

Ingineria Programării

Cursul 8 – 7 Aprilie
adiftene@infoiasi.ro

Cuprins

- ▶ Din Cursurile trecute...
 - Design Patterns (Creational Patterns, Structural Patterns)
- ▶ Behavioral Patterns
 - Chain of Responsibility
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - Observer
 - State
 - Strategy
 - Template Method
 - Visitor

Din Cursurile Trecute

- ▶ GOF: Creational Patterns, Structural Patterns, Behavioral Patterns
- ▶ Creational Patterns
- ▶ Structural Patterns

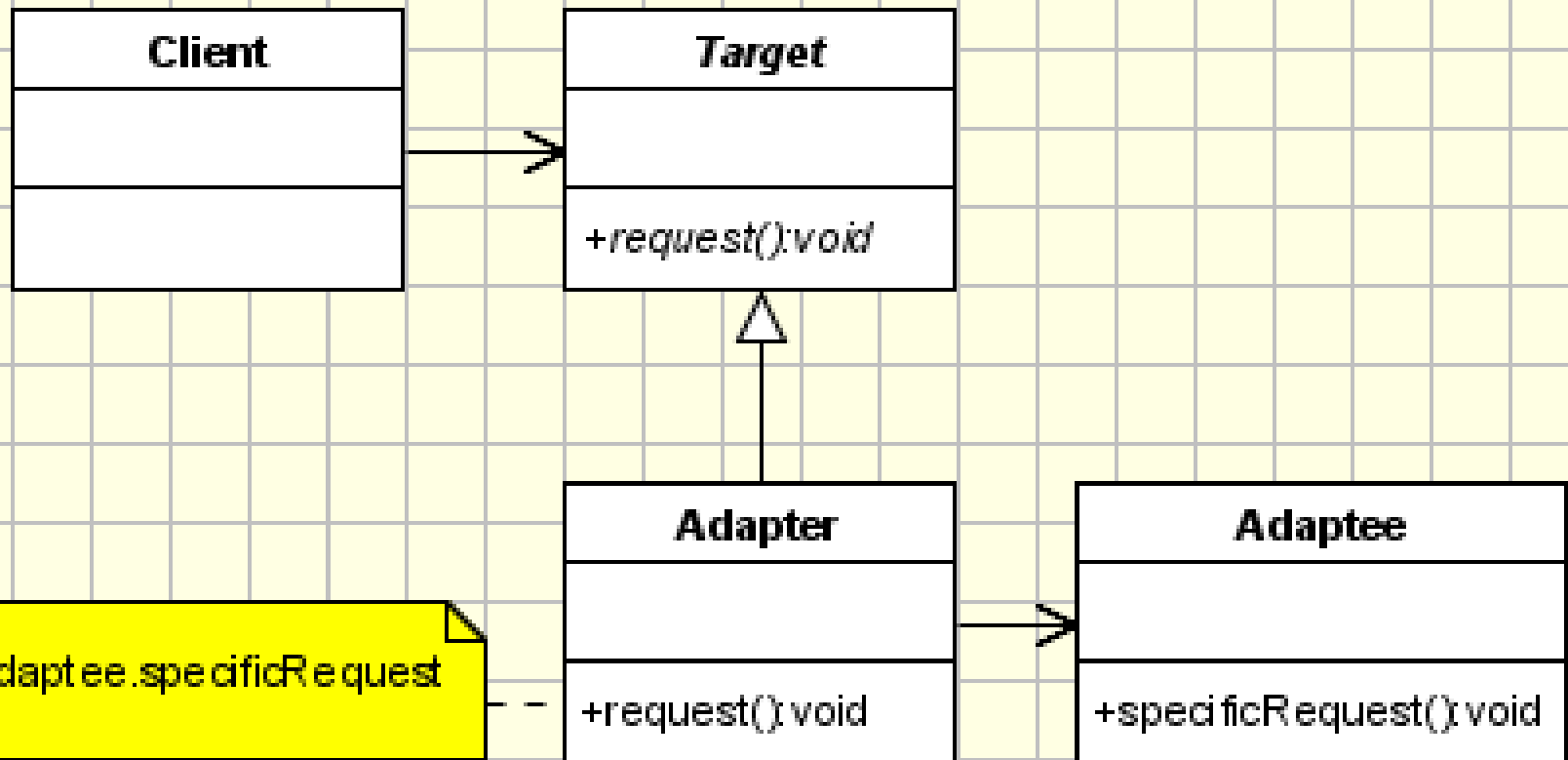
Din cursurile trecute – CP

- ▶ **Abstract Factory** – computer components
- ▶ **Builder** – children meal
- ▶ **Factory Method** – Hello <Mr/Ms>
- ▶ **Prototype** – Cell division
- ▶ **Singleton** – server log files

