DESIGN PATTERNS

Course 6

PREVIOUS COURSE CONTENT

- Structural patterns
  - Decorator
  - Proxy
- Behavioral patterns
  - State
  - Visitor
  - Null Objectss

CURRENT CURSE CONTENT

- Behavioral patterns
  - Template Method
  - Command
  - Interpreter
  - Mediator
  - Memento

TEMPLATE METHOD

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.

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.

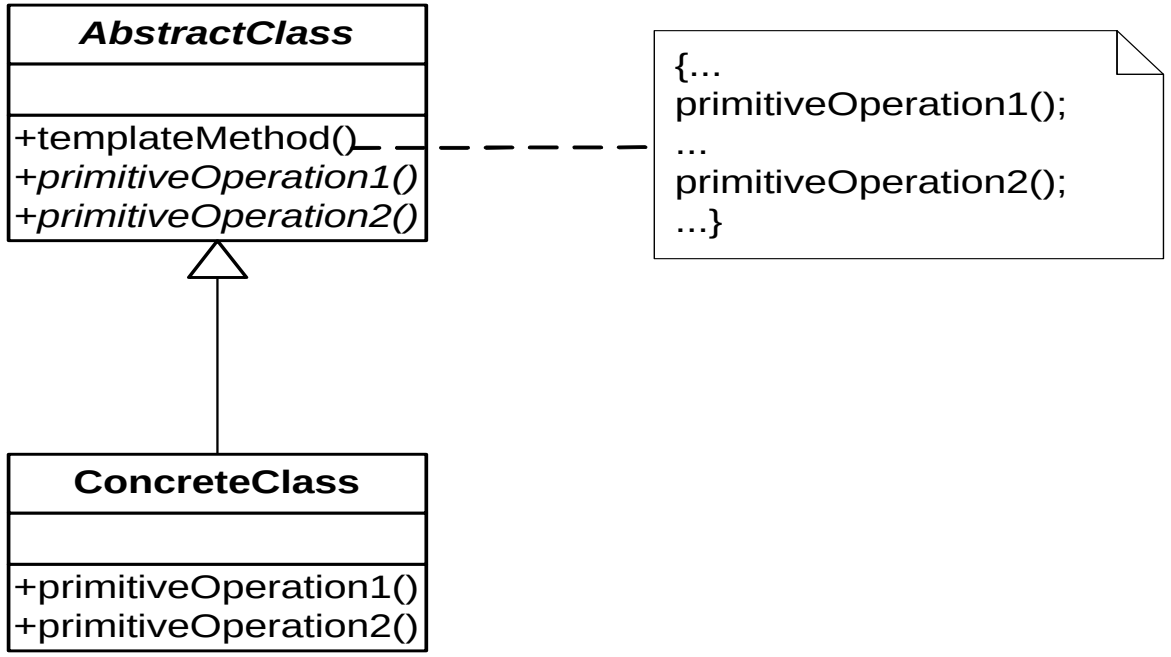
TEMPLATE METHOD. EXAMPLE

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.

- When refactoring is performed and common behavior is identified among classes. A abstract base class containing all the common code (in the template method) should be created to avoid code duplication.

- TEMPLATE METHOD - respects
  - **Hollywood Priciple:** Don't call us we will call you.

TEMPLATE METHOD



AbstractClass

- defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.

ConcreteClass

- implements the primitive operations to carry out subclass-specific steps of the algorithm.

TEMPLATE METHOD

Example

- Develop an application for a travel agency. The travel agency is managing each trip. All the trips contain common behavior but there are several packages. For example each trip contains the basic steps:

  - The tourists are transported to the holiday location by plane/train/ships,...
  - Each day they are visiting something
  - They are returning back home.

TEMPLATE METHOD

```
public class Trip {
        public final void performTrip(){
                doComingTransport();
                doDayA();
                doDayB();
                doDayC();
                doReturningTransport
        }
        public abstract void doComingTransport();
        public abstract void doDayA();
        public abstract void doDayB();
        public abstract void doDayC();
        public abstract void doReturningTransport();
}
```

TEMPLATE METHOD

```
public class PackageA extends Trip {
	public void doComingTransport() {
		System.out.println("The turists are comming by air ...");
	}
	public void doDayA() {
		System.out.println("The turists are visiting the aquarium
...");
	}
	public void doDayB() {
		System.out.println("The turists are going to the beach
...");
	}
	public void doDayC() {
		System.out.println("The turists
are going to mountains ...");
	}
	public void doReturningTransport()
{
		System.out.println("The turists
are going home by air ...");
	}
}
```

TEMPLATE METHOD

```
public class PackageB extends Trip {
	public void doComingTransport() {
		System.out.println("The turists are comming by train ...");
	}
	public void doDayA() {
		System.out.println("The turists are visiting the mountain ...");
	}
	public void doDayB() {
		System.out.println("The turists are going to the beach ...");
	}
	public void doDayC() {
		System.out.println("The turists are going to zoo ...");
	}
	public void doReturningTransport() {
		System.out.println("The turists are going home by train ...");
	}
}
```

