Clients ask the collection object to create an iterator object.
- Clients use the *hasNext()*, *next()* methods to access the elements of the collection class.

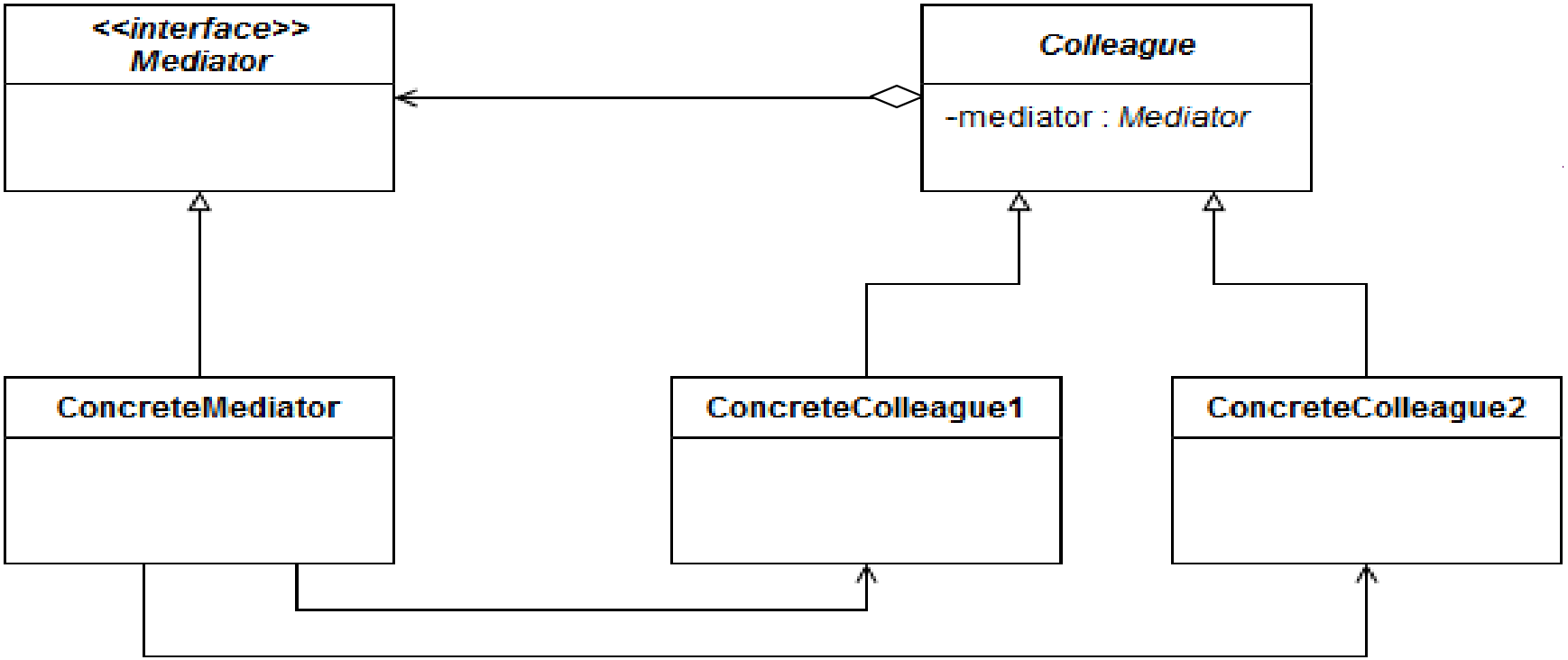
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor



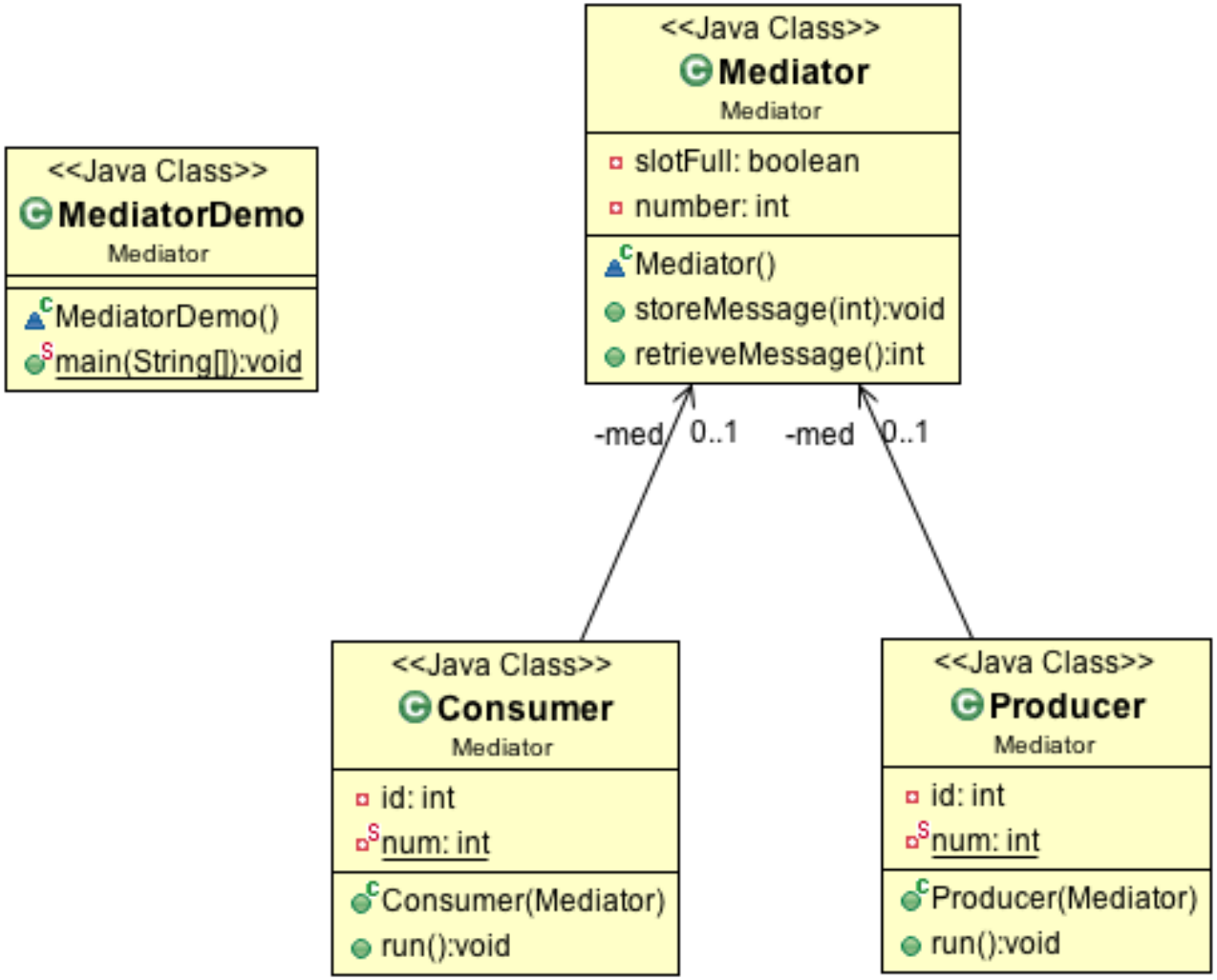
Motivation

- **Problem**
  - We want to design reusable components, but dependencies between the potentially reusable pieces demonstrate the "spaghetti code" phenomenon - trying to scoop a single serving result in an "all or nothing clump".
- **Intent**
  - Define an object that encapsulates how a set of objects interact.
  - Design an intermediary to decouple many peers.
    - Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
    - Promote the many-to-many relationships between interacting peers to "full object status".



- Mediator defines the interface the Colleague objects use to communicate
- Colleague defines the abstract class holding a reference to the Mediator
- ConcreteMediator encapsulates the interaction logic between Colleague objects
- ConcreteColleague1 and ConcreteColleague2 communicate only through the Mediator

Example


<<Java Class>>
MediatorDemo
Mediator
MediatorDemo()
main(String[]):void
<<Java Class>>
Mediator
Mediator
slotFull: boolean
number: int
Mediator()
storeMessage(int):void
retrieveMessage():int
-med
0..1
-med
0..1
<<Java Class>>
Consumer
Mediator
id: int
num: int
Consumer(Mediator)
run():void
<<Java Class>>
Producer
Mediator
id: int
num: int
Producer(Mediator)
run():void

Example

```
class Mediator {
        private boolean slotFull = false;
        private int number;

        public synchronized void storeMessage(int num) {
                while (slotFull == true) {
                        try { wait();
                        } catch (InterruptedException e) {
                                // ...
                        }
                }
                slotFull = true;
                number = num;
                notifyAll();
        }

        public synchronized int retrieveMessage() {
                // ...
        }
}
```

Example

```
class Producer extends Thread {
	// 2. Producers are coupled only to the Mediator
	private Mediator med;
	private int id;
	private static int num = 1;

	public Producer(Mediator m) {
		med = m;
		id = num++;
	}

	public void run() {
		int num;
		while (true) {
			med.storeMessage(num = (int) (Math.random() * 100));
			System.out.println("p" + id + "-" + num + "  ");
		}
	}
}
```

Example

```
class Consumer extends Thread {
	// 3. Consumers are coupled only to the Mediator
	private Mediator med;
	private int id;
	private static int num = 1;

	public Consumer(Mediator m) {
		med = m;
		id = num++;
	}

	public void run() {
		while (true) {
			System.out.println("\tc"+ id + "-" + med.retrieveMessage()+" ");
		}
	}
}
```

Example

```
class MediatorDemo {
        public static void main(String[] args) {
                Mediator mb = new Mediator();
                new Producer(mb).start();
                new Producer(mb).start();
                new Consumer(mb).start();
                new Consumer(mb).start();
                new Consumer(mb).start();
                new Consumer(mb).start();
        }
}
```

```
c1-94
p1-94
p1-50
p2-86
                c4-50
                c2-86
p1-82
                c4-82
p1-55
                c4-55
p2-70
                c1-70
                c1-23
p2-23
p2-52
                c2-52
```

Example 2

```
// Mediator Interface
interface ChatMediator {
    void sendMessage(String message, User user);
    void addUser(User user);
}

// Concrete Mediator
class ChatRoom implements ChatMediator {
    private List<User> users;

    public ChatRoom() {       this.users = new ArrayList<>();    }

    public void sendMessage(String message, User user) {
        for (User u : users) {
            // Message should not be sent to the user who sent it
            if (!u.equals(user)) {
                u.receive(message);
            }
        }
    }

          public void addUser(User user) {       users.add(user);    }
}
```

Example 2

```
// Colleague Interface
abstract class User {
    protected ChatMediator mediator;
    protected String name;

    public User(ChatMediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }
    public abstract void send(String message);
    public abstract void receive(String message);
}

// Concrete Colleagues
class ChatUser extends User {
    public ChatUser(ChatMediator mediator, String name) {
        super(mediator, name);
    }
    public void send(String message) {
        System.out.println(name + ": Sending message: " + message);
        mediator.sendMessage(message, this);
    }
    public void receive(String message) {
        System.out.println(name + ": Received message: " + message);
    }
}
```

Example 2

```
// Client
public class Client {
    public static void main(String[] args) {
        ChatMediator chatRoom = new ChatRoom();

        User user1 = new ChatUser(chatRoom, "Alice");
        User user2 = new ChatUser(chatRoom, "Bob");
        User user3 = new ChatUser(chatRoom, "Charlie");

        chatRoom.addUser(user1);
        chatRoom.addUser(user2);
        chatRoom.addUser(user3);

        user1.send("Hello, everyone!");
        user2.send("Hi, Alice!");
        user3.send("Good morning, all!");
    }
}
```

```
Alice: Sending message: Hello, everyone!
Bob: Received message: Hello, everyone!
Charlie: Received message: Hello, everyone!
Bob: Sending message: Hi, Alice!
Alice: Received message: Hi, Alice!
Charlie: Received message: Hi, Alice!
Charlie: Sending message: Good morning, all!
Alice: Received message: Good morning, all!
Bob: Received message: Good morning, all!
```

Examples in Java libraries

- java.util.Timer
  - all scheduleXXX() method
- java.util.concurrent.Executor
  - execute()
- java.util.concurrent.ExecutorService
  - invokeXXX() and submit() methods
- java.util.concurrent.ScheduledExecutorService
  - all scheduleXXX() methods
- java.lang.reflect.Method
  - invoke()

Check list

- Identify a collection of interacting objects that would benefit from mutual decoupling.
- Encapsulate those interactions in the abstraction of a new class.
- Create an instance of that new class and rework all "peer" objects to interact with the Mediator only.
- Balance the principle of decoupling with the principle of distributing responsibility evenly.
- Be careful not to create a "controller" or "god" object.