SP – Adapter

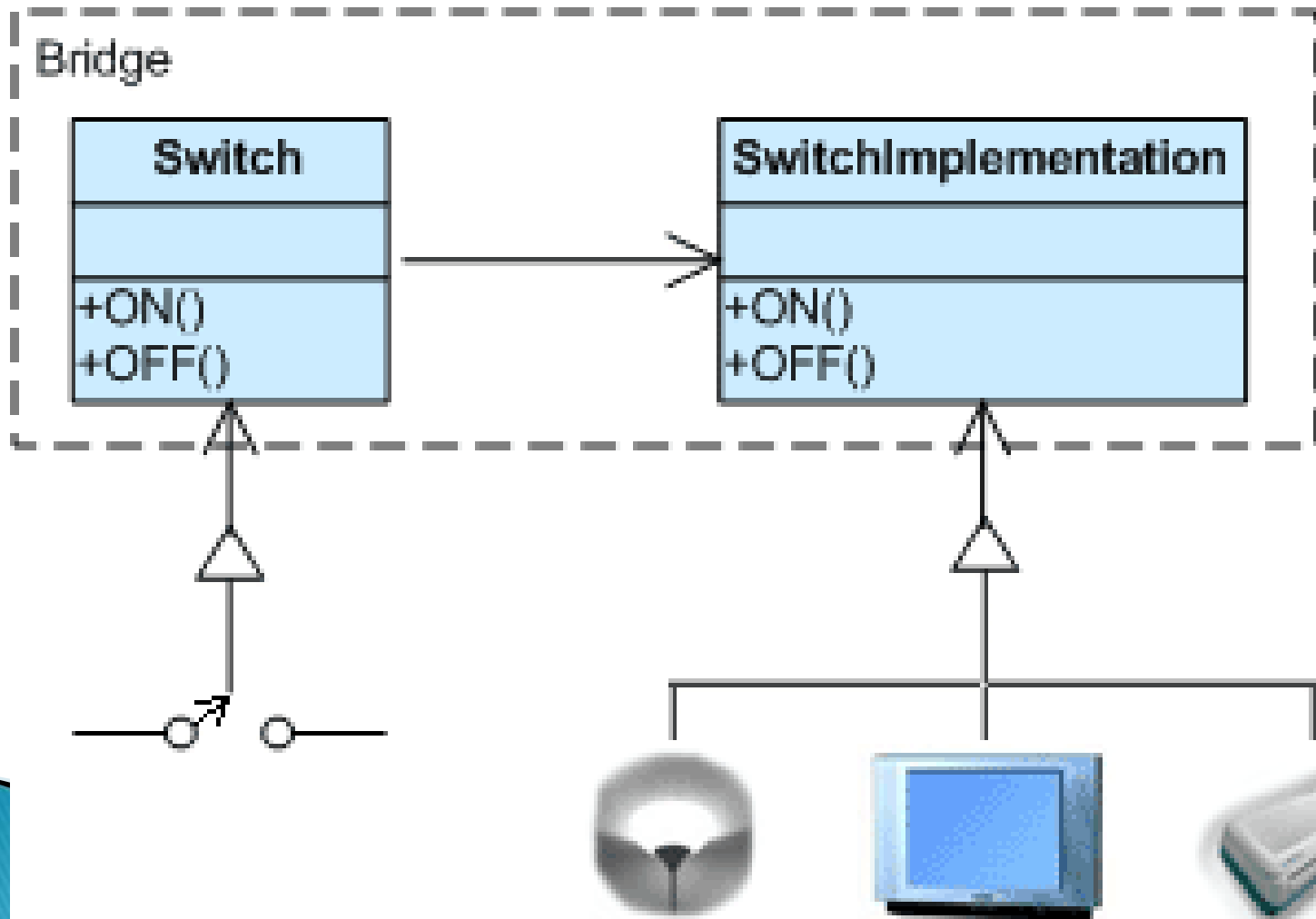
▶ Adapter – socket–plug

cd: Adapter Implementation - UML Class Diagram



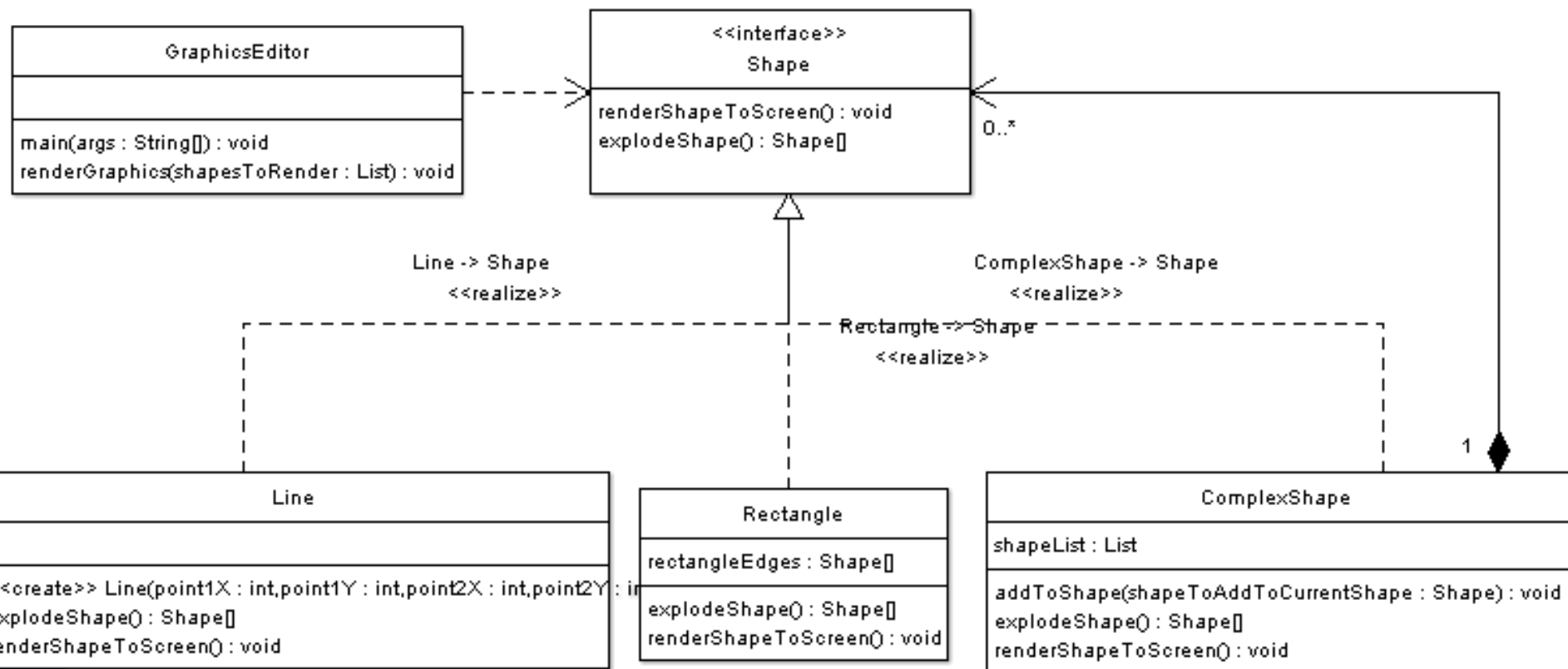
SP – Bridge

► Bridge – drawing API



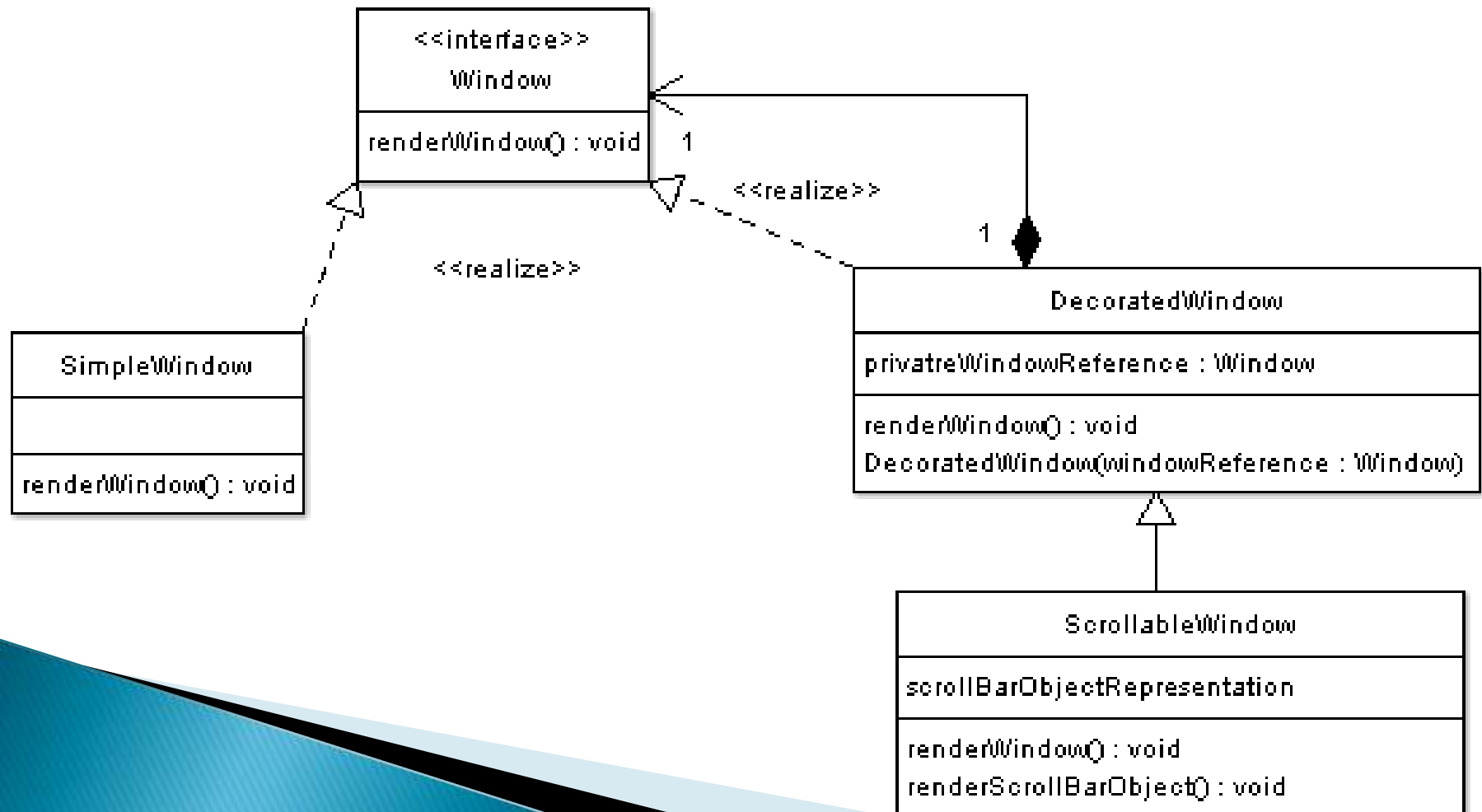
SP – Composite

► Composite – employee hierarchy



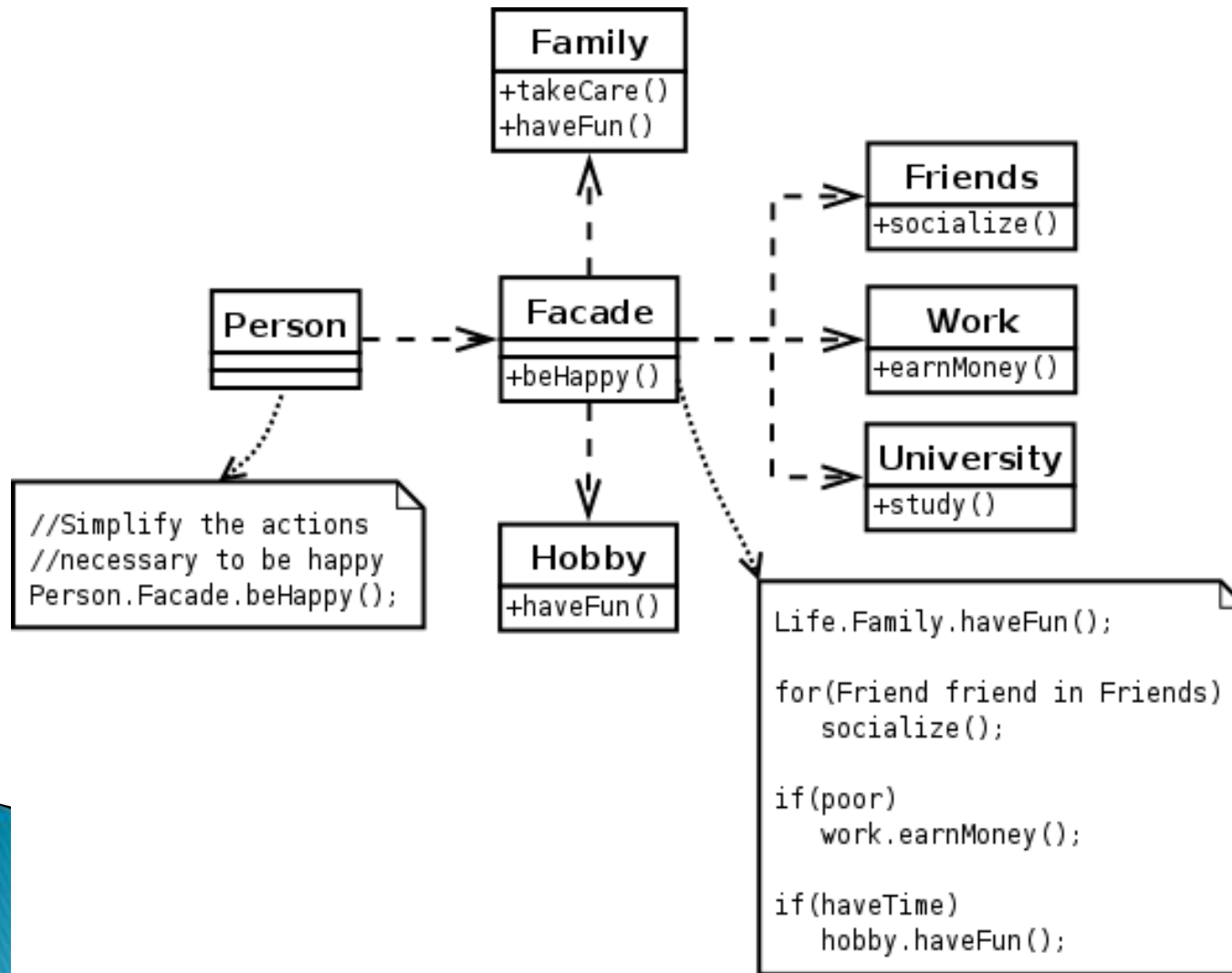
SP – Decorator

► Decorator – Christmas tree



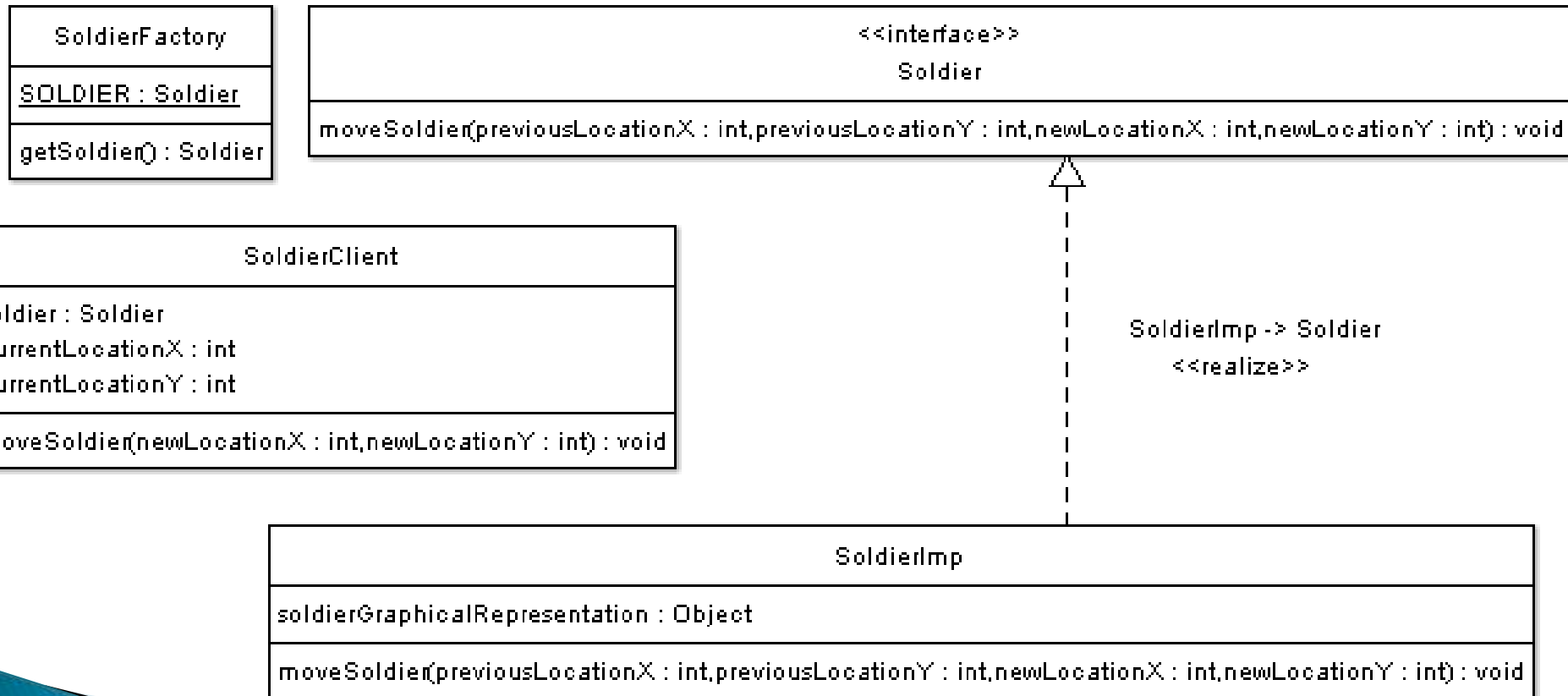