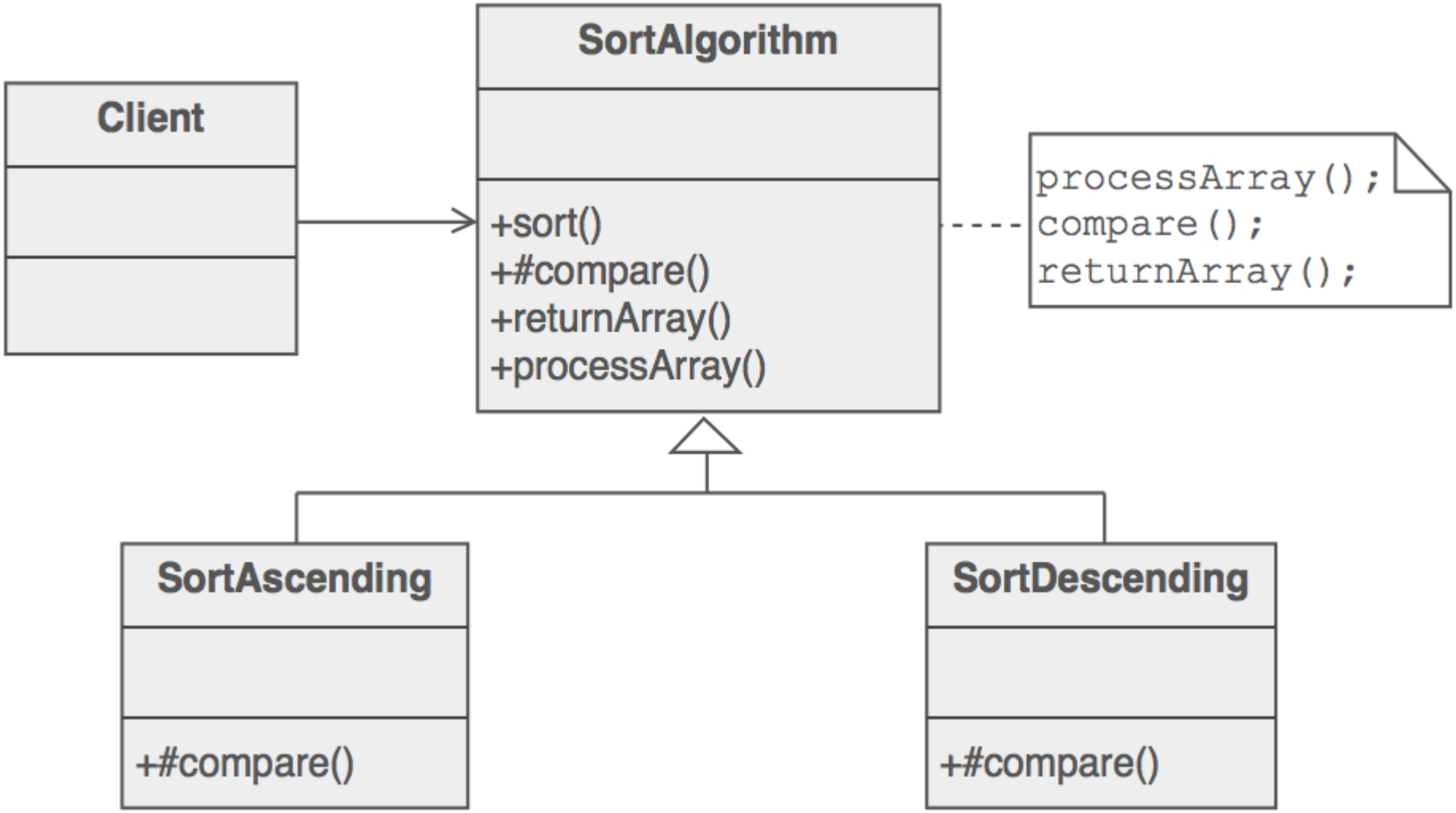
TEMPLATE METHOD

```
public class TemplatePatternDemo {
   public static void main(String[] args) {

      Trip trip= new PackageA();
      trip.performTrip();
      System.out.println();
      trip= new PackageB();
      trip.performTrip();
   }
}
```