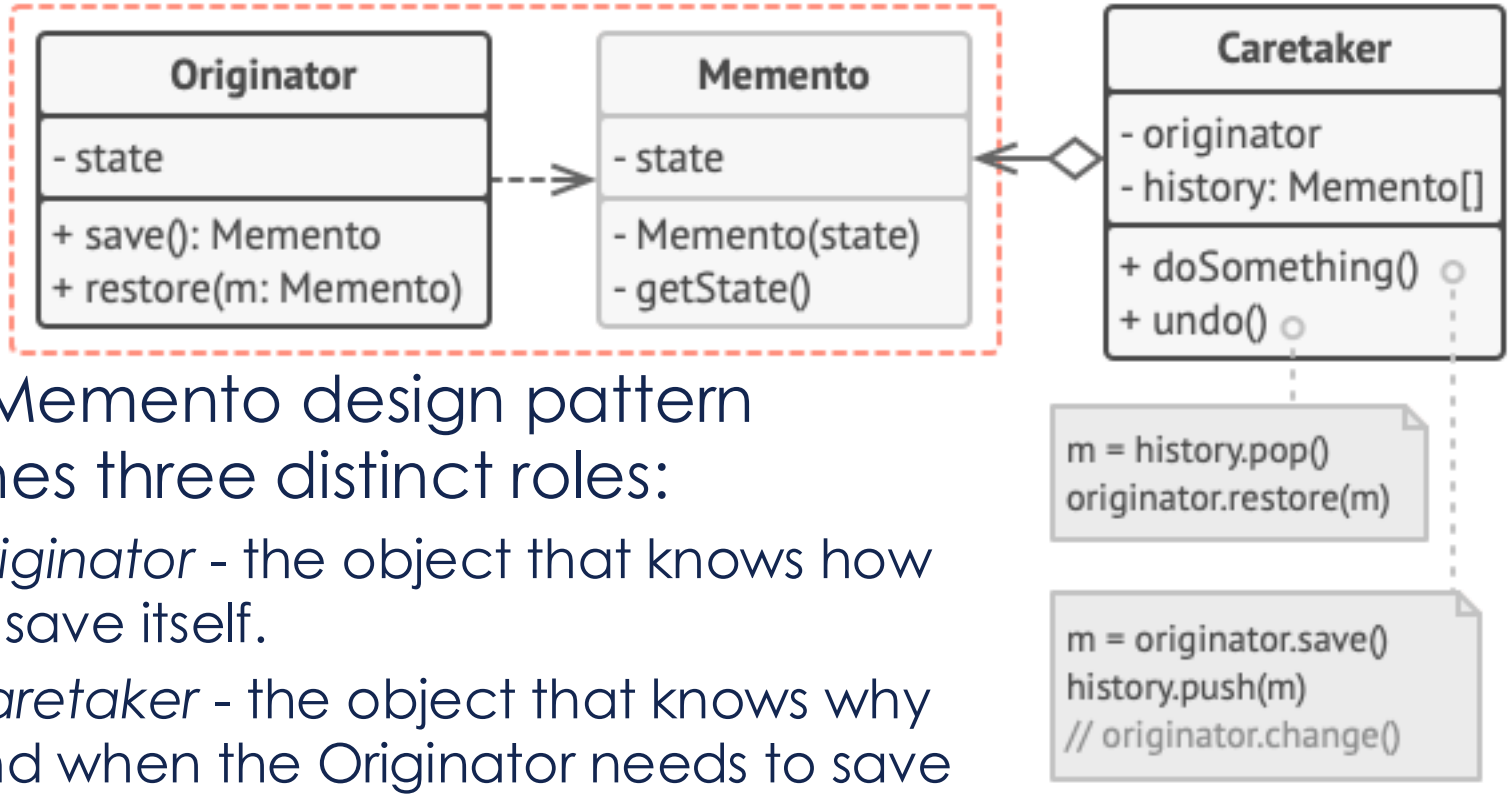
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Motivation

- **Problem**
  - Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).
- **Intent**
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
  - A magic cookie that encapsulates a "check point" capability.
  - Promote undo or rollback to full object status.

Structure



- The Memento design pattern defines three distinct roles:
  - *Originator* - the object that knows how to save itself.
  - *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
  - *Memento* - the snapshot that is written and read by the Originator, and shepherded by the Caretaker.

Example

```
class Memento {
  private String state;
  public Memento(String stateToSave) { state = stateToSave; }
  public String getSavedState() { return state; }
}

class Originator {
  private String state; // simple example
  public void set(String state) {
     this.state = state;
  }
  public Memento saveToMemento() {
     return new Memento(state);
  }
  public void restoreFromMemento(Memento m) {
     state = m.getSavedState();
  }
  @Override public String toString() { return state; }
}
```

Example

```
class Caretaker {
  private Stack<Memento> savedStates = new Stack<Memento>();

  public void addMemento(Memento m) {
            savedStates.push(m);
      }
  public boolean hasMemento() {
            return !savedStates.isEmpty();
      }
  public Memento getMemento() {
            return savedStates.pop();
      }
}
```

Example

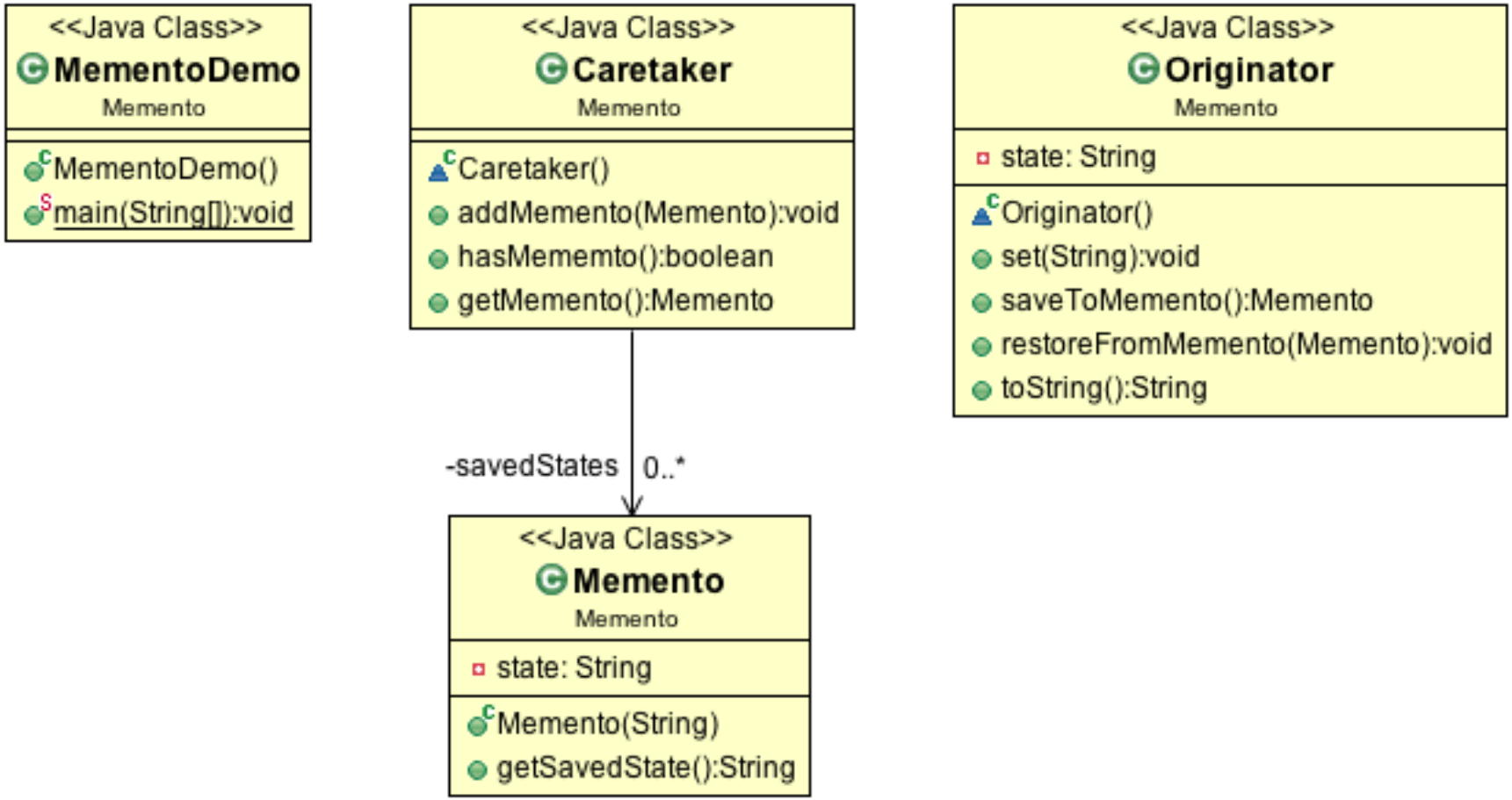
```
public class MementoDemo {
  public static void main(String[] args) {
    Caretaker caretaker = new Caretaker();

    Originator originator = new Originator();
    for (int i= 1; i<=5; i++) {
      originator.set("State " + i);
      System.out.println("Originator: state set to "+ originator);
      caretaker.addMemento( originator.saveToMemento() );
      System.out.println("Memento saved");
    }

    while (caretaker.hasMemento()) {
      originator.restoreFromMemento( caretaker.getMemento() );
      System.out.println("Originator: after restore: "+originator);
    }
  }
}
```

Example

```
public class MementoDemo {
  public static void main(String[] args) {
    Caretaker caretaker = new Caretaker(

    Originator originator = new Originato
    for (int i= 1; i<=5; i++) {
      originator.set("State " + i);
      System.out.println("Originator: sta
      caretaker.addMemento( originato
      System.out.println("Memento sav
    }

    while (caretaker.hasMememto()) {
      originator.restoreFromMemento(
      System.out.println("Originator: aft
    }
  }
}
```

```
Originator: state set to State 1
Memento saved
Originator: state set to State 2
Memento saved
Originator: state set to State 3
Memento saved
Originator: state set to State 4
Memento saved
Originator: state set to State 5
Memento saved
Originator: after restore: State 5
Originator: after restore: State 4
Originator: after restore: State 3
Originator: after restore: State 2
Originator: after restore: State 1
```

Example


<<Java Class>>
MementoDemo
Memento
MementoDemo()
main(String[]):void
<<Java Class>>
Caretaker
Memento
Caretaker()
addMemento(Memento):void
hasMememto():boolean
getMemento():Memento
-savedStates
0..*
<<Java Class>>
Originator
Memento
state: String
Originator()
set(String):void
saveToMemento():Memento
restoreFromMemento(Memento):void
toString():String
<<Java Class>>
Memento
Memento
state: String
Memento(String)
getSavedState():String

Examples in Java libraries

- Serializable
  - Any *java.io.Serializable* implementation can simulate the Memento.

- It allows making snapshots of an object's state and restoring it in future.

Check list

- Identify the roles of “caretaker” and “originator”.
- Create a Memento class
- Caretaker knows when to "check point" the originator.
- Originator creates a Memento and copies its state to that Memento.
- Caretaker holds on to (but cannot peek into) the Memento.
- Caretaker knows when to "roll back" the originator.
- Originator reinstates itself using the saved state in the Memento.

Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Motivation

- "Null sucks." - Doug Lea
- "I call it my billion-dollar mistake." - Sir C. A. R. Hoare, on his invention of the null reference

```
Request request = someObj.getRequest(command);
if (request != null) {
  // do something useful
}

public void doSomethingUseful(Request request) {
  if (request == null) {
    logger.warning("null command");
  }
  // do the real stuff
}
```

Motivation

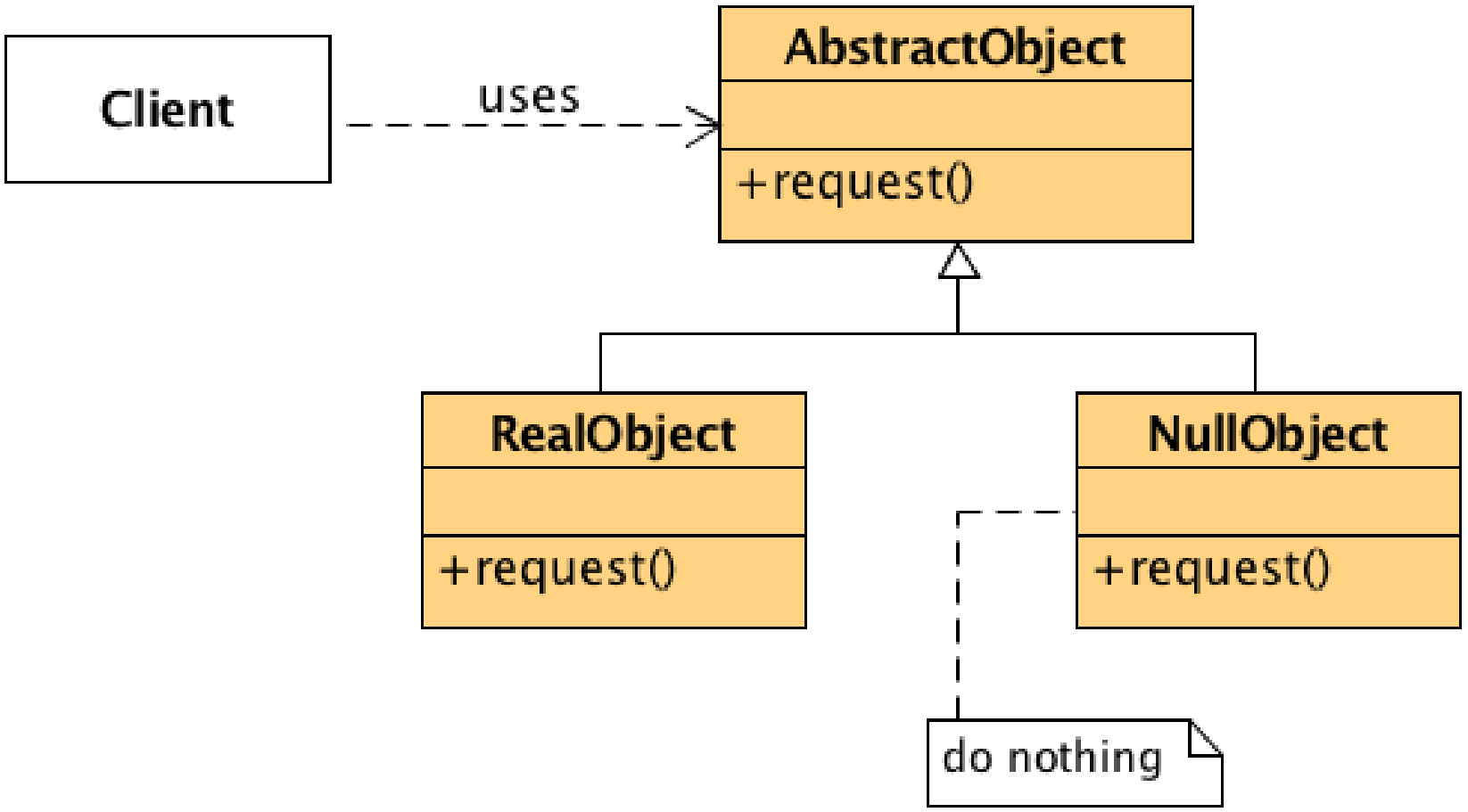
- **Problem**
  - How can the absence of an object — the presence of a null reference — be treated transparently?
- **Intent**
  - The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default **do nothing behavior**.
  - Use the Null Object pattern when you want to abstract the handling of null away from the client.