SP – Façade

► Façade – store keeper



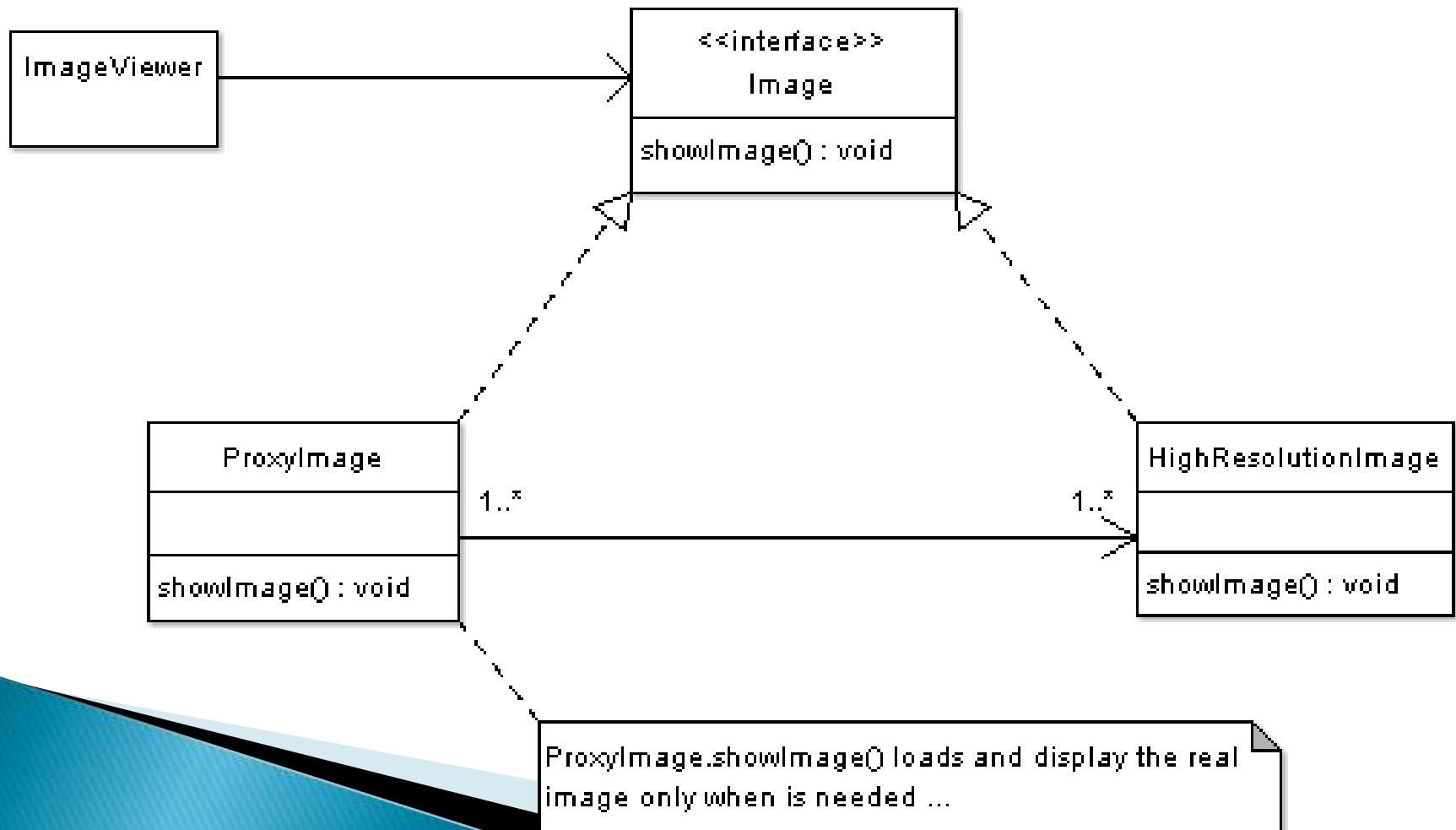
SP – Flyweight

► Flyweight – FontData



SP – Proxy

► Proxy – ATM access



Behavioral Patterns 1

- ▶ Behavioral patterns are concerned with **algorithms and the assignment of responsibilities between objects**
- ▶ These patterns **characterize complex control flow** that's difficult to follow at run-time