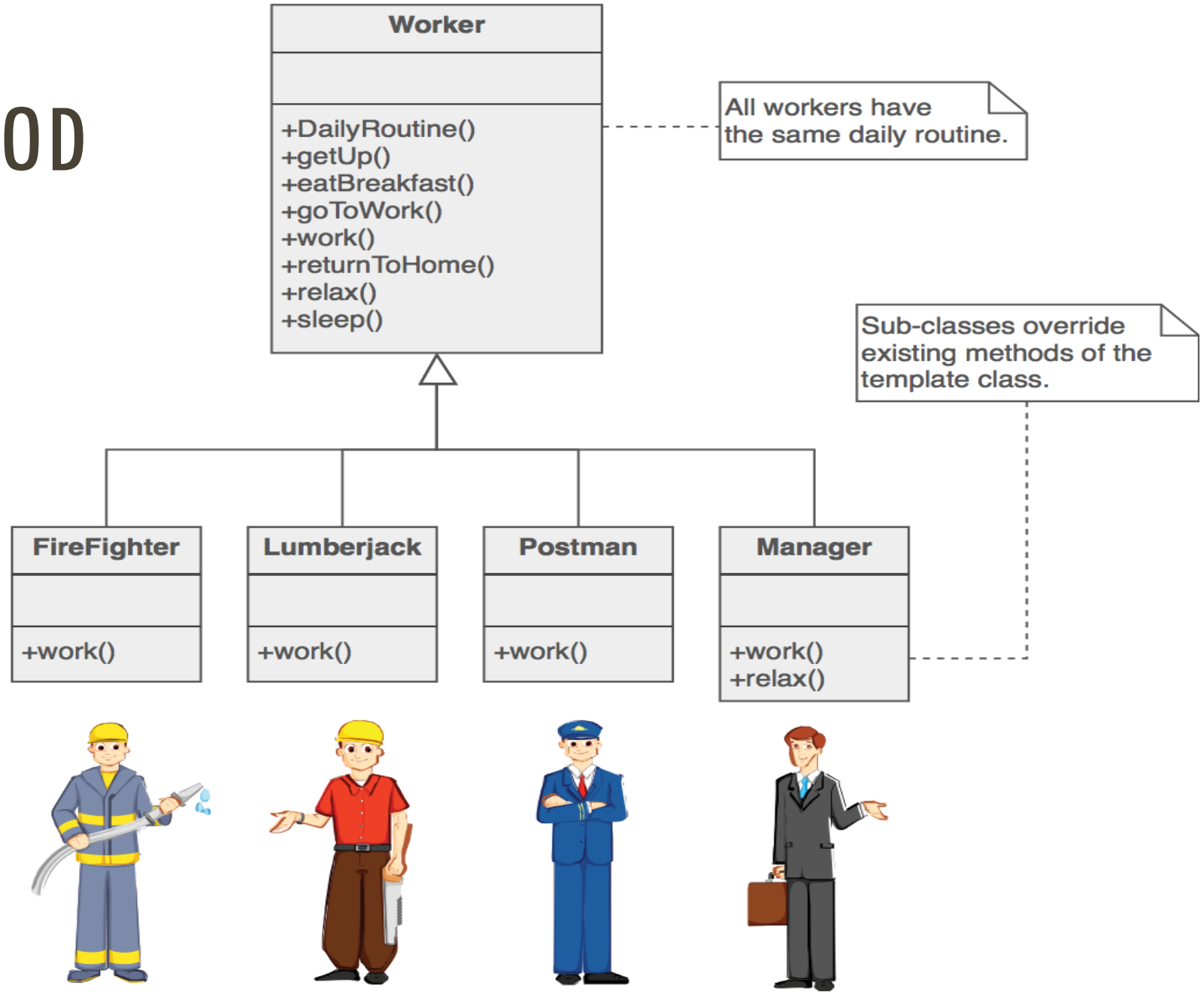
TEMPLATE METHOD

EXAMPLE



TEMPLATE METHOD

EXAMPLE



TEMPLATE METHOD

Used in java API

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.
- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

TEMPLATE METHOD

Advantages

- No code duplication between the classes
- Inheritance and Not Composition
- By taking advantage of polymorphism the superclass automatically calls the methods of the correct subclasses.

Disadvantages

- base classes tend to get cluttered up with a lot of seemingly unrelated code.
- Program flow is a little more difficult to follow - without the help of stepping through the code with a debugger.

COMMAND

Intent

- encapsulate a request in an object
- allows the parameterization of clients with different requests
- allows saving the requests in a queue

Problem

- Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

COMMAND

Command

- declares an interface for executing an operation;

ConcreteCommand

- extends the Command interface, implementing the Execute method by invoking the corresponding operation on Receiver. It defines a link between the Receiver and th action.

Client

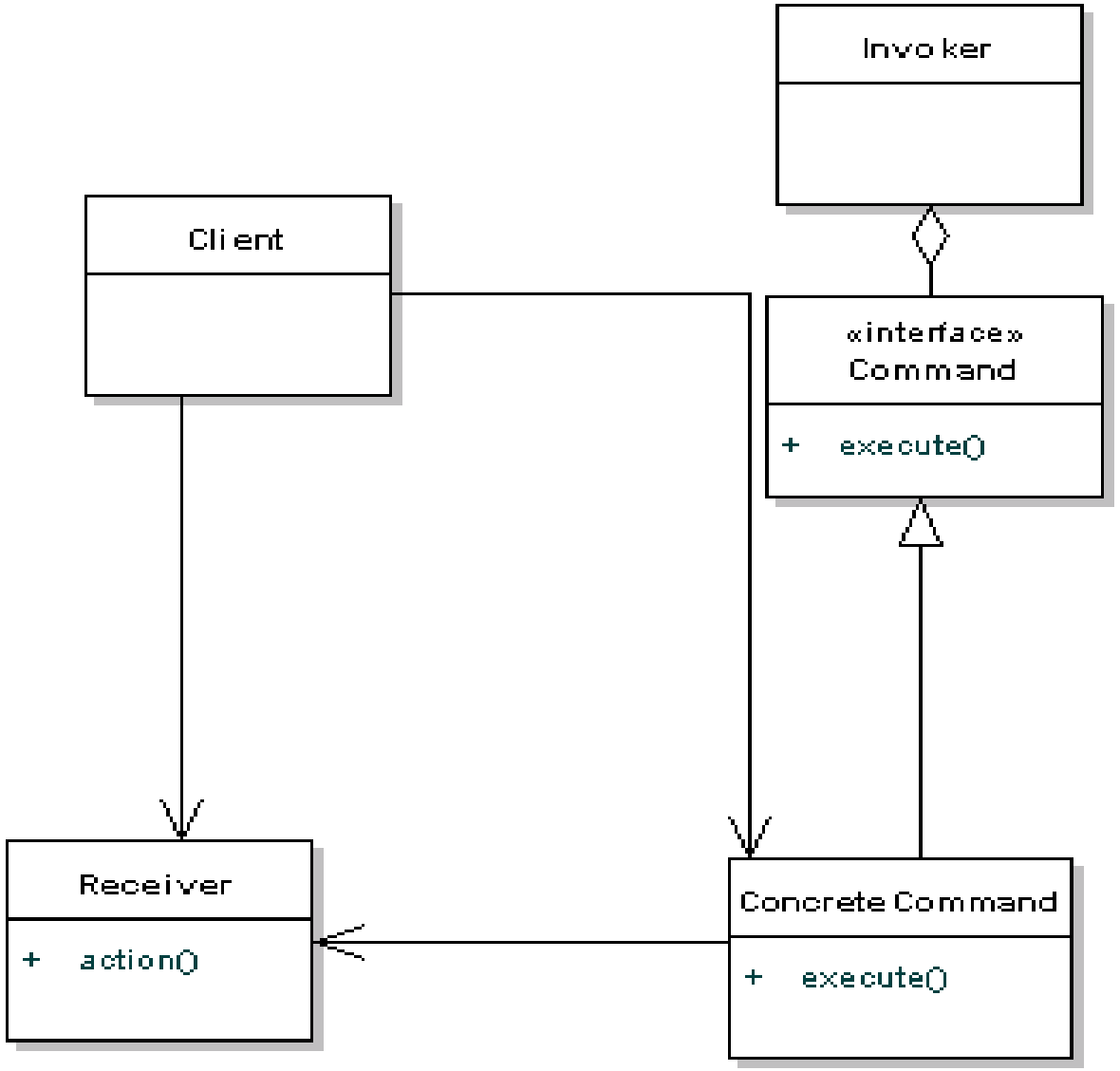
- creates a ConcreteCommand object and sets its receive