Null Object

```
private Request getRequest(String command) {
        if (command.equals("A"))
     return new ARequest();
   else if (command.equals("B"))
     return new BRequest();
        else
              return null;
}
```

```
private Request getRequest(String command) {
        if (command.equals("A"))
     return new ARequest();
   else if (command.equals("B"))
     return new BRequest();
        else
              return new NullRequest();
}
```

Structure


Client
uses
AbstractObject
+request()
RealObject
+request()
NullObject
+request()
do nothing

Example

```
public List<String> returnCollection() {
      //remainder omitted
  if (/*some condition*/) {
    return null;
  } else {    // return collection
  }
}
```

Example

```
public List<String> returnCollection() {
      //remainder omitted
  if (/*some condition*/) {
    return null;
  } else {    // return collection
  }
}

public List<String> returnCollection() {
      //remainder omitted
  if (/*some condition*/) {
    return Collections.emptyList();
  } else {
    // return collection
  }
}
```

```
// now we can do ...
if (obj.returnCollection().size() > 0) {
        // remainder omitted
```

Collections.emptyList()

```
public static final List EMPTY_LIST = new EmptyList<Object>();
//...

public static final <T> List<T> emptyList() {
 return (List<T>) EMPTY_LIST;
}

private static class EmptyList<E> extends AbstractList<E>
//...
        public int size() {return 0;}
        public boolean isEmpty() {return true;}

        public boolean contains(Object obj) {return false;}
        public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
        public Object[] toArray() { return new Object[0]; }
        public E get(int index) {
                throw new IndexOutOfBoundsException("Index: "+index);
        }
//...
```

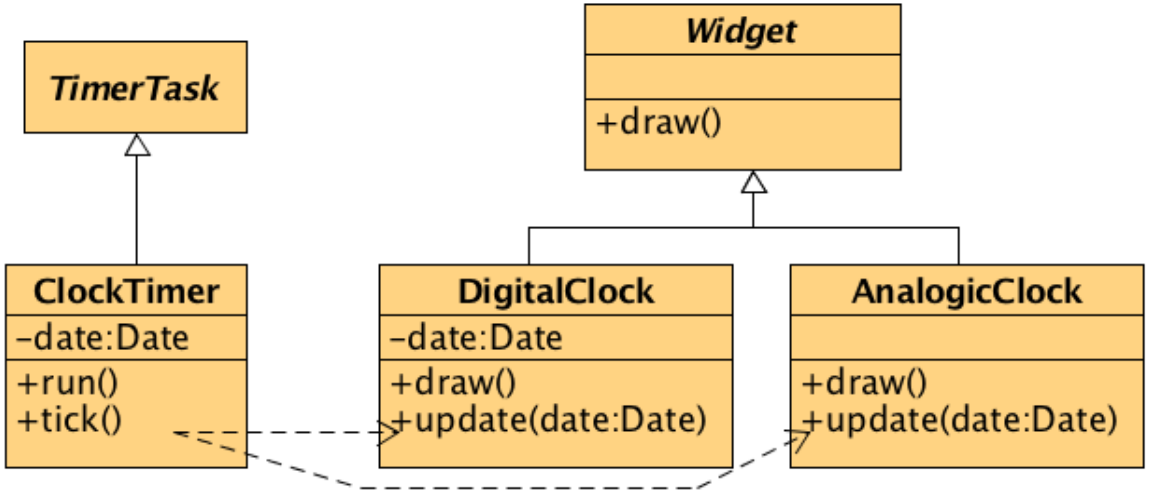
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Motivating example

- we want to implement a desktop widget representing a clock
  - two types of clocks are possible: digital and analogic
  - digital clock shows time in hh:mm format, analogic in hh:mm:ss
- there are two responsibilities: storing and updating time, displaying time
- want to reuse as much as possible of the classes, e.g. to make an alarm application

Solution (?)

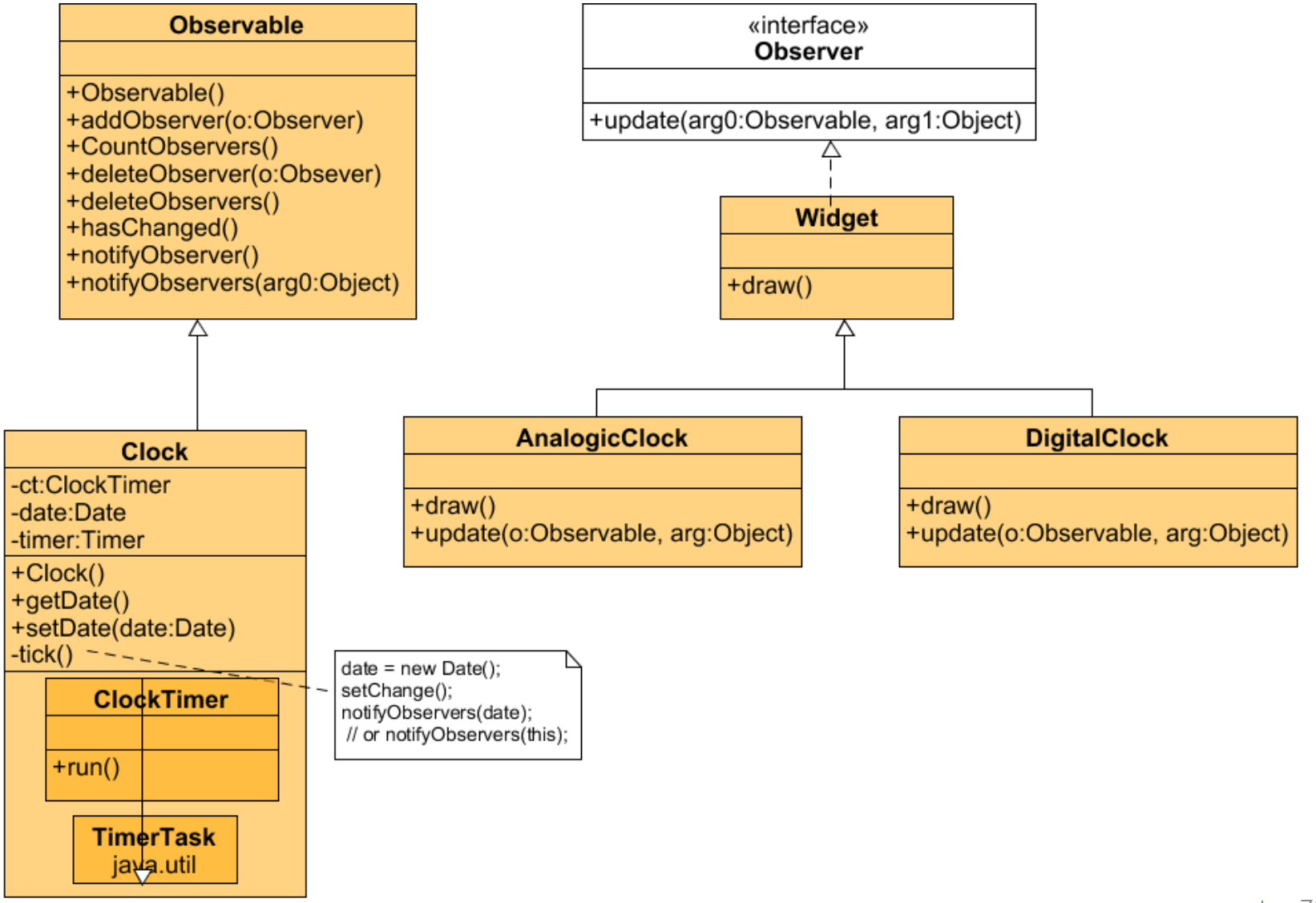


- ClockTimer is a (or has an object) thread able to periodically ask system date and call method tick
  - tick invokes update/s and sends it/them the present date update redraws the clock accordingly, if needed
  - but we don't want class ClockTimer to depend on (know) DigitalClock, AnalogicClock in order to reuse it

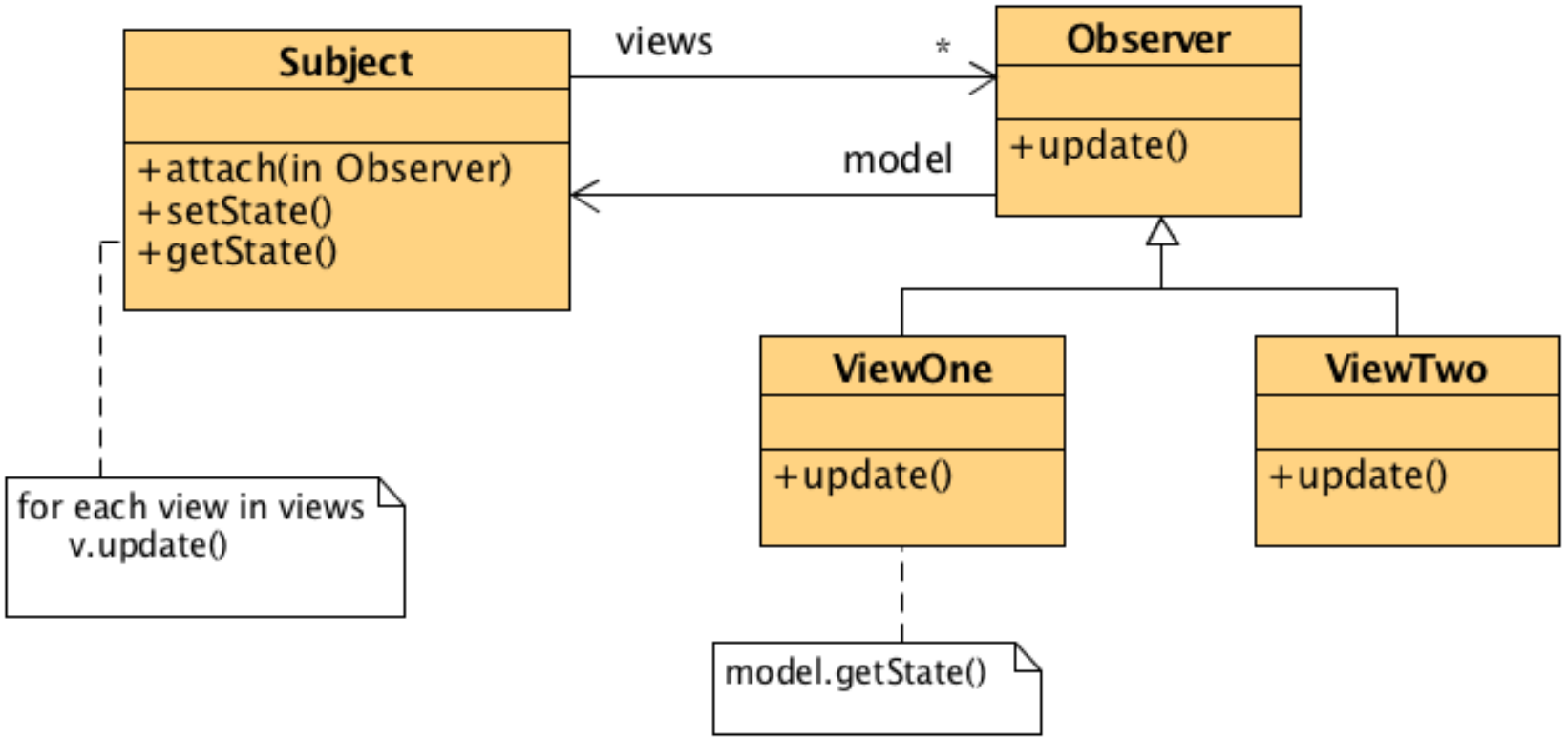
Motivation

- **Problem**
  - A large monolithic design does not scale well as new graphing or monitoring requirements are created.
- **Intent**
  - Define a **one-to-many dependency** between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
  - The "View" part of Model-View-Controller.