- ▶ They shift your focus away from flow of control to let you **concentrate just on the way objects are interconnected**

Behavioral Patterns 2

- ▶ **Encapsulating variation** is a theme of many behavioral patterns
- ▶ When an **aspect of a program changes frequently**, these patterns define an object that encapsulates that aspect
- ▶ Then other **parts of the program can collaborate with the object** whenever they depend on that aspect

Behavioral Patterns 3

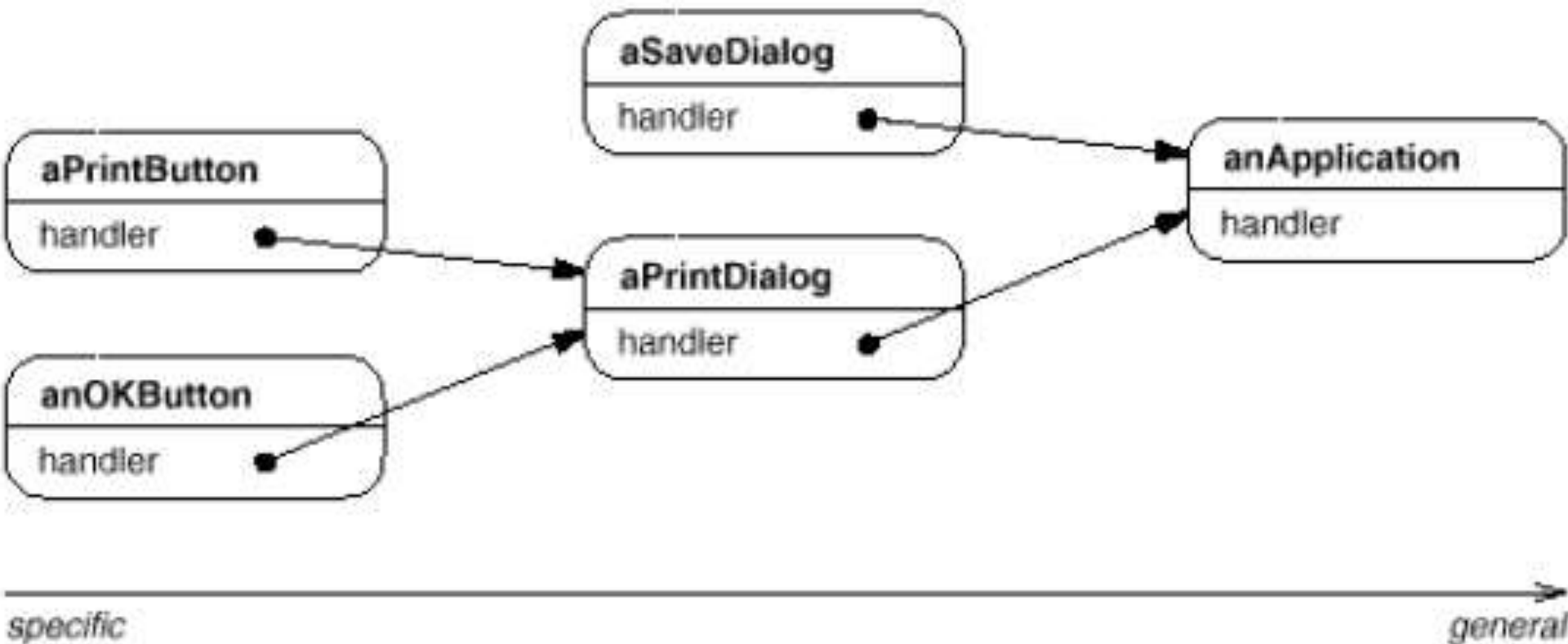
- ▶ These patterns describe **aspects of a program that are likely to change**
- ▶ Most patterns have two kinds of objects:
 - the **new object(s)** that encapsulate the aspect,
 - and the **existing object(s)** that use the new ones
- ▶ Usually the **functionality of new objects would be an integral part of the existing objects** were it not for the pattern