Invoker

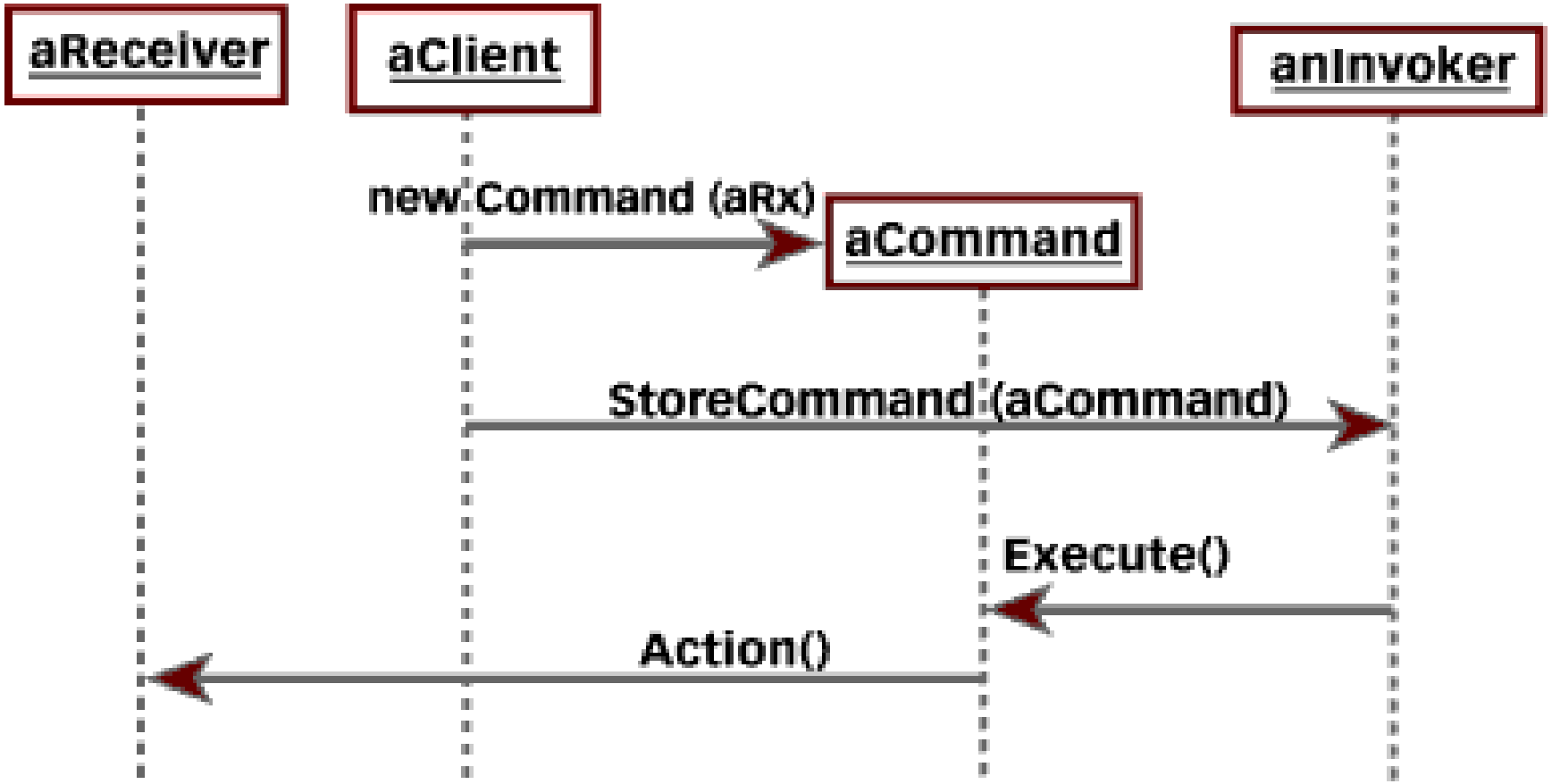
- asks the command to carry out the request;

Receiver

- knows how to perform the operations;



COMMAND


aReceiver
aClient
anInvoker
new Command (aRx)
aCommand
StoreCommand (aCommand)
Execute()
Action()

COMMAND. EXAMPLE

```
public void
actionPerformed(ActionEvent e)
{
  Object o = e.getSource();
  if (o = fileNewMenuItem)
    doFileNewAction();
  else if (o = fileOpenMenuItem)
    doFileOpenAction();
  else if (o =
fileOpenRecentMenuItem)
    doFileOpenRecentAction();
  else if (o = fileSaveMenuItem)
    doFileSaveAction();
  // and more ...
}
```

COMMAND. EXAMPLE

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

public class FileOpenMenuItem extends JMenuItem implements Command
{
  public void execute()
  {
    // your business logic goes here
  }
}
```

COMMAND. EXAMPLE