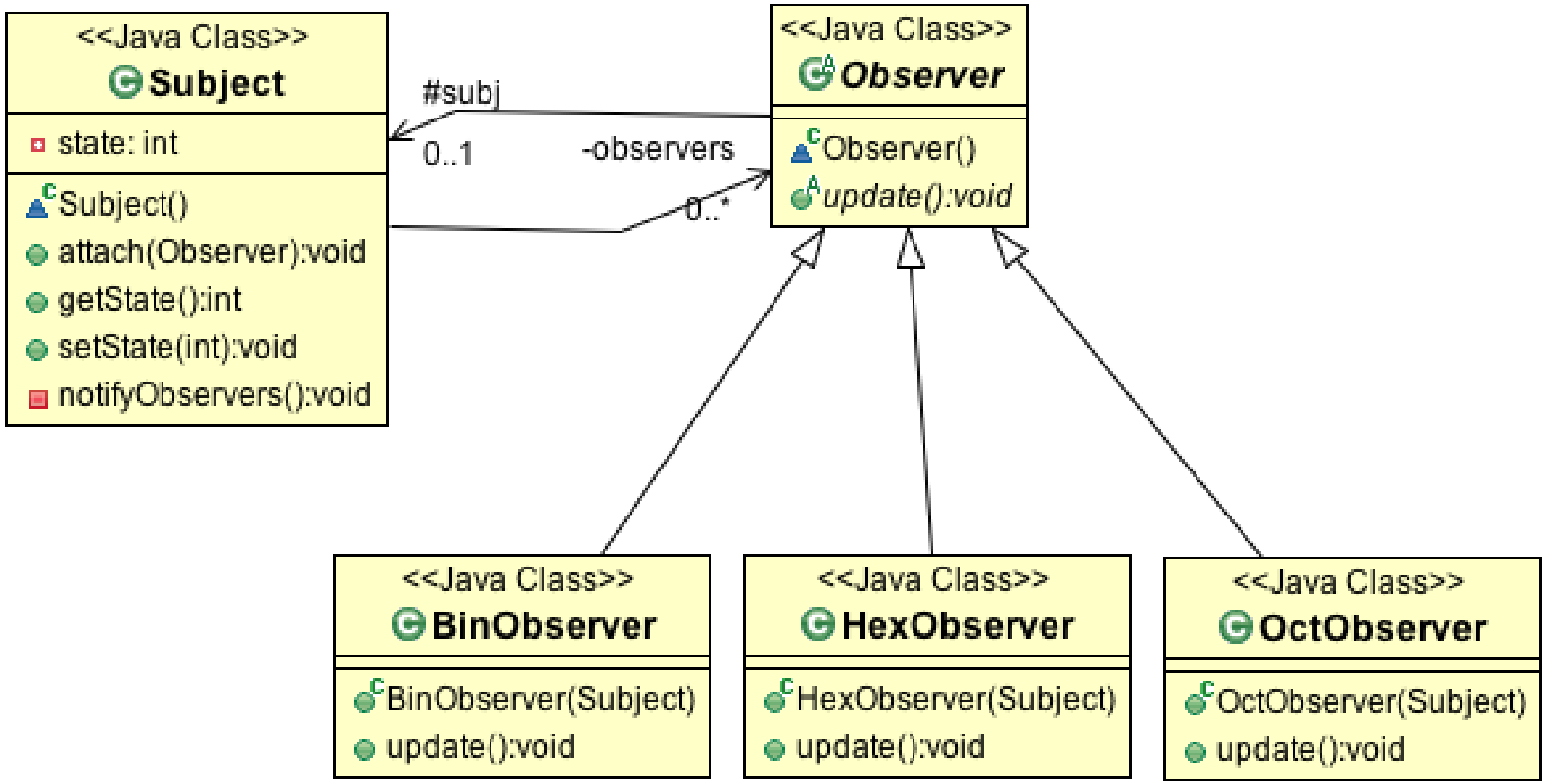
Solution


Observable
+Observable()
+addObserver(o:Observer)
+CountObservers()
+deleteObserver(o:Obsever)
+deleteObservers()
+hasChanged()
+notifyObserver()
+notifyObservers(arg0:Object)
«interface»
Observer
+update(arg0:Observable, arg1:Object)
Widget
+draw()
Clock
-ct:ClockTimer
-date:Date
-timer:Timer
+Clock()
+getDate()
+setDate(date:Date)
-tick()
ClockTimer
+run()
TimerTask
java.util
date = new Date();
setChange();
notifyObservers(date);
// or notifyObservers(this);
AnalogicClock
+draw()
+update(o:Observable, arg:Object)
DigitalClock
+draw()
+update(o:Observable, arg:Object)

Structure


Subject
+attach(in Observer)
+setState()
+getState()
views
*
Observer
+update()
model
ViewOne
+update()
ViewTwo
+update()
for each view in views
v.update()
model.getState()

Example


<<Java Class>>
Subject
state: int
Subject()
attach(Observer):void
getState():int
setState(int):void
notifyObservers():void
#subj
0..1
-observers
0..*
<<Java Class>>
Observer
Observer()
update():void
<<Java Class>>
BinObserver
BinObserver(Subject)
update():void
<<Java Class>>
HexObserver
HexObserver(Subject)
update():void
<<Java Class>>
OctObserver
OctObserver(Subject)
update():void

Example

```
class Subject {
	private List<Observer> observers = new ArrayList<>();
	private int state;
	public void attach(Observer o) {
		observers.add(o);
	}
	public int getState() {
		return state;
	}
	public void setState(int in) {
		state = in;
		notifyObservers();
	}
	private void notifyObservers() {
		for (Observer obs : observers) {
			obs.update();
		}
	}
}
```

Example

```
abstract class Observer {
	protected Subject subj;

	public abstract void update();
}

class HexObserver extends Observer {
	public HexObserver(Subject s) {
		subj = s;
		subj.attach(this); // Observers register themselves
	}

	public void update() {
		System.out.println("HexObserver saw "
				+ Integer.toHexString(subj.getState()));
	}                                                    // Observers "pull" information
}
```

Example

```
class OctObserver extends Observer {
        public OctObserver(Subject s) {
                subj = s;
                subj.attach(this);
        }
        public void update() {
                System.out.println("OctObserver saw "
                                + Integer.toOctalString(subj.getState()));
}       }

class BinObserver extends Observer {
        public BinObserver(Subject s) {
                subj = s;
                subj.attach(this);
        }
        public void update() {
                System.out.println("BinObserver saw "
                                + Integer.toBinaryString(subj.getState()));
}       }
```

Example

```
public class ObserverDemo {
	public static void main(String[] args) {
		Subject sub = new Subject();
		// Client configures the number and type of Observers
		new HexObserver(sub);
		new OctObserver(sub);
		new BinObserver(sub);
		Scanner scan = new Scanner(System.in);
		while (true) {
			System.out.print("\nEnter a number: ");
			sub.setState(scan.nextInt());
		}
	}
}
```

```
Enter a number: 25
HexObserver saw 19
OctObserver saw 31
BinObserver saw 11001

Enter a number: 77
HexObserver saw 4d
OctObserver saw 115
BinObserver saw 1001101
```

Observer in java libraries

- All implementations of java.util.EventListener
  - practically all over Swing components

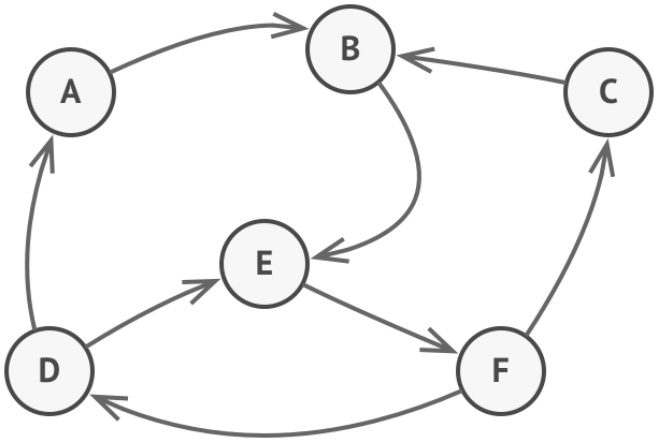
- Less commonly used:
  - class java.util.Observable
    - knows its observers (any number of objects)
    - provides an interface for adding and removing Observer objects.
  - interface java.util.Observer
    - defines an updating interface for objects that should be notified of changes in Observable

Check list

- Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
- Model the independent functionality with a "subject" abstraction.
- Model the dependent functionality with an "observer" hierarchy.
- The Subject is coupled only to the Observer base class.
- The client configures the number and type of Observers.
- Observers register themselves with the Subject.
- The Subject broadcasts events to all registered Observers.
- The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor



Motivation

- **Problem**
  - A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state.
  - Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.
- **Intent**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - An object-oriented state machine.
    - The State pattern is closely related to the concept of a Finite-State Machine.

Motivating problem

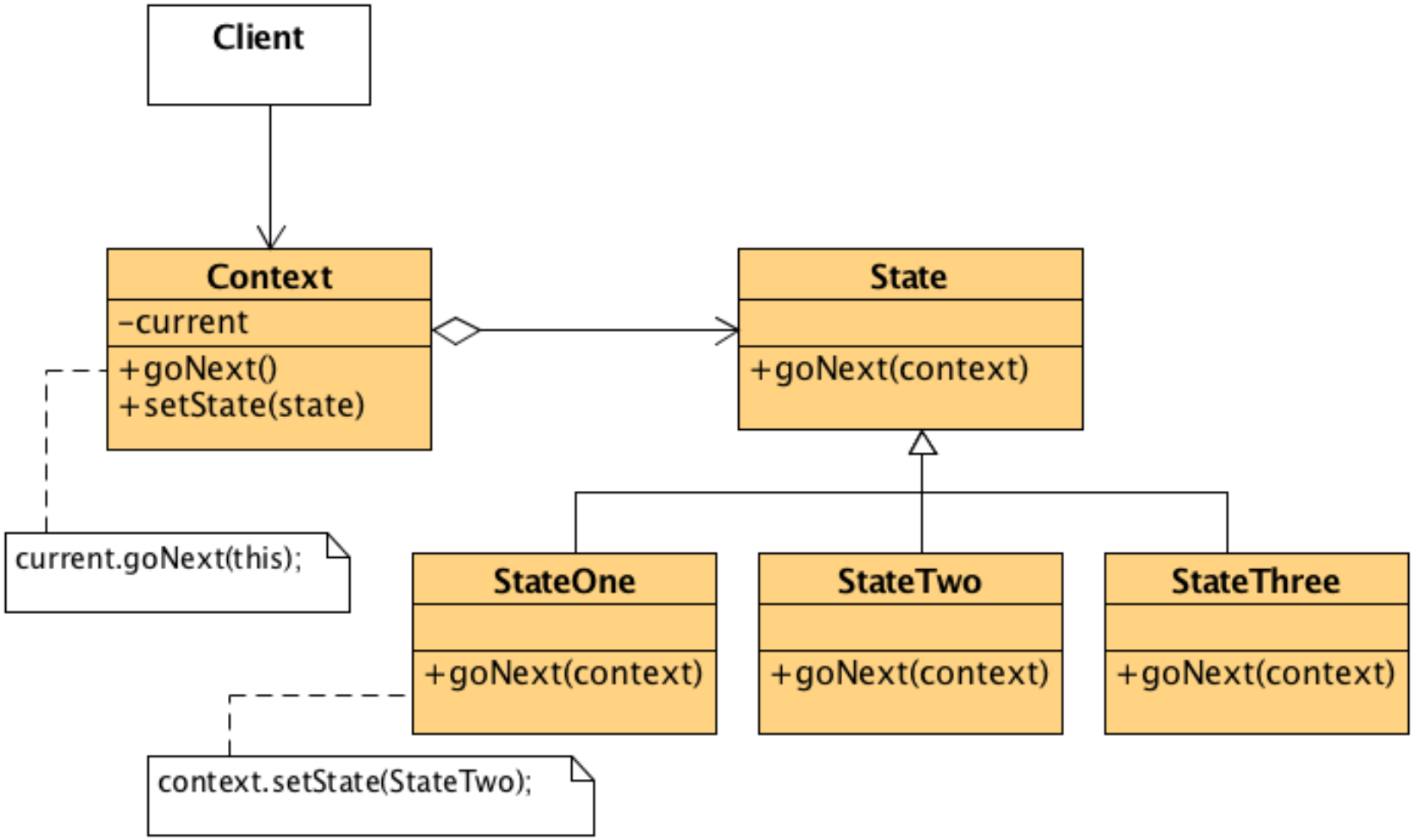
```
class CeilingFanPullChain1 {
        private int currentState;
        public CeilingFanPullChain1() {
                currentState = 0;
        }
        public void pull() {
                if (currentState == 0) {
                        currentState = 1;
                        System.out.println("   low speed");
                } else if (currentState == 1) {
                        currentState = 2;
                        System.out.println("   medium speed");
                } else if (currentState == 2) {
                        currentState = 3;
                        System.out.println("   high speed");
                } else {
                        currentState = 0;
                        System.out.println("   turning off");
}       }       }
```

Motivating problem

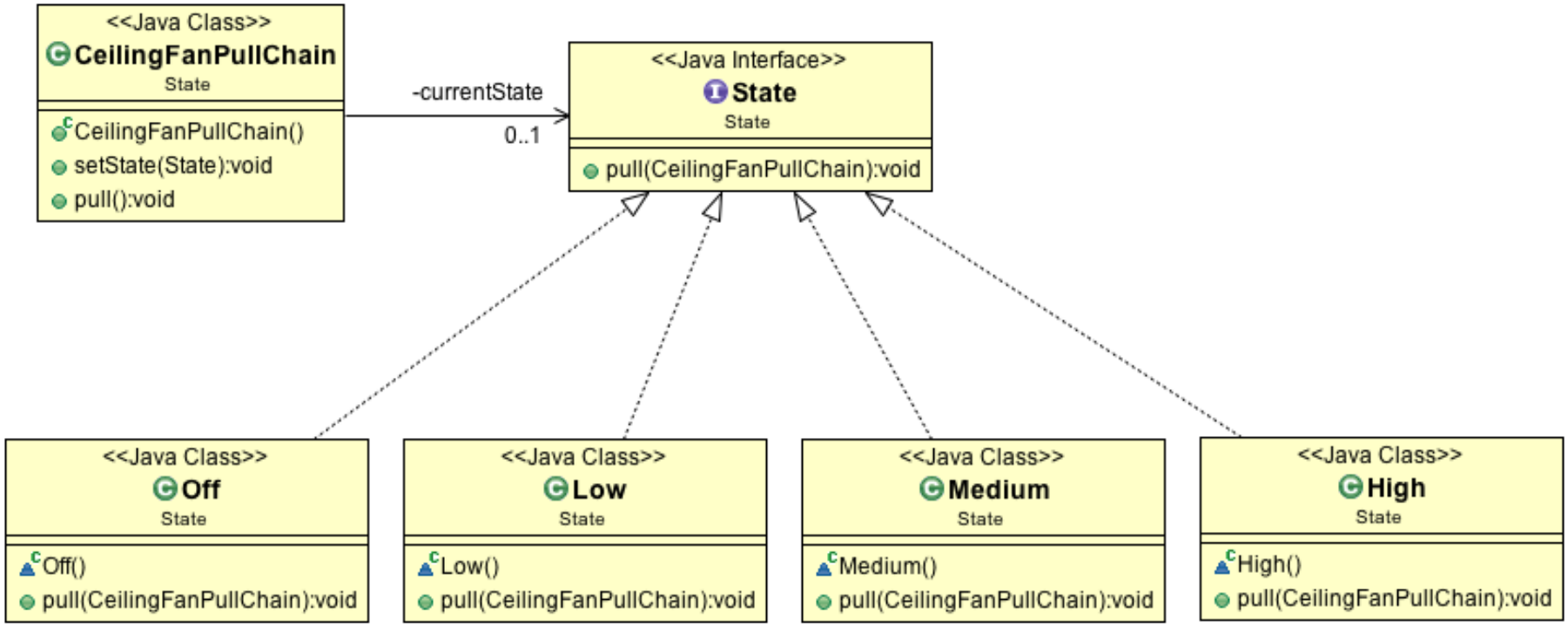
```
public class StateDemo {
	public static void main(String[] args) {
		CeilingFanPullChain1 chain = new CeilingFanPullChain1();
		while (true) {
			System.out.print("Enter.. ");
			Scanner scan = new Scanner(System.in);
			scan.nextLine();
			chain.pull();
		}
	}
}
```

```
Enter..
  low speed
Enter..
  medium speed
Enter..
  high speed
Enter..
  turning off
Enter..
  low speed
Enter..
  medium speed
```