Chain of Responsibility

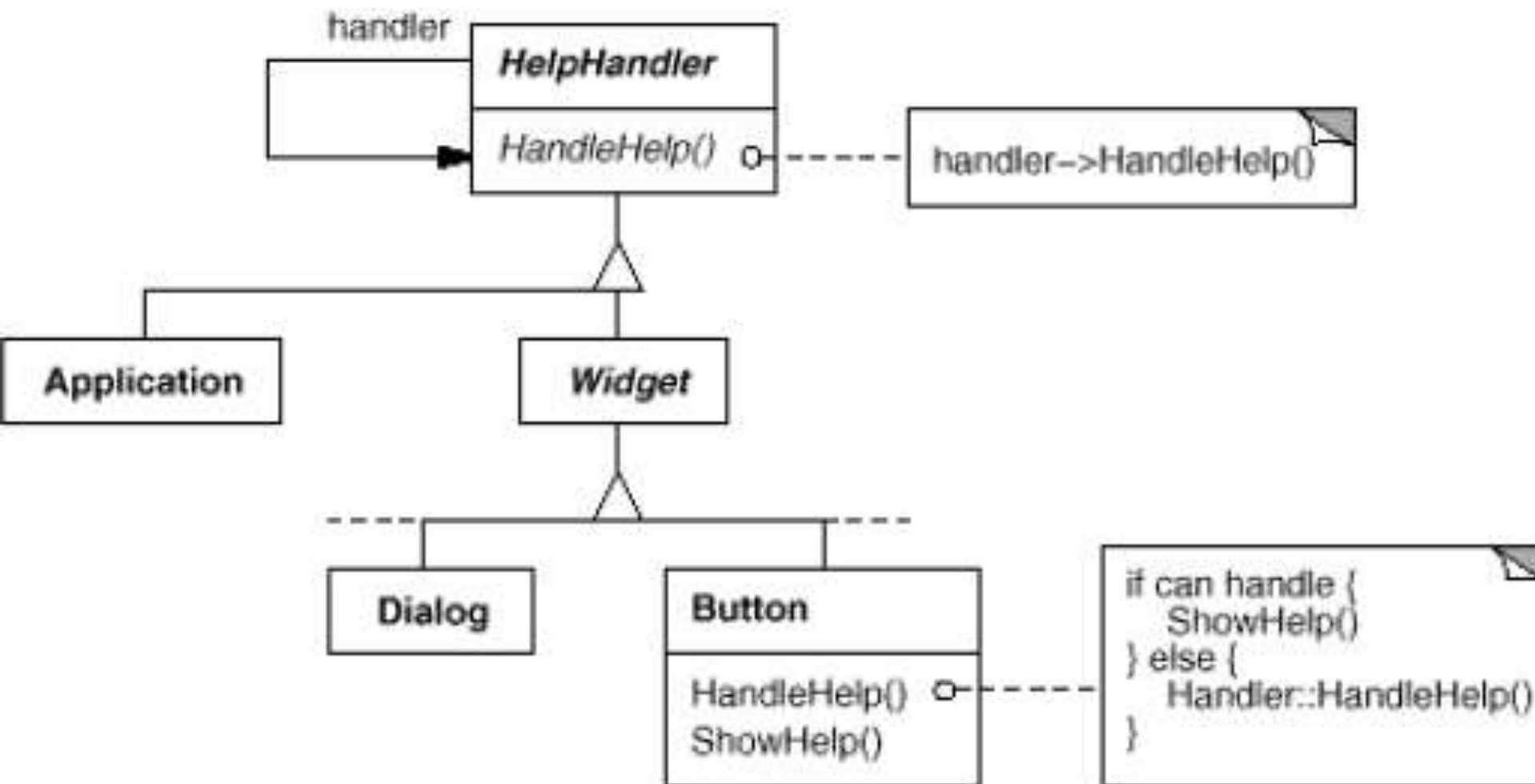
- ▶ **Intent** – Chain the receiving objects and **pass the request along the chain until an object handles it**
- ▶ **Motivation** – Consider a context-sensitive help facility for a graphical user interface. The help that's provided depends on the part of the interface that's selected and its context. If no specific help information exists for that part of the interface, then the help system should display a more general help message about the immediate context

Chain of Responsibility – Idea

- ▶ It's natural to organize help information from the most specific to the most general



Chain of Responsibility – Structure



Chain of Responsibility

- ▶ **Applicability** – Use this pattern when
 - more than one object may handle a request, and the handler isn't known *a priori*
 - you want to issue a request to one of several objects without specifying the receiver explicitly
 - the set of objects that can handle a request should be specified dynamically

Chain of Responsibility – Example

- ▶ Suppose, we have a multi level filter and gravel of different sizes and shapes. We need to filter this gravel of different sizes to approx size categories
- ▶ We will put the gravel on the multi-level filtration unit, with the filter of maximum size at the top and then the sizes descending. The gravel with the maximum sizes will stay on the first one and rest will pass, again this cycle will repeat until, the finest of the gravel is filtered and is collected in the sill below the filters
- ▶ Each of the filters will have the sizes of gravel which cannot pass through it. And hence, we will have approx similar sizes of gravels grouped

Chain of Responsibility – Java 1

```
public class Matter {  
    private int size;  
    private int quantity;  
  
    public int getSize() {return size;}  
  
    public void setSize(int size) {this.size = size;}  
  
    public int getQuantity() {return quantity;}  
  
    public void setQuantity(int quantity) {  
        this.quantity = quantity;  
    }  
}
```

Chain of Responsibility – Java 2

```
public class Sill {  
    public void collect(Matter gravel) {}  
}
```