```
public void actionPerformed(ActionEvent e)
{
  Command command = (Command)e.getSource();
  command.execute();
}
```

COMMAND

Advantages

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- You can assemble commands into a composite command. In general, composite commands are an instance of the Composite pattern.
- It's easy to add new Commands, because you don't have to change existing classes.

Disadvantages

- Proliferation of little classes, that are more readable

COMMAND

- Java API examples
  - ActionListener
  - Comparator
  - Runnable / Thread

INTERPRETER

Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

Problem

- A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine"

INTERPRETER. EXAMPLE

- Language translation
  - SQL parsing
  - Symbol processing engine
  - Music
    - Grammar = musical notes
    - Interprets = musicians, playing the music
  - …

INTERPRETER

Context

- Contains information that is global to the interpreter.

AbstractExpression

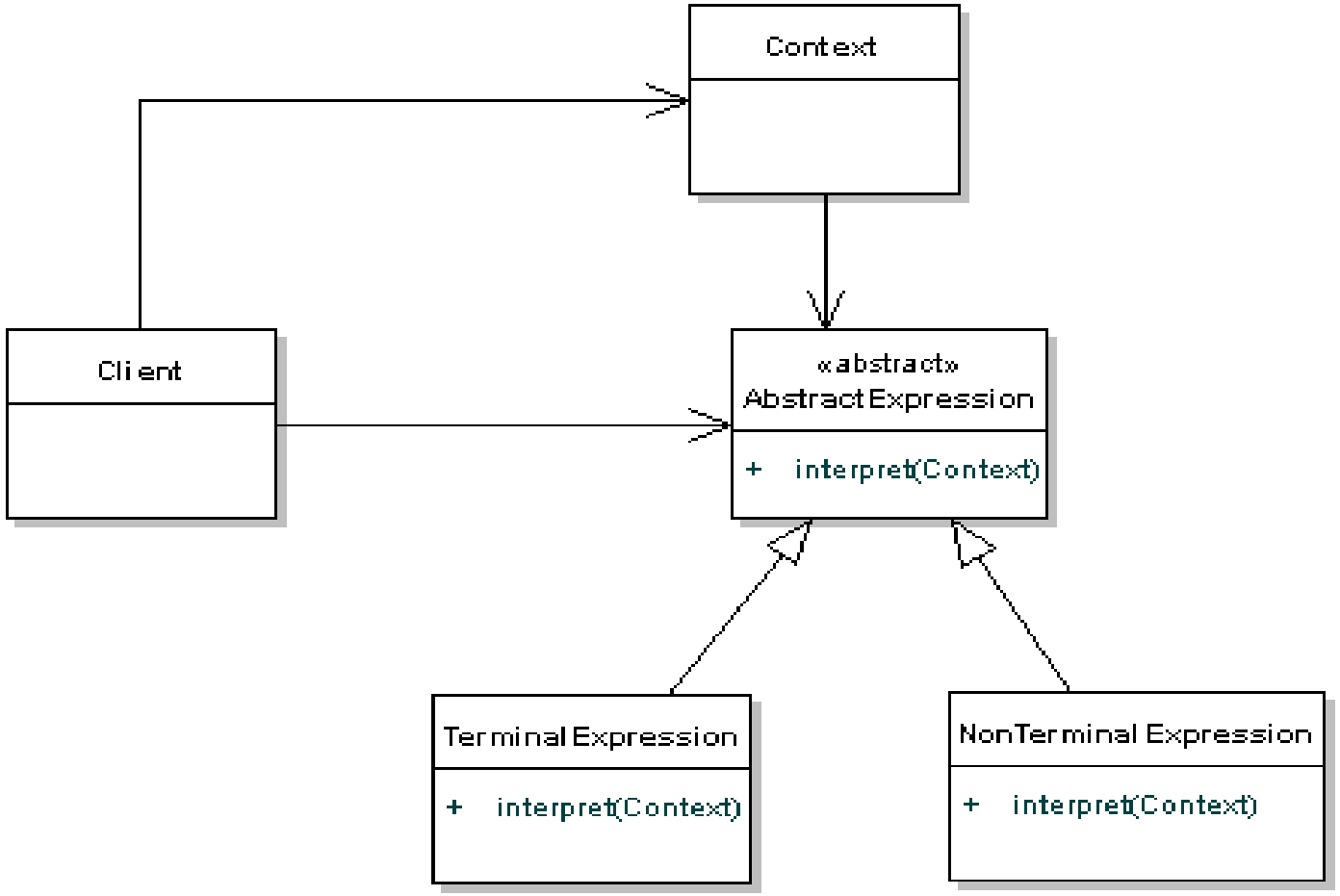
- Provides an interface for executing an operation.

TerminalExpression

- Implements the interpret interface associated with any terminal expressions in the defined grammar.

Client

- Either builds the Abstract Syntax Tree, or the AST is passed through to the client.
- An AST is composed of both TerminalExpressions and NonTerminalExpressions.
- The client will start using the interpreter



INTERPRETER

Example:

- Rule validation
  - Each expression is interpreted and the output for each expression is a boolean value.
  - Rule example:
    - John is male? true
    - Julie is a married women? true

INTERPRETER

```
//abstract expression
public interface Expression {
   public boolean interpret(String context);
}
```

```
//concrete expression
public class TerminalExpression implements Expression {
   private String data;
   public TerminalExpression(String data){
      this.data = data;
   }
   @Override
   public boolean interpret(String context) {
      if(context.contains(data)) return true;
      return false;
   }
}
```

INTERPRETER

```
public class OrExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public OrExpression(Expression expr1,
            Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) || expr2.interpret(context);
   }
}
```

```
public class AndExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public AndExpression(Expression expr1,
                  Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) && expr2.interpret(context);
   }
}
```

INTERPRETER

```
public class InterpreterPatternDemo {

public static void main(String[] args) {
      Expression isMale = getMaleExpression();
      Expression isMarriedWoman =
                        getMarriedWomanExpression();

      System.out.println("John is male? " +
                        isMale.interpret("John"));
      System.out.println("Julie is a married women? " +
                        isMarriedWoman.interpret("Married
Julie"));
   }
```

```
   //Rule: Robert and John are male
   public static Expression getMaleExpression(){
      Expression robert = new TerminalExpression("Robert");
      Expression john = new TerminalExpression("John");
      return new OrExpression(robert, john);
   }