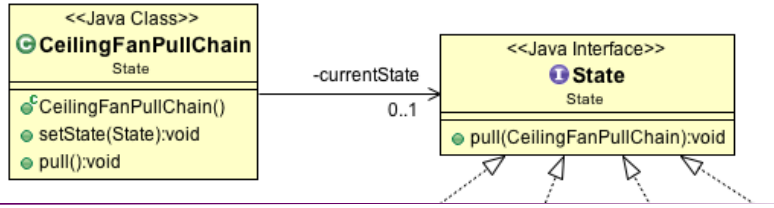
Structure


Client
Context
-current
+goNext()
+setState(state)
State
+goNext(context)
current.goNext(this);
StateOne
+goNext(context)
StateTwo
+goNext(context)
StateThree
+goNext(context)
context.setState(StateTwo);

Motivating problem: Solution

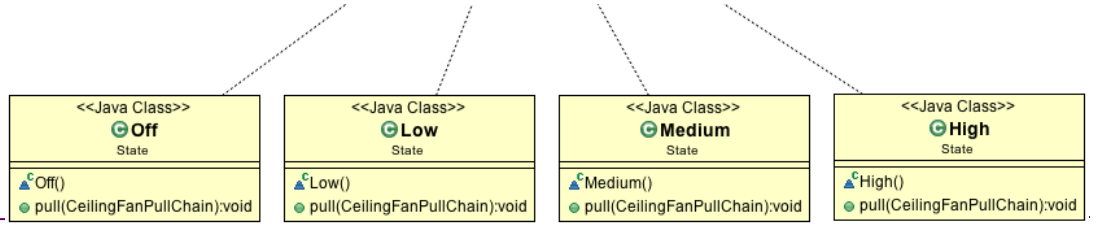


Solution




```
class CeilingFanPullChain {
        private State currentState;

        public CeilingFanPullChain() {
                currentState = new Off();
        }

        public void setState(State s) {
                currentState = s;
        }

        public void pull() {
                currentState.pull(this);
        }
}

interface State {
        void pull(CeilingFanPullChain wrapper);
}
```

Solution




```
class Off implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Low()); System.out.println(" low speed");
}   }

class Low implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Medium()); System.out.println(" medium speed");
}   }

class Medium implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new High()); System.out.println(" high speed");
}   }

class High implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Off()); System.out.println(" turning off");
}   }
```

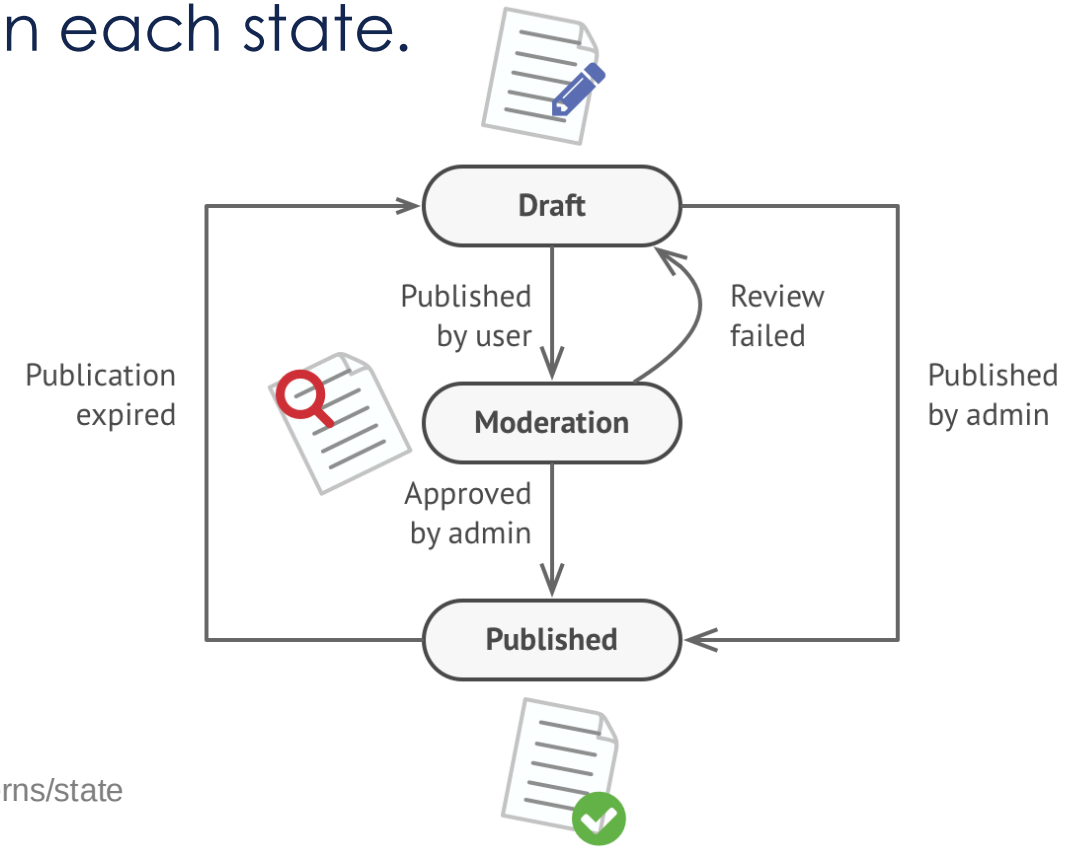
Solution – the client is the same

```
public class StateDemo {
	public static void main(String[] args) {
		CeilingFanPullChain1 chain = new CeilingFanPullChain1();
		while (true) {
			System.out.print("Enter.. ");
			Scanner scan = new Scanner(System.in);
			scan.nextLine();
			chain.pull();
		}
	}
}
```

```
Enter..
  low speed
Enter..
  medium speed
Enter..
  high speed
Enter..
  turning off
Enter..
  low speed
Enter..
  medium speed
```

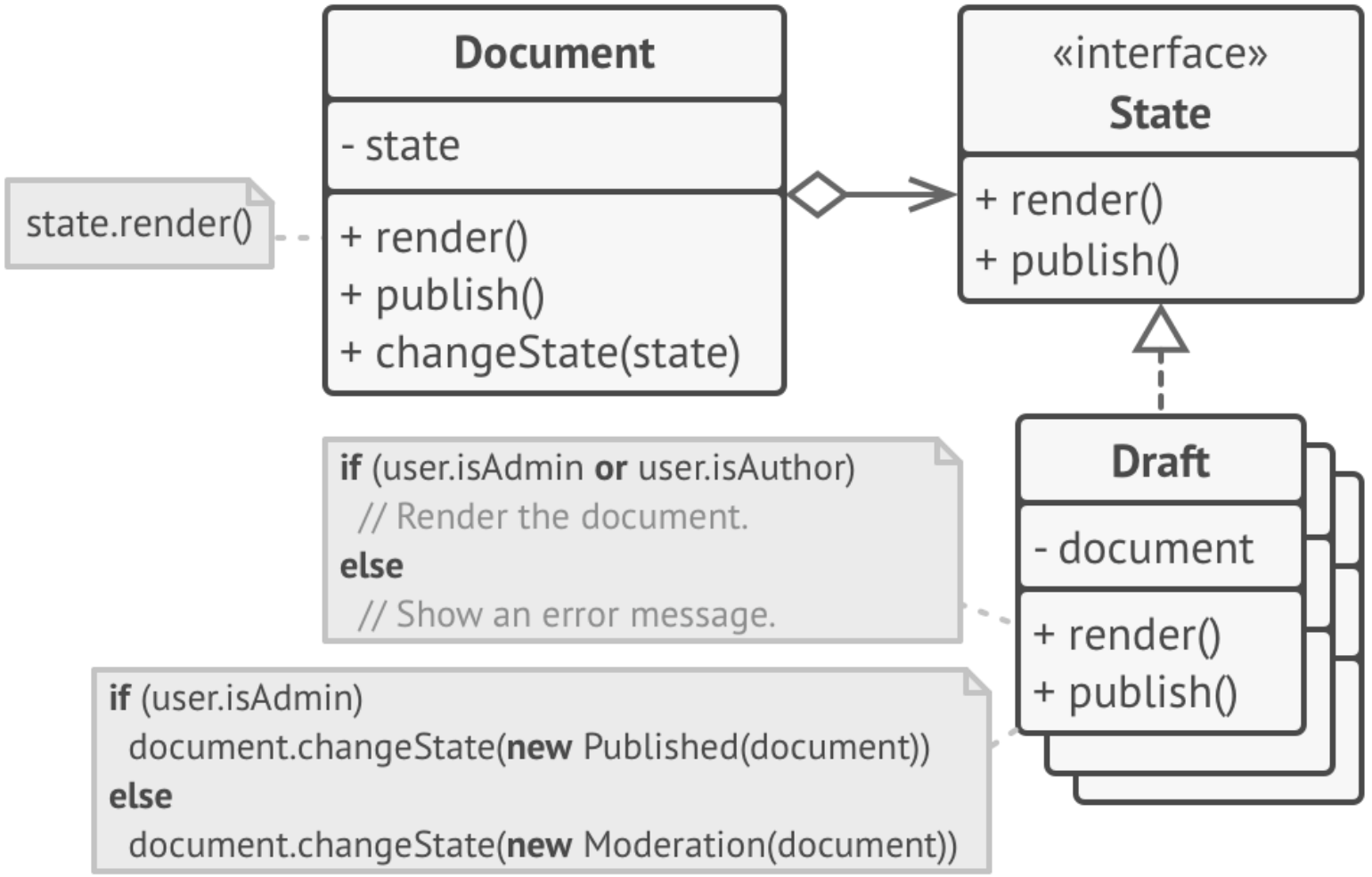
Another example

- A **Document** can be in one of three states:
  - **Draft**, **Moderation** and **Published**.
- The publish method of the document works a little bit differently in each state.



https://refactoring.guru/design-patterns/state

Solution



https://refactoring.guru/design-patterns/state

Applicability – when we have..

- **Object that behaves differently depending on its current state**, that has many states, and the state-specific code changes frequently.
  - The pattern suggests extracting all state-specific code into a set of distinct classes. As a result, you can add new states or change existing ones independently of each other.
- **Class polluted with massive conditionals** that alter how the class behaves according to the current values of the class's fields.
  - The State pattern lets you extract branches of these conditionals into methods of corresponding state classes. While doing so, you can also clean temporary fields and helper methods involved in state-specific code out of your main class.

Check list

- Identify an existing class, or create a new class, that will serve as the "state machine" from the client's perspective. That class is the "wrapper" class.
- Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter: an instance of the wrapper class.
- Create a State derived class for each domain state. These derived classes only override the methods they need to override.
- The wrapper class maintains a "current" State object.
- All client requests to the wrapper class are simply delegated to the current State object, and the wrapper object's this pointer is passed.
- The State methods change the "current" state in the wrapper object as appropriate.

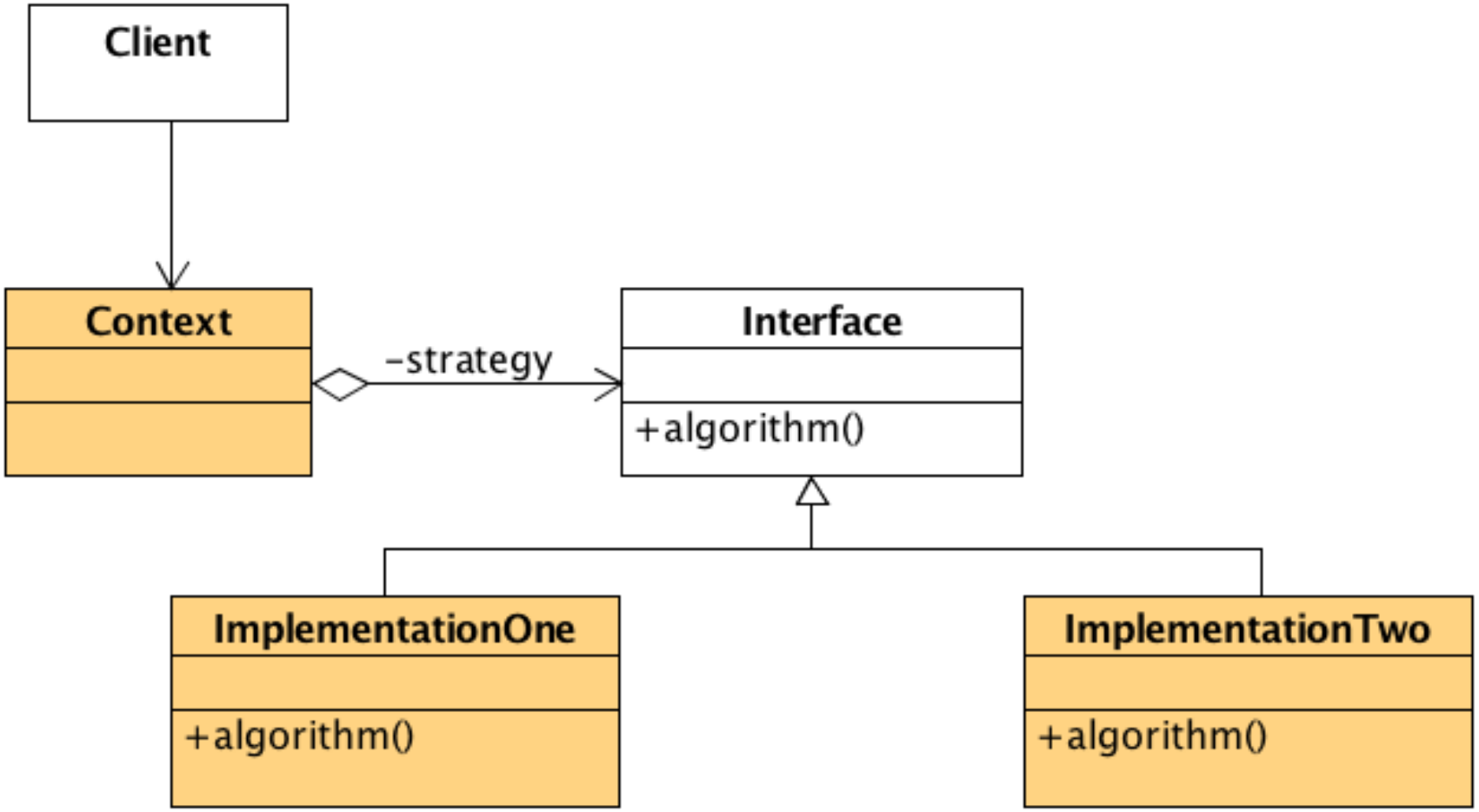
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