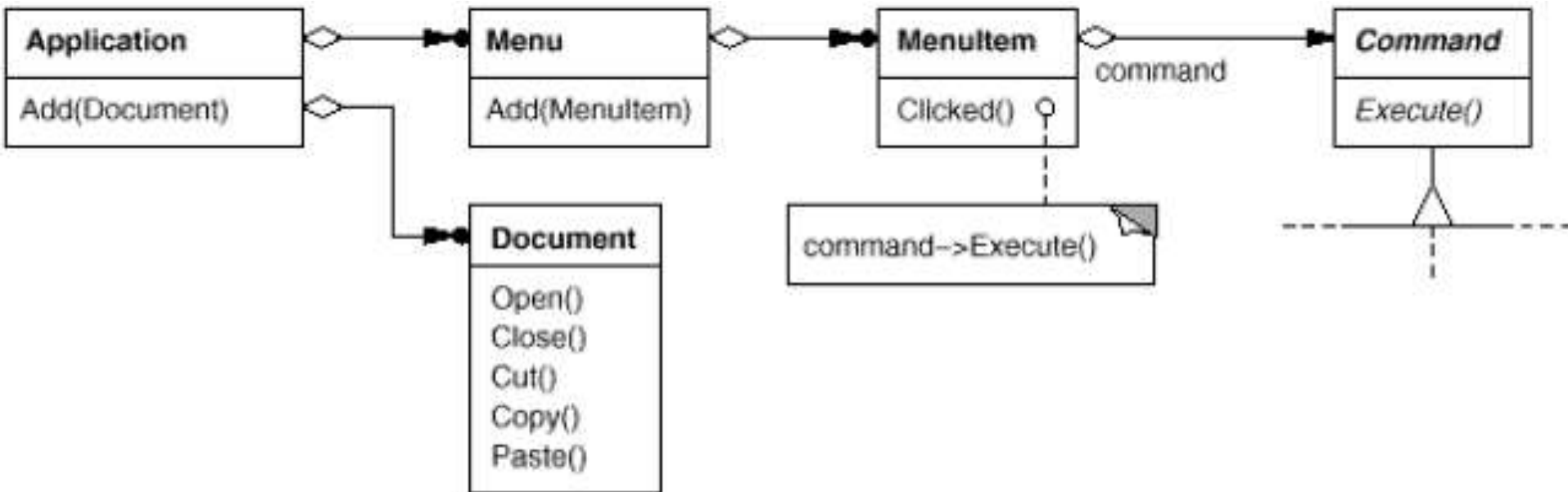
```
public class Filter1 extends Sill {  
    private int size;  
  
    public Filter1(int size) {this.size = size;}  
  
    public void collect(Matter gravel) {  
        for(int i = 0; i < gravel.getQuantity(); i++) {  
            if(gravel.getSize() < size) {  
                super.collect(gravel);  
            }  
            else {  
                //collect here. that means, only matter with less size will  
                pass  
            }  
        }  
    }  
}
```

Command

- ▶ **Intent – Encapsulate a request as an object**, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- ▶ **Also Known As – Action, Transaction**
- ▶ **Motivation – Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request**
- ▶ For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input. **But the toolkit can't implement the request explicitly in the button or menu**, because only applications that use the toolkit know what should be done on which object

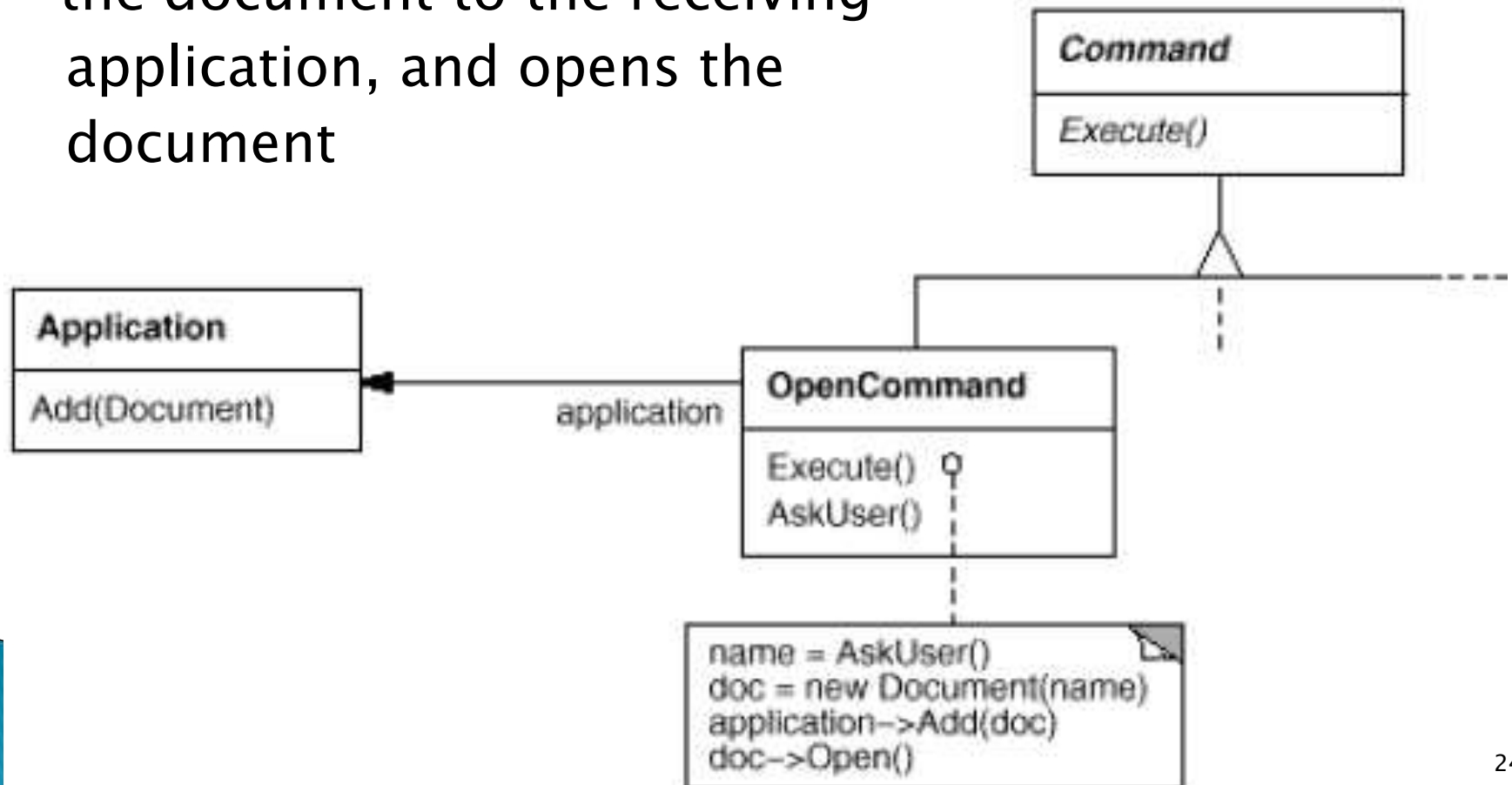
Command 2

- ▶ The key to this pattern is an abstract Command class, which declares an interface for executing operations



Command – Structure

- ▶ OpenCommand prompts the user for a document name, creates a corresponding Document object, adds the document to the receiving application, and opens the document



Command – Example



- ▶ A classic example of this pattern is a restaurant:
 - A **customer** goes to restaurant and orders the food according to his/her choice
 - The **waiter/ waitress** takes the order (command, in this case) and hands it to the *cook* in the kitchen
 - The **cook** can make several types of food and so, he/she prepares the ordered item and hands it over to the *waiter/waitress* who in turn serves to the *customer*