   //Rule: Julie is a married women
   public static Expression getMarriedWomanExpression(){
      Expression julie = new TerminalExpression("Julie");
      Expression married =
                        new TerminalExpression("Married");
      return new AndExpression(julie, married);
   }
}
```

INTERPRETER

- Interpreter pattern can be used when we can create a syntax tree for a grammar.

- Interpreter pattern requires a lot of error checking and a lot of expressions and code to evaluate them, it gets complicated when the grammar becomes more complicated and hence hard to maintain and provide efficiency.

- java.util.Pattern and subclasses of java.text.Format are some of the examples of interpreter pattern used in JDK.

MEDIATOR

Intent

- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to "full object status".

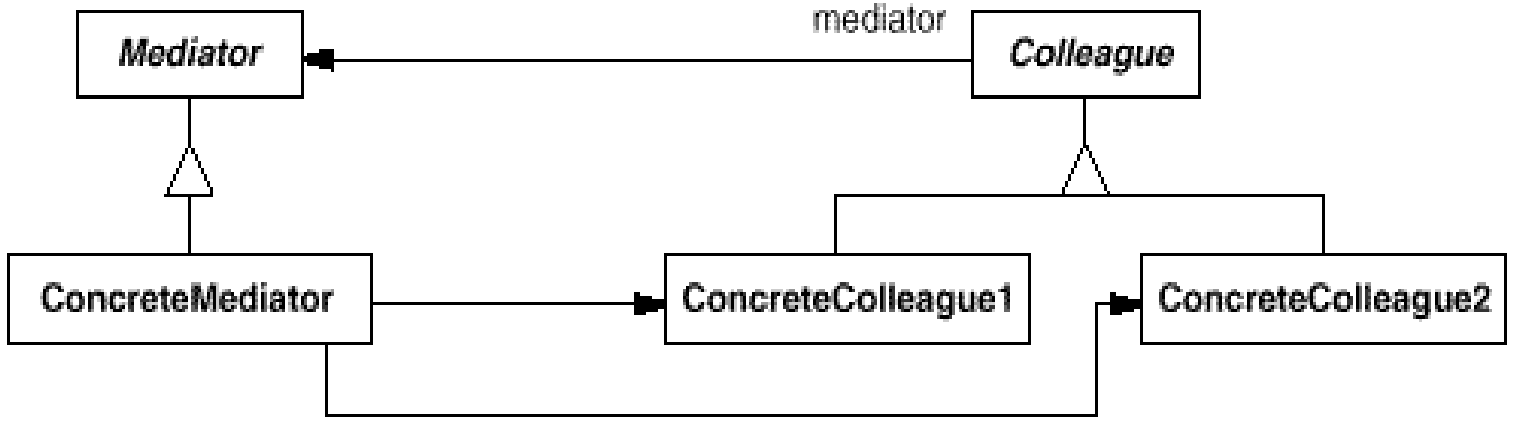
Problem

- We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

MEDIATOR. EXAMPLE

- GUI components
  - Dialog window is a collection of graphic and non-graphic controls
  - Dialog class provides the mechanism to facilitate the interaction between controls
- JMS (JAVA MESSAGE SERVICE)
  - Allows applications to subscribe and publish data to other applications
- Chat application
  - In a chat application we can have several participants
  - Not a good idea to connect each participant to all the other
  - Solution is to have a hub where all participants will connect

MEDIATOR



Mediator

- defines an interface for communicating with Colleague objects

ConcreteMediator

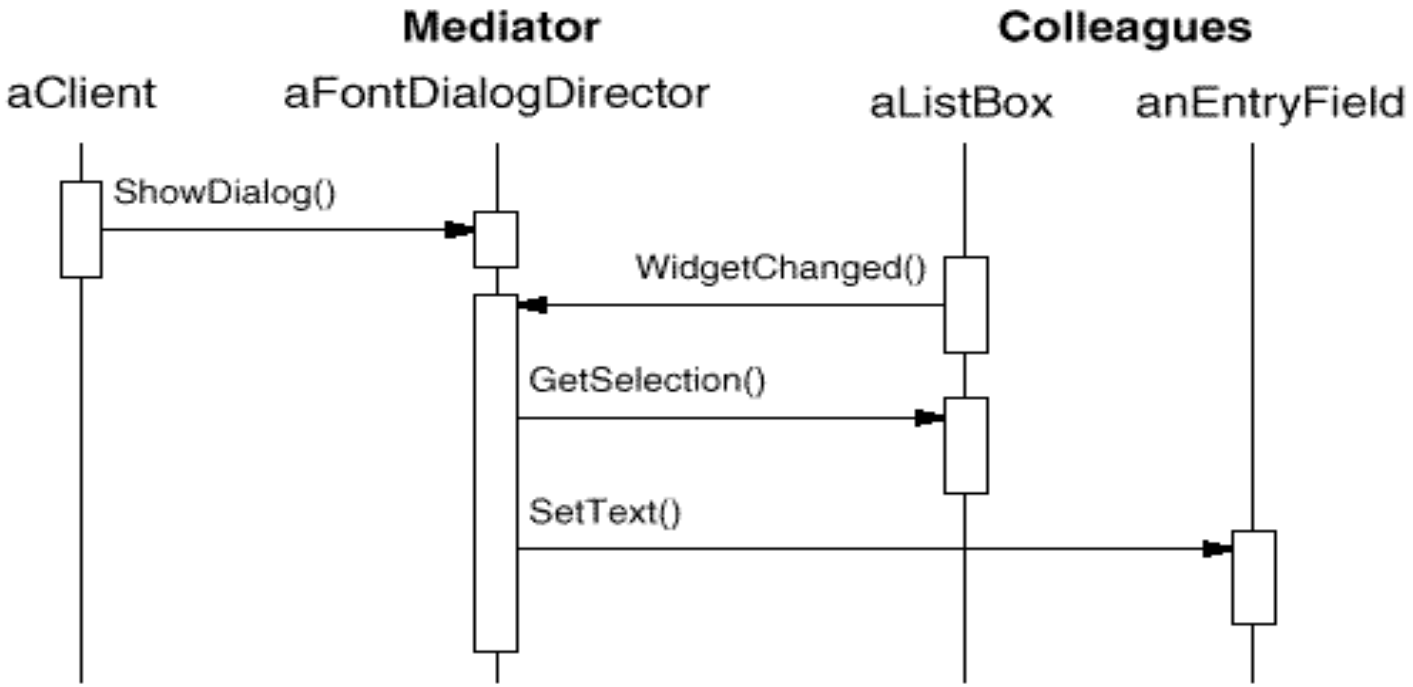
- knows and maintains its colleagues
- implements cooperative behavior by coordinating Colleagues

Colleague classes

- each Colleague class knows its Mediator object
- each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

MEDIATOR

Coolaboration diagram for the GUI components case



MEDIATOR

Example:

A chat between users

MEDIATOR

```
// Mediator interface

public interface ChatMediator {

    public void sendMessage(String msg, User user);

    void addUser(User user);
}
```

MEDIATOR

```
//colleague interface
public abstract class User {
    protected ChatMediator mediator;
    protected String name;

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

MEDIATOR

```
//concrete mediator
public class ChatMediatorImpl implements ChatMediator {

    private List<User> users;

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

    @Override
    public void addUser(User user){
        this.users.add(user);
    }

    @Override
        public void sendMessage(String msg, User user) {
            for(User u : this.users){
                //message should not be received
                //by the user sending it
                if(u != user){
                    u.receive(msg);
                }
            }
        }
    }
```

MEDIATOR

```
//concrete Collegue
public class UserImpl extends User {
     public UserImpl(ChatMediator med, String name) {
        super(med, name);
    }
     @Override
    public void send(String msg){
        System.out.println(this.name+": Sending Message="+msg);
        mediator.sendMessage(msg, this);
    }
    @Override
    public void receive(String msg) {
        System.out.println(this.name+": Received Message:"+msg);
    }
}
```

MEDIATOR

```
//pattern client
public class ChatClient {

    public static void main(String[] args) {
        ChatMediator mediator = new ChatMediatorImpl();
        User user1 = new UserImpl(mediator, "Pankaj");
        User user2 = new UserImpl(mediator, "Lisa");
        User user3 = new UserImpl(mediator, "Saurabh");
        User user4 = new UserImpl(mediator, "David");
        mediator.addUser(user1);
        mediator.addUser(user2);
        mediator.addUser(user3);
        mediator.addUser(user4);

        user1.send("Hi All");

    }
}
```

MEDIATOR

When to use mediator pattern?