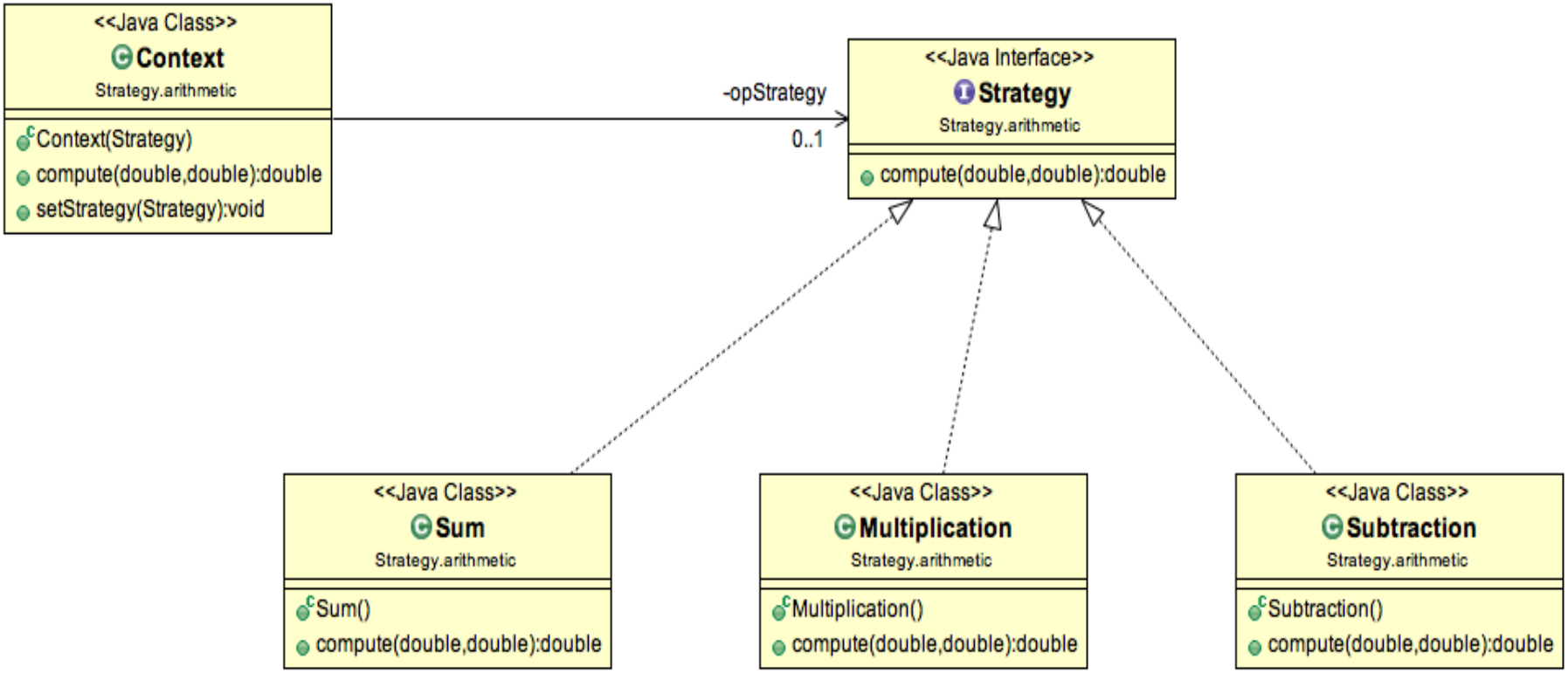
Motivation

- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from the clients that use it.
  - Capture the abstraction in an interface, bury implementation details in derived classes.
    - no impact when the number of derived classes changes,
    - no impact when the implementation of a derived class changes.

Structure


Client
Context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

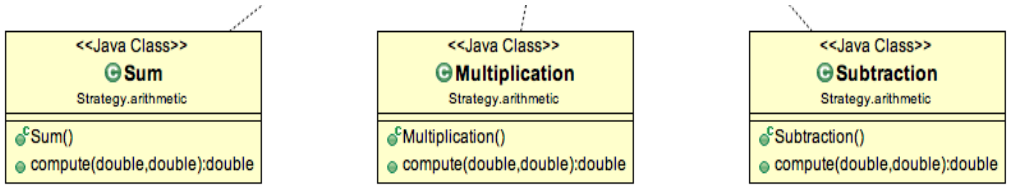
Example 1


<<Java Class>>
Context
Strategy.arithmetic
Context(Strategy)
compute(double,double):double
setStrategy(Strategy):void
-opStrategy
0..1
<<Java Interface>>
Strategy
Strategy.arithmetic
compute(double,double):double
<<Java Class>>
Sum
Strategy.arithmetic
Sum()
compute(double,double):double
<<Java Class>>
Multiplication
Strategy.arithmetic
Multiplication()
compute(double,double):double
<<Java Class>>
Subtraction
Strategy.arithmetic
Subtraction()
compute(double,double):double

Example 1

```
public interface Strategy {
        double compute(double elem1, double elem2);
}

public class Context {
        private Strategy opStrategy;
        public Context(Strategy operation) {
                this.opStrategy = operation;
        }
        public void setStrategy(Strategy strategy) {
                opStrategy = strategy;
        }
        public double compute(double firstNumber, double secondNumber) {
                return opStrategy.compute(firstNumber, secondNumber);
        }
}
```

Example 1




```
public class Sum implements Strategy {
        @Override
        public double compute(double elem1, double elem2) {
                return elem1 + elem2;
        }
}

public class Multiplication implements Strategy {
        @Override
        public double compute(double elem1, double elem2) {
                return elem1 * elem2;
        }
}

public class Subtraction implements Strategy {
        @Override
        public double compute(double elem1, double elem2) {
                return elem1 - elem2;
        }
}
```

Example 1

```
public class StrategyDemo {

	public static void main(String[] args) {
		double e1 = 5, e2 = 33;

		Context c = new Context(new Sum());
		System.out.println("Result: " + c.compute(e1, e2));

		c.setStrategy(new Subtraction());
		System.out.println("Result: " + c.compute(e1, e2));

		c.setStrategy(new Multiplication());
		System.out.println("Result: " + c.compute(e1, e2));
	}

}
```

```
Result: 38.0
Result: -28.0
Result: 165.0
```

Example 2 – we want this flexibility

```
public class SortingStrategyMain {
	public static void main(String args[]) {
		int[] var = { 1, 2, 3, 4, 5 };

		ContextWithSorting bub = new ContextWithSorting(new BubbleSort());
		bub.sort(var);

		ContextWithSorting quick = new ContextWithSorting(new QuickSort());
		quick.sort(var);
	}
}
```

```
sorting array using bubble sort strategy
sorting array using quick sort strategy
```

Example 2

```
interface SortStrategy {
        public void sort(int[] numbers);
}


class ContextWithSorting {
        private final SortStrategy strategy;
        public Context(SortStrategy strategy) {
                this.strategy = strategy;
        }
        public void sort(int[] input) {
                strategy.sort(input);
        }
}
```

Example 2

```
class BubbleSort implements SortStrategy {
        public void sort(int[] numbers) {
                System.out.println("sorting array using bubble sort strategy");
        }
}
class InsertionSort implements SortStrategy {
        public void sort(int[] numbers) {
                System.out.println("sorting array using insertion sort strategy");
        }
}
class QuickSort implements SortStrategy {
        public void sort(int[] numbers) {
                System.out.println("sorting array using quick sort strategy");
        }
}
class MergeSort implements SortStrategy {
        public void sort(int[] numbers) {
                System.out.println("sorting array using merge sort strategy");
        }
}
```

Example 3 – client side only

```
// compressing files in several formats
public class Client
{

  public static void main(String[] args)
  {
            // get a list of files (fileList)
              ...
      Compression ctx = new CompressionContext();
            ctx.setCompressionStrategy(new ZipCompressionStrategy());
            ctx.createArchive(fileList);
            ctx.setCompressionStrategy(new RarCompressionStrategy());
            ctx.createArchive(fileList);
            ctx.setCompressionStrategy(new NoCompressionStrategy());
            ctx.createArchive(fileList);

  }
}
```

UML Solution ?

Example 4 – with lambda expressions

```
// Example with lambda expressions
interface Computation<T> {
        public T compute(T n, T m);
}

public class StrategyPatternWithLambdas {
        public static void main(String[] args) {
        List<Computation<Integer>> computations =
         Arrays.asList(
                (n, m)-> { return n+m; },
                (n, m)-> { return n*m; },
                        (n, m)-> { return n-m; },
                                (n, m)-> { return Integer.parseInt(
                                                Integer.toString(n)+Integer.toString(m));}
         );
        computations.forEach((comp) -> System.out.println(comp.compute(10, 4)));
        }
}
```

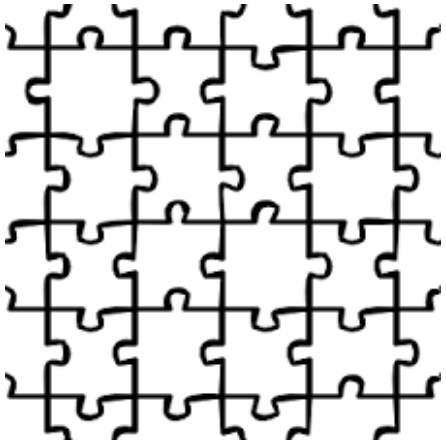
```
14
40
6
104
```

Short comparison

- Strategy is like State except in its intent.
  - Also the difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- Strategy lets you change the guts of an object. Decorator lets you change the skin.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the 'handle/body' idiom. They differ in intent - that is, they solve different problems.

Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor



Motivating example

- You want to program several games like chess, checkers, snakes and ladders, poker . . . They are played differently but have in common:
  - several players, each playing against the others only one is playing at a given time
  - playing order is : first player, second player . . . last, first, second . . .
  - game starts at an initial state
  - there is some game over condition, a unique winner

Motivating example

```
class Monopoly extends Game {
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
    void makePlay(int player) {
        // Process one turn of player
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    void printWinner() {
        // Display who won
    }
    // ...
}
```

```
class Chess extends Game {
    void initializeGame() {
        // Initialize players
        // Put the pieces on the board
    }
    void makePlay(int player) {
        // Process a turn of player
    }
    boolean endOfGame() {
        // Return true if Checkmate or
        // Stalemate has been reached
    }
    void printWinner() {
        // Display the winning player
    }
    // ...
}
```

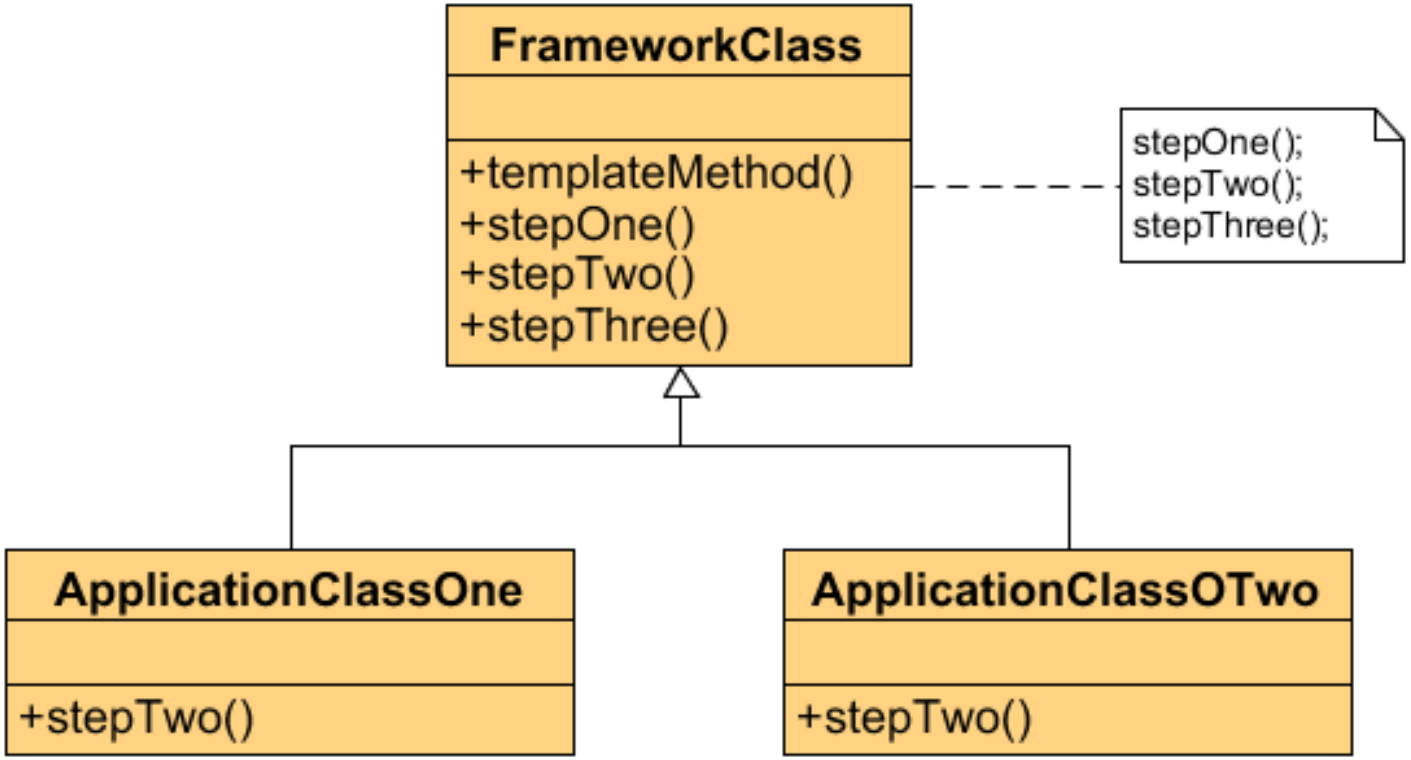
Template method

```
public abstract class Game {
   protected int playersCount;

       public abstract void initializeGame();
       public abstract void makePlay(int player);
       public abstract boolean endOfGame();
       public abstract void printWinner();
   /* A template method : */
  public final void playOneGame(int playersCount) {
      this.playersCount = playersCount;
      initializeGame();
      int j = 0;
      while (!endOfGame()) {
         makePlay(j);    j = (j + 1) % playersCount;
      }
      printWinner();
   }
}
```

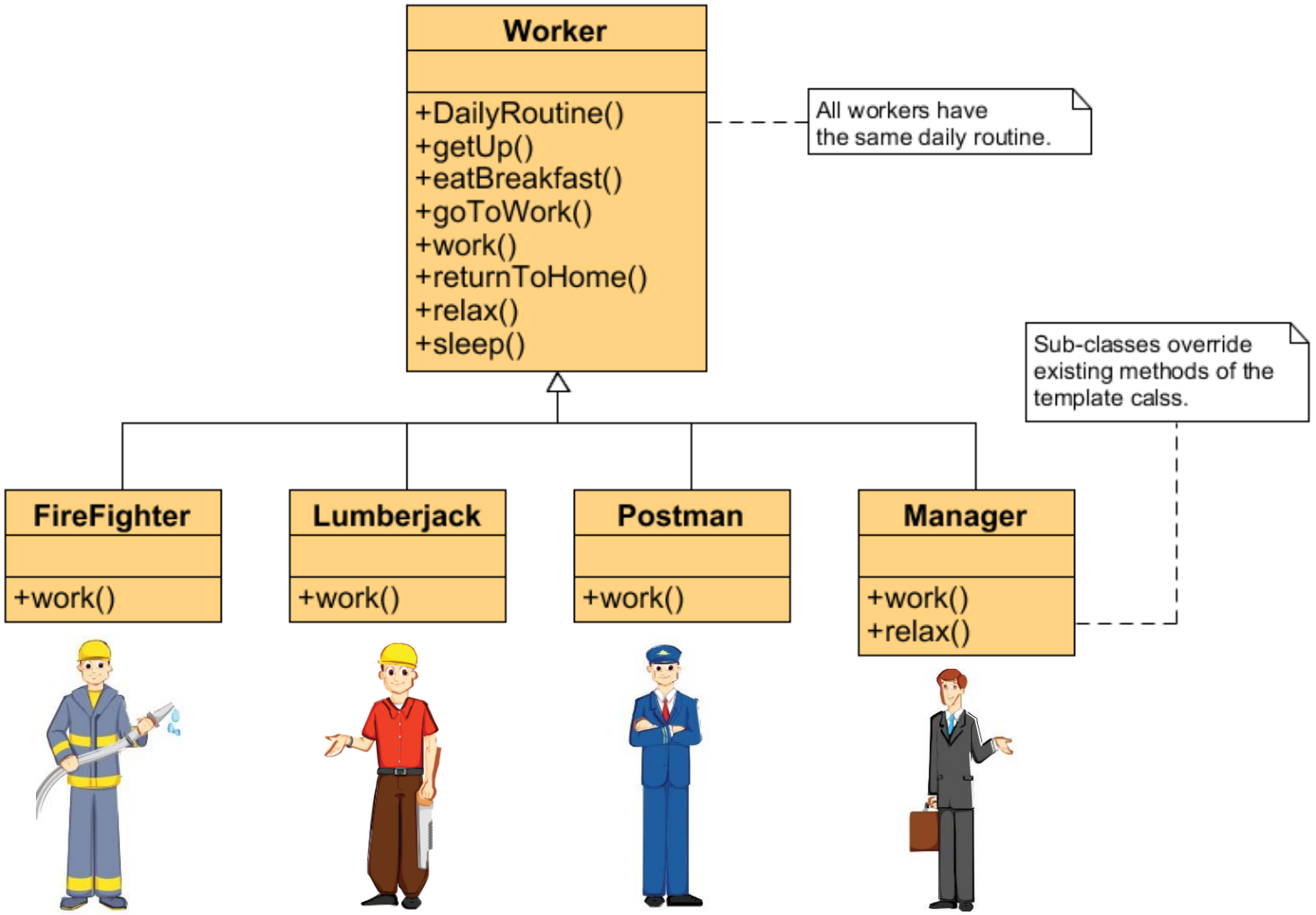
Motivation

- **Problem**
  - Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.
- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
  - Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

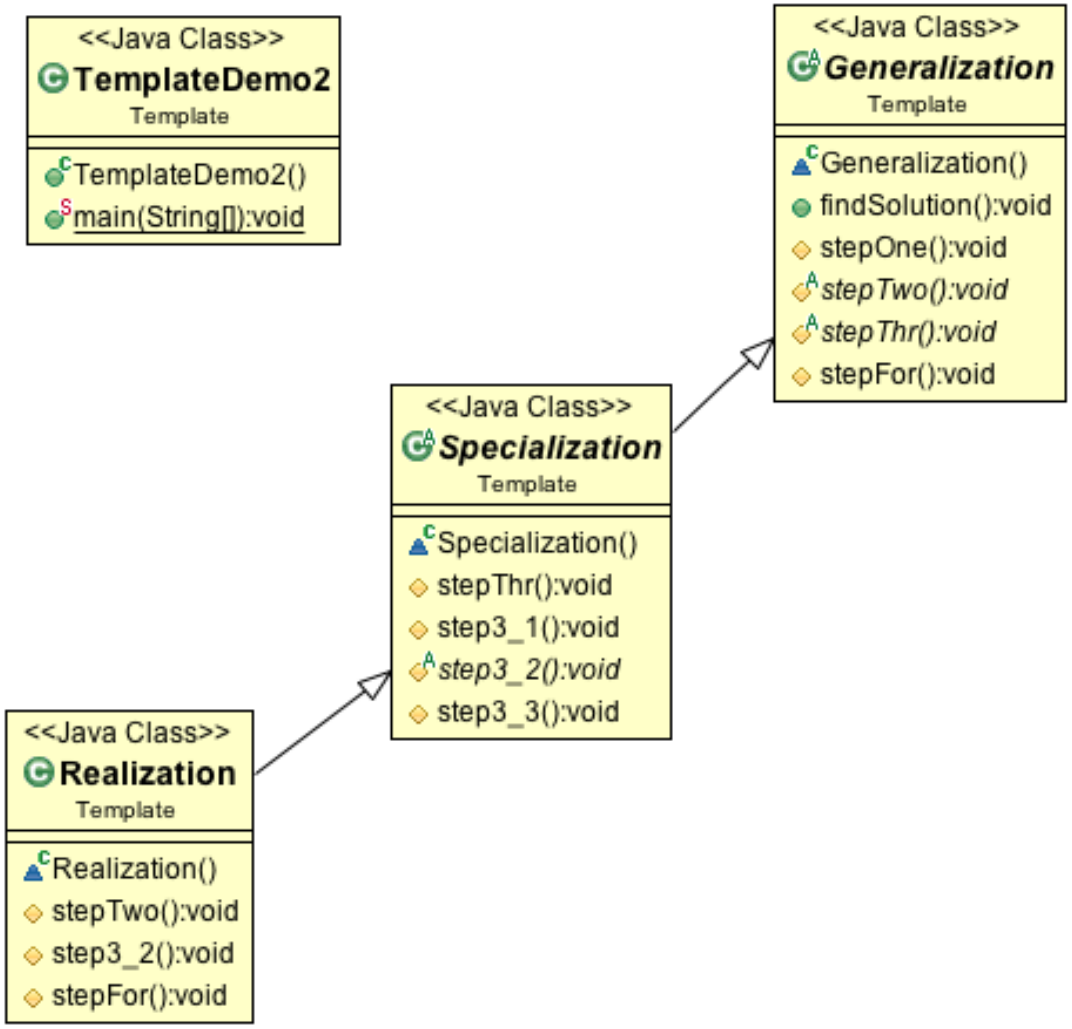
Structure


FrameworkClass
+templateMethod()
+stepOne()
+stepTwo()
+stepThree()
stepOne();
stepTwo();
stepThree();
ApplicationClassOne
+stepTwo()
ApplicationClassOTwo
+stepTwo()

Example 1


Worker
+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()
All workers have the same daily routine.
Sub-classes override existing methods of the template calss.
FireFighter
+work()
Lumberjack
+work()
Postman
+work()
Manager
+work()
+relax()

Example 2


<<Java Class>>
TemplateDemo2
Template
TemplateDemo2()
main(String[]):void
<<Java Class>>
Generalization
Template
Generalization()
findSolution():void
stepOne():void
stepTwo():void
stepThr():void
stepFor():void
<<Java Class>>
Specialization
Template
Specialization()
stepThr():void
step3_1():void
step3_2():void
step3_3():void
<<Java Class>>
Realization
Template
Realization()
stepTwo():void
step3_2():void
stepFor():void

Example 2

```
abstract class Generalization {
// 1. Standardize the skeleton of an algorithm in a "template" method
        public void findSolution() {
            stepOne();
            stepTwo();
            stepThr();
            stepFor();
        }
// 2. Common implementations of individual steps are defined in base class
        protected void stepOne()
                { System.out.println("Generalization.stepOne" ); }
 // 3. Steps requiring peculiar impls are ";placeholders" in the base class
   abstract protected void stepTwo();
   abstract protected void stepThr();
   protected void stepFor()
                { System.out.println( "Generalization.stepFor" ); }
}
```

Example 2

```
abstract class Specialization extends Generalization {
// 4. Derived classes can override placeholder methods
// 1. Standardize the skeleton of an algorithm in a "template" method
  protected void stepThr() {
    step3_1();
    step3_2();
    step3_3();
  }
// 2. Common implementations of individual steps are defined in base class
  protected void step3_1()
            { System.out.println( "Specialization.step3_1" ); }
// 3. Steps requiring peculiar impls are "placeholders" in the base class
  abstract protected void step3_2();
  protected void step3_3()
            { System.out.println( "Specialization.step3_3" ); }
}
```

Example 2

```
class Realization extends Specialization {
// 4. Derived classes can override placeholder methods
  protected void stepTwo(){ System.out.println("Realization.stepTwo" ); }
      protected void step3_2(){ System.out.println("Realization.step3_2" ); }
// 5. Derived classes can override implemented methods
// 6. Derived classes can override and "call back to" base class methods
  protected void stepFor() {
    System.out.println( "Realization.stepFor" );
    super.stepFor();
      }
}

public class TemplateDemo2 {
        public static void main( String[] args ) {
          Generalization algorithm =
                        new Realization();
          algorithm.findSolution();
      }
}
```

```
Generalization.stepOne
Realization.stepTwo
Specialization.step3_1
Realization.step3_2
Specialization.step3_3
Realization.stepFor
Generalization.stepFor
```

Template Methods in Java libraries

- All non-abstract methods of
  - java.io.InputStream
  - java.io.OutputStream
  - java.io.Reader
  - java.io.Writer

  - java.util.AbstractList
  - java.util.AbstractSet
  - java.util.AbstractMap

  - …

Check list

- Standardize the skeleton of an algorithm in a base class "template" method
- Common implementations of individual steps are defined in the base class
- Steps requiring peculiar implementations are "placeholders" in base class
- Derived classes can override placeholder methods
- Derived classes can override implemented methods
- Derived classes can override and "call back to" base class methods