- When one or more objects must interact with several different objects.
- When centralized control is desired
- When simple object need to communicate in complex ways.
- When you want to reuse an object that frequently interacts with other objects

MEDIATOR

Benefits

- Limits subclassing & specialization, allowing Colleague classes to be reused w/o any changes
- Decouples colleagues, which facilitates independent variations of the colleague and mediator classes.
- Simplifies protocol by replacing many-to-many interaction with one-to-one interaction
- Abstracts object behaviors & cooperation. This allows you to deal w/ an object's small-scale behavior separately form its interaction with other objects

Disadvantages

- Reducing the complexity of Colleagues increases the complexity of the Mediator itself. In some situations, maintaining a large Mediator may become as daunting a task as operating w/o a one

MEMENTO

Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.

Problem

- Need to restore an object back to its previous state

MEMENTO. EXAMPLES

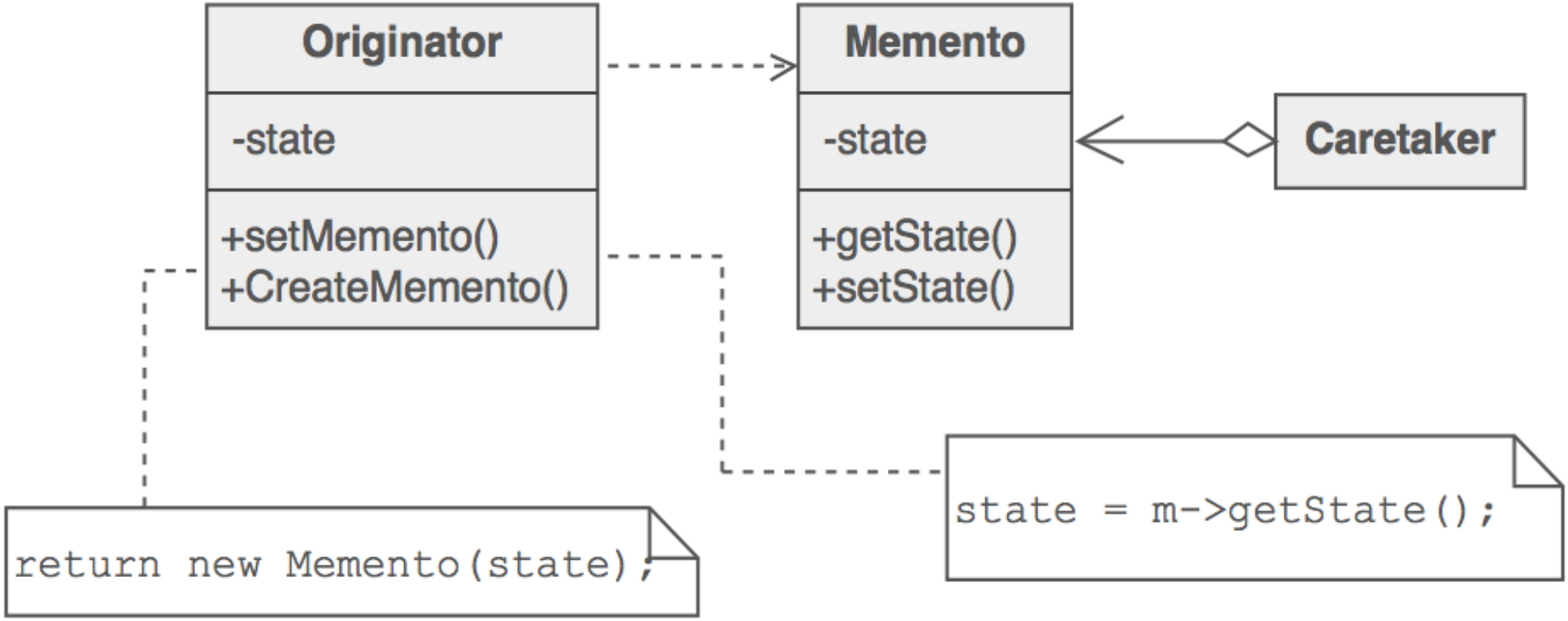
- Undo and restore operations in most software.

- Database transactions
  - A transaction can contain multiple operations on the database
  - Each operation can succeed or fail
  - A transaction guarantees that if all operations succeed, the transaction would commit and would be final
  - Rolling back mechanism uses the memento design pattern