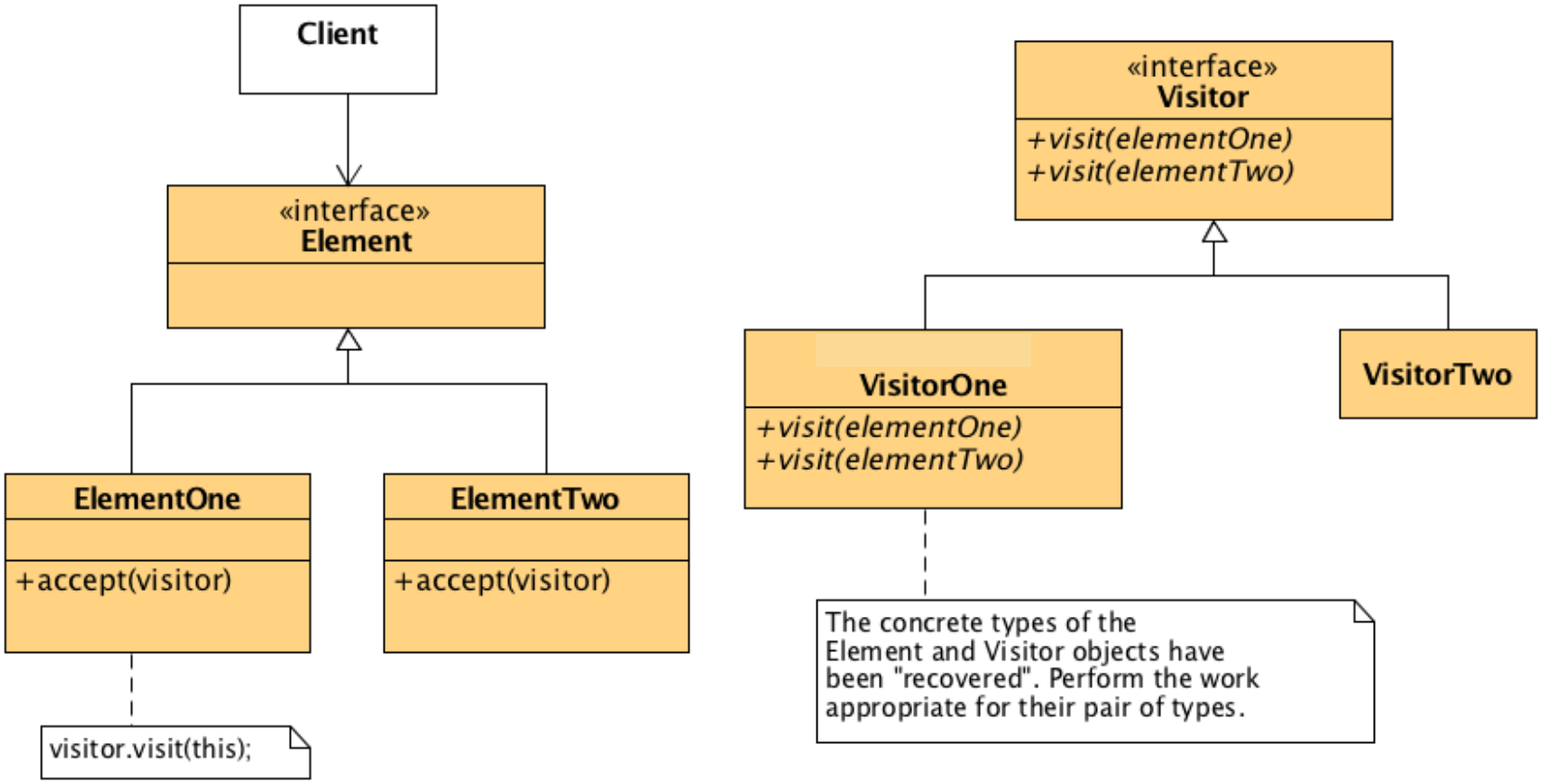
Behavioral design patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

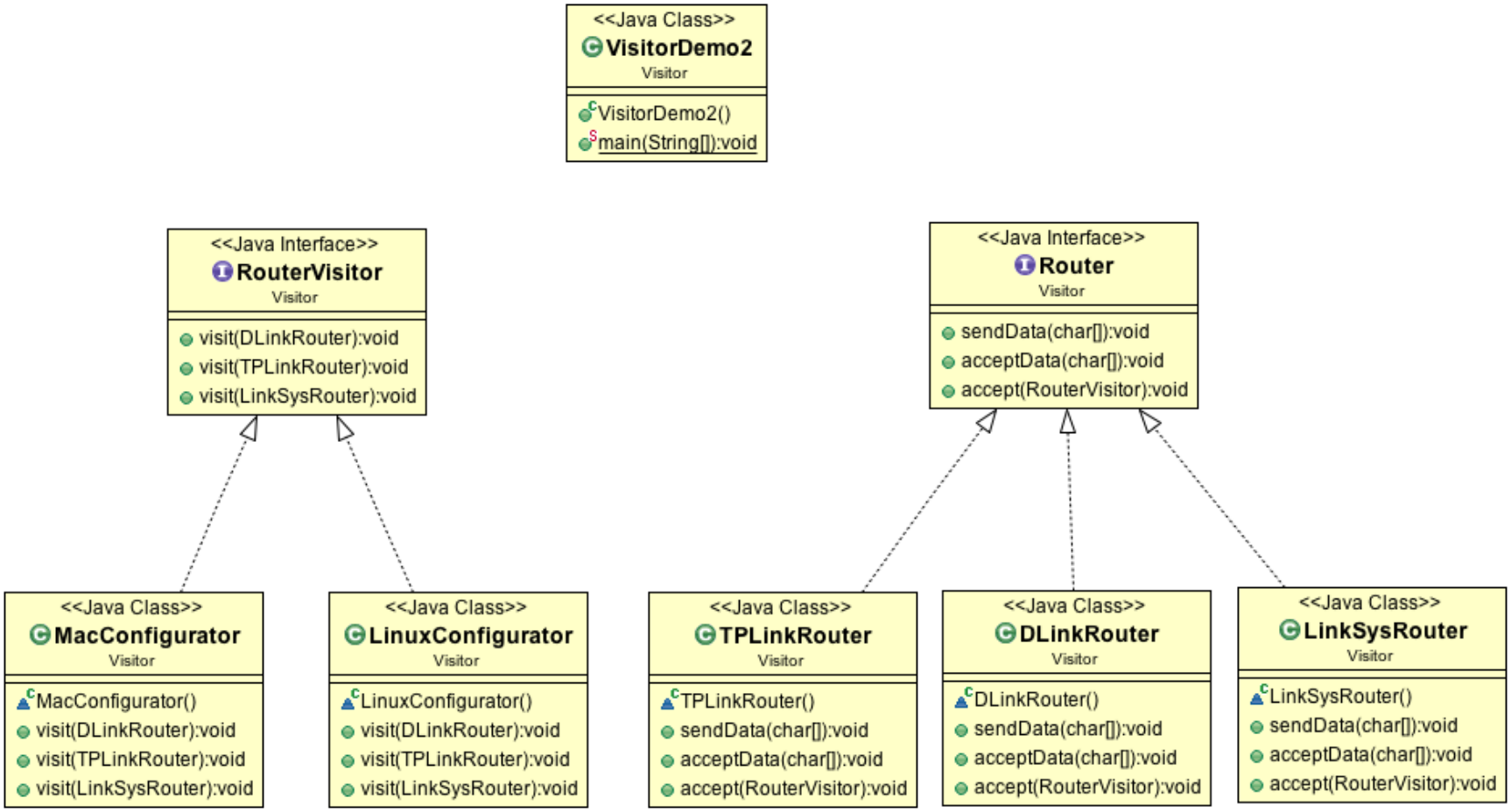
Motivation

- **Problem**
  - Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.
  - You want to avoid "polluting" the node classes with these operations.
- **Intent**
  - Represent an operation to be performed on the elements of an object structure.
  - Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Structure


Client
«interface»
Element
ElementOne
+accept(visitor)
ElementTwo
+accept(visitor)
visitor.visit(this);
«interface»
Visitor
+visit(elementOne)
+visit(elementTwo)
VisitorOne
+visit(elementOne)
+visit(elementTwo)
VisitorTwo
The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types.

Example


<<Java Class>>
VisitorDemo2
Visitor
VisitorDemo2()
main(String[]):void
<<Java Interface>>
RouterVisitor
Visitor
visit(DLinkRouter):void
visit(TPLinkRouter):void
visit(LinkSysRouter):void
<<Java Interface>>
Router
Visitor
sendData(char[]):void
acceptData(char[]):void
accept(RouterVisitor):void
<<Java Class>>
MacConfigurator
Visitor
MacConfigurator()
visit(DLinkRouter):void
visit(TPLinkRouter):void
visit(LinkSysRouter):void
<<Java Class>>
LinuxConfigurator
Visitor
LinuxConfigurator()
visit(DLinkRouter):void
visit(TPLinkRouter):void
visit(LinkSysRouter):void
<<Java Class>>
TPLinkRouter
Visitor
TPLinkRouter()
sendData(char[]):void
acceptData(char[]):void
accept(RouterVisitor):void
<<Java Class>>
DLinkRouter
Visitor
DLinkRouter()
sendData(char[]):void
acceptData(char[]):void
accept(RouterVisitor):void
<<Java Class>>
LinkSysRouter
Visitor
LinkSysRouter()
sendData(char[]):void
acceptData(char[]):void
accept(RouterVisitor):void

Example

```
interface Router {
        public void sendData(char[] data);
        public void acceptData(char[] data);
        public void accept(RouterVisitor v); // for visitor
}
class DLinkRouter implements Router {
        @Override public void sendData(char[] data)   { /* ... */  }
        @Override public void acceptData(char[] data) { /* ... */  }
        @Override public void accept(RouterVisitor v) {  v.visit(this);     }
}
class LinkSysRouter implements Router {
        @Override public void sendData(char[] data)   { /* ... */  }
        @Override public void acceptData(char[] data) { /* ... */  }
        @Override public void accept(RouterVisitor v) {  v.visit(this);     }
}
class TPLinkRouter implements Router {
        @Override public void sendData(char[] data)   { /* ... */  }
        @Override public void acceptData(char[] data) { /* ... */  }
        @Override public void accept(RouterVisitor v) {  v.visit(this);     }
}
```

Example

```
interface RouterVisitor {
        public void visit(DLinkRouter router);
        public void visit(TPLinkRouter router);
        public void visit(LinkSysRouter router);
}
class MacConfigurator implements RouterVisitor {
        @Override public void visit(DLinkRouter router) {
                // .. configuration here
                System.out.println("DLinkRouter Configuration for Mac complete !!");
        }
        @Override public void visit(TPLinkRouter router) {
                // .. configuration here
                System.out.println("TPLinkRouter Configuration for Mac complete !!");
        }

        @Override public void visit(LinkSysRouter router) {
                // .. configuration here
                System.out.println("LinkSysRouter Configuration for Mac complete !!");
        }
}
```

Example

```
class LinuxConfigurator implements RouterVisitor{

    @Override public void visit(DLinkRouter router) {
        // .. configuration here
        System.out.println("DLinkRouter Configuration for Linux complete !!");
    }

    @Override public void visit(TPLinkRouter router) {
        // .. configuration here
        System.out.println("TPLinkRouter Configuration for Linux complete !!");
    }

    @Override public void visit(LinkSysRouter router) {
        // .. configuration here
        System.out.println("LinkSysRouter Configuration for Linux complete !!");
    }
}
```

Example

```
public class VisitorDemo2 {
	public static void main(String s[]) {

		Router[] routers = { new DLinkRouter(),
					new TPLinkRouter(),
					new LinkSysRouter() };
		RouterVisitor[] visitors= {
					new MacConfigurator(),
					new LinuxConfigurator() };

		for (Router router: routers)
			for (RouterVisitor rvisitor : visitors)
				router.accept(rvisitor);
	}
}
```

```
DLinkRouter Configuration for Mac complete !!
DLinkRouter Configuration for Linux complete !!
TPLinkRouter Configuration for Mac complete !!
TPLinkRouter Configuration for Linux complete !!
LinkSysRouter Configuration for Mac complete !!
LinkSysRouter Configuration for Linux complete !!
```

Java SE – Interface FileVisitor<T>

```
Path start = Paths.get("/someFolder");
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
   @Override
   public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
      throws IOException
   {
      System.out.println("Visiting File: "+file.getFileName());
      return FileVisitResult.CONTINUE;
   }
   @Override
   public FileVisitResult postVisitDirectory(Path dir, IOException e)
      throws IOException
   {
      if (e == null) {
            System.out.println("Leaving Folder: "+dir.getFileName());
         return FileVisitResult.CONTINUE;
      } else {
         // directory iteration failed
         throw e;
      }
   }
});
```

```
Visiting File: banner.pdf
Visiting File: index.html
Leaving Folder: preview
Visiting File: cartaz_A4.pdf
Visiting File: cartaz_A1.pdf
Leaving Folder: conf
...
```

Consequences

- add a new operation is easy: make a new visitor
- add a new class X in the structure is difficult: add a new *visitX(X)* every visitor
- may combine with Iterator to traverse the structure
- structure may be a Composite
- visitor object may accumulate state while traversing (e.g., count number of visited objects of each type)
- may be forced to break encapsulation of visited classes to allow visitors do their job

Check list

- Confirm that the current hierarchy will be fairly stable and that the public interface of these classes is sufficient for the access the Visitor classes will require.
  - If these conditions are not met, then the Visitor pattern is not a good match.
- Create a Visitor base class with a *visit(ElementXxx)* method for each Element derived type.
- Add an *accept(Visitor)* method to the Element hierarchy.
  - The implementation in each Element derived class is always the same – accept(Visitor v) { v.visit(this); }.
- The Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is coupled to each Element derived class.
- Create a Visitor derived class for each "operation" to be performed on Element objects.
  - visit() implementations will rely on the Element's public interface.

Behavioral patterns – Summary

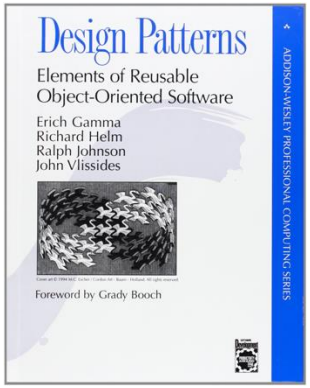
- Chain of responsibility
  - A way of passing a request between a chain of objects
- Command
  - Encapsulate a command request as an object
- Iterator
  - Sequentially access the elements of a collection
- Mediator
  - Defines simplified communication between classes
- Memento
  - Capture and restore an object's internal state

Behavioral patterns – Summary

- Null Object
  - Designed to act as a default value of an object
- Observer
  - A way of notifying change to a number of classes
- State
  - Alter an object's behavior when its state changes
- Strategy
  - Encapsulates an algorithm inside a class
- Template method
  - Defer the exact steps of an algorithm to a subclass
- Visitor
  - Defines a new operation to a class without change

Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.
- Effective Java, 2nd Ed., Joshua Bloch



- *Design Patterns Explained Simply (sourcemaking.com)*
- *https://refactoring.guru/design-patterns*