- History

- Persistency; save / load state between executions of program

MEMENTO



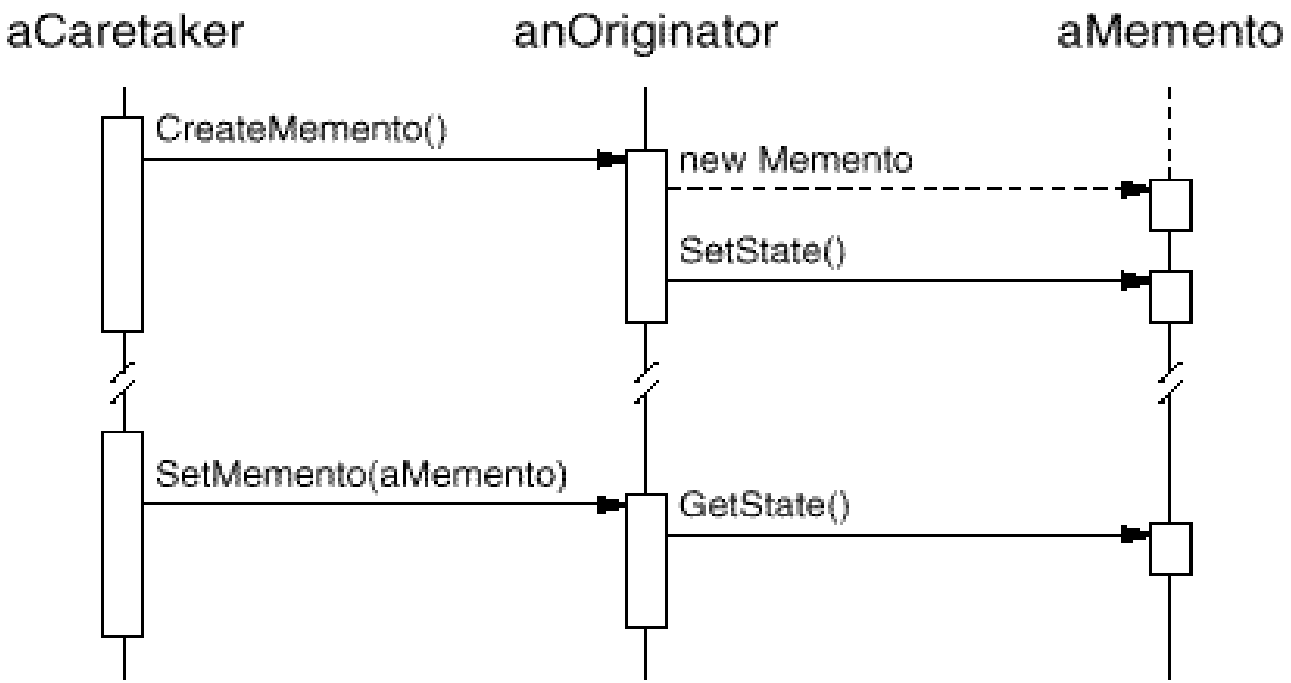
Originator - the object that knows how to save itself

- The state variable contains information that represents the state of the Originator object. This is the variable that will be saved and restored.
- The CreateMemento method is used to save the state of the Originator.
- The SetMemento method restores the Originator by accepting a Memento object, unpackage it, and sets its state variable using the state variable from the Memento

Caretaker - the object that knows why and when the Originator needs to save and restore itself.

Memento stores the historical information of the Originator. The information is stored in its state variable.

MEMENTO


aCaretaker
anOriginator
aMemento
CreateMemento()
new Memento
SetState()
SetMemento(aMemento)
GetState()

MEMENTO

Implementation example

- Class Diet
  - Person name
  - Numbers of days that remain until the diet finish
  - Person

MEMENTO

```
// originator - object whose state we want to save
public class DietInfo {
	String personName;
	int dayNumber;
	int weight;
	public DietInfo(String personName, int dayNumber, int weight) {
		this.personName = personName;
		this.dayNumber = dayNumber;
		this.weight = weight;
	}
	public String toString() {
		return "Name: " + personName + ", day number: " +
dayNumber + ", weight: " + weight;
	}
```

```
public void setDayNumberAndWeight(int dayNumber, int weight) {
		this.dayNumber = dayNumber;
		this.weight = weight;
}

public Memento save() {
		return new Memento(personName, dayNumber, weight);
}

public void restore(Object objMemento) {
		Memento memento = (Memento) objMemento;
		personName = memento.mementoPersonName;
		dayNumber = memento.mementoDayNumber;
		weight = memento.mementoWeight;
}
```

MEMENTO

```
// originator - object whose state we want to save
public class DietInfo {
        ……
        // memento - object that stores the saved state of the originator
        private class Memento {
                String mementoPersonName;
                int mementoDayNumber;
                int mementoWeight;
                public Memento(String personName, int dayNumber, int weight) {
                        mementoPersonName = personName;
                        mementoDayNumber = dayNumber;
                        mementoWeight = weight;
                }
        }}
```

MEMENTO

```
// caretaker - saves and restores a DietInfo object's state via a memento
// note that DietInfo.Memento isn't visible to the caretaker so we need to cast the memento to Object
public class DietInfoCaretaker {
        Object objMemento;

        public void saveState(DietInfo dietInfo) {
                objMemento = dietInfo.save();
        }

        public void restoreState(DietInfo dietInfo) {
                dietInfo.restore(objMemento);
        }
}
```

MEMENTO

```
public class MementoDemo {
	public static void main(String[] args) {
		// caretaker
		DietInfoCaretaker dietInfoCaretaker = new
DietInfoCaretaker();
		// originator
		DietInfo dietInfo = new DietInfo("Fred", 1, 100);
		System.out.println(dietInfo);

		dietInfo.setDayNumberAndWeight(2, 99);
		System.out.println(dietInfo);

			System.out.println("Saving state.");
			dietInfoCaretaker.saveState(dietInfo);

		dietInfo.setDayNumberAndWeight(3, 98);
		System.out.println(dietInfo);

		dietInfo.setDayNumberAndWeight(4, 97);
		System.out.println(dietInfo);

		System.out.println("Restoring saved state.");
		dietInfoCaretaker.restoreState(dietInfo);
		System.out.println(dietInfo);
	}
}
```

MEMENTO

Benefits

- Since object oriented programming dictates that objects should encapsulate their state it would violate this law if objects' internal variables were accessible to external objects. The memento pattern provides a way of recording the internal state of an object in a separate object without violating this law
- The memento eliminates the need for multiple creation of the same object for the sole purpose of saving its state.
- The memento simplifies the Originator since the responsibility of managing Memento storage is no longer centralized at the Originator but rather distributed among the Caretakers

Drawbacks

- The Memento object must provide two types of interfaces: a narrow interface to the Caretaker and a wide interface to the Originator. That is, it must acts like a black box to everything except for the class that created it.
- Using Mementos might be expensive if the Originator must store a large portion of its state information in the Memento or if the Caretakers constantly request and return the Mementos to the Originator.