Command – Java 1

```
public class Order {  
    private String command;  
    public Order(String command) {  
        this.command = command;  
    }  
}
```

```
public class Waiter {  
    public Food takeOrder(Customer cust, Order  
    order) {  
        Cook cook = new Cook();  
        Food food = cook.prepareOrder(order, this);  
        return food;  
    }  
}
```

Command – Java 2

```
public class Cook {  
    public Food prepareOrder(Order order, Waiter  
        waiter) {  
        Food food = getCookedFood(order);  
        return food;  
    }  
    public Food getCookedFood(Order order) {  
        Food food = new Food(order);  
        return food;  
    }  
}
```

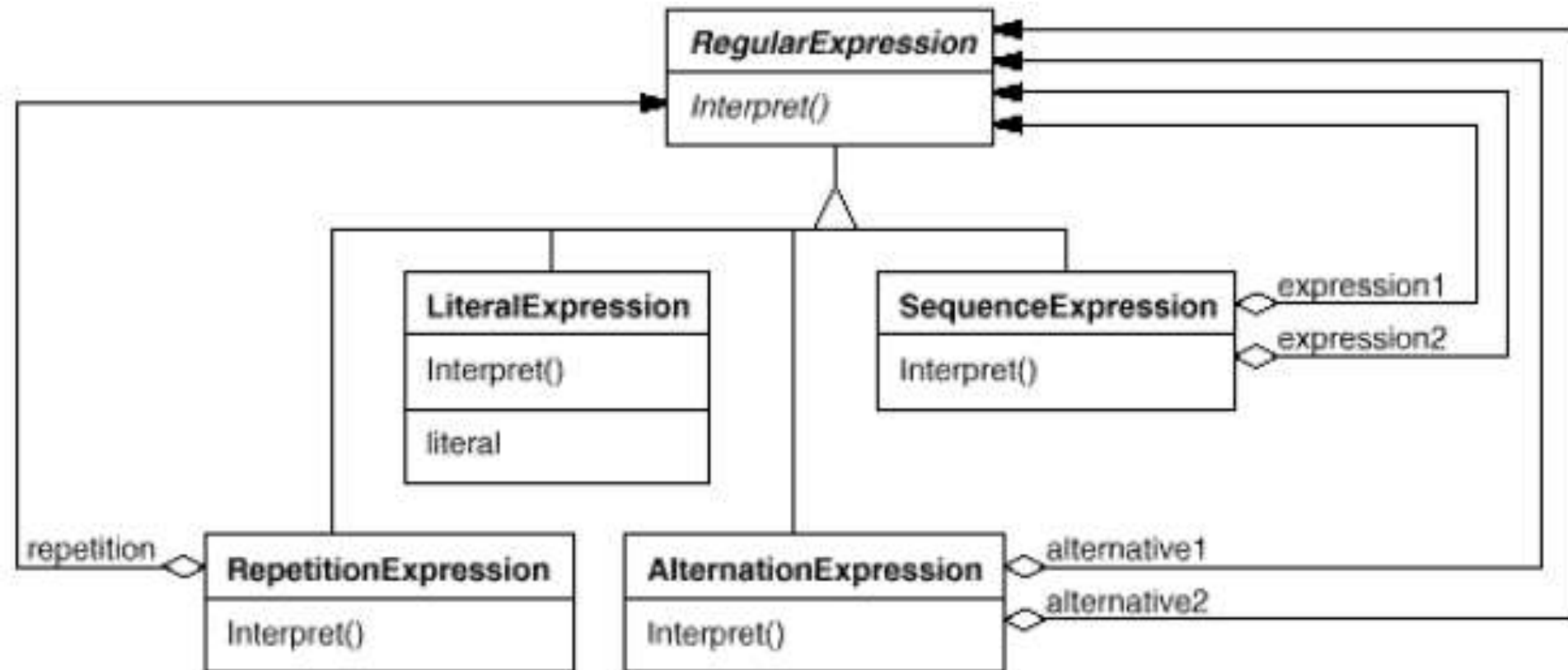
Interpreter

- ▶ **Intent** – Given a language, define a representation for its grammar along with an interpreter
- ▶ **Motivation** – If a particular kind of problem occurs **often enough**, then it might be worthwhile to **express instances of the problem as sentences in a simple language**. Then you can build an **interpreter** that solves the problem by interpreting these sentences.
- ▶ For example, *searching for strings that match a pattern is a common problem*. Regular expressions are a standard language for specifying patterns of strings

Interpreter – Grammar

- ▶ Suppose the following grammar defines the regular expressions:
 - $\text{expression} ::= \text{literal} \mid \text{alternation} \mid \text{sequence} \mid \text{repetition} \mid '(' \text{ expression } ')'$
 - $\text{alternation} ::= \text{expression} '|' \text{ expression}$
 - $\text{sequence} ::= \text{expression} \& \text{ expression}$
 - $\text{repetition} ::= \text{expression} '^'$
 - $\text{literal} ::= 'a' \mid 'b' \mid 'c' \mid \dots \{ 'a' \mid 'b' \mid 'c' \mid \dots \}^*$

Interpreter – Regular expressions



Interpreter – Example



- ▶ The “musical notes” is an “Interpreted Language”. The musicians read the notes, interpret them according to “Sa, Re, Ga, Ma...” or “Do, Re, Me...” etc. and play the instruments, what we get in output is musical sound waves. Think of a program which can take the Sa, Re, Ga, Ma etc. and produce the sounds for the frequencies.
- ▶ For Sa, the frequency is 256 Hz, similarly, for Re, it is 288Hz and for Ga, it is 320 Hz etc...
- ▶ We can have it at one of the two places, one is a constants file, “token=value” and the other one being in a properties file

Interpreter – Java 1

- ▶ *MusicalNotes.properties*

Sa=256

Re=288

Ga=320

```
public class NotesInterpreter {  
    private Note note;  
    public void getNoteFromKeys(Note note) {  
        Frequency freq = getFrequency(note);  
        sendNote(freq);  
    }  
  
    private Frequency getFrequency(Note note) {  
        // Get the frequency from properties file using ResourceBundle  
        // and return it.  
        return freq;  
    }  
  
    private void sendNote(Frequency freq) {  
        NotesProducer producer = new NotesProducer();  
        producer.playSound(freq);  
    }  
}
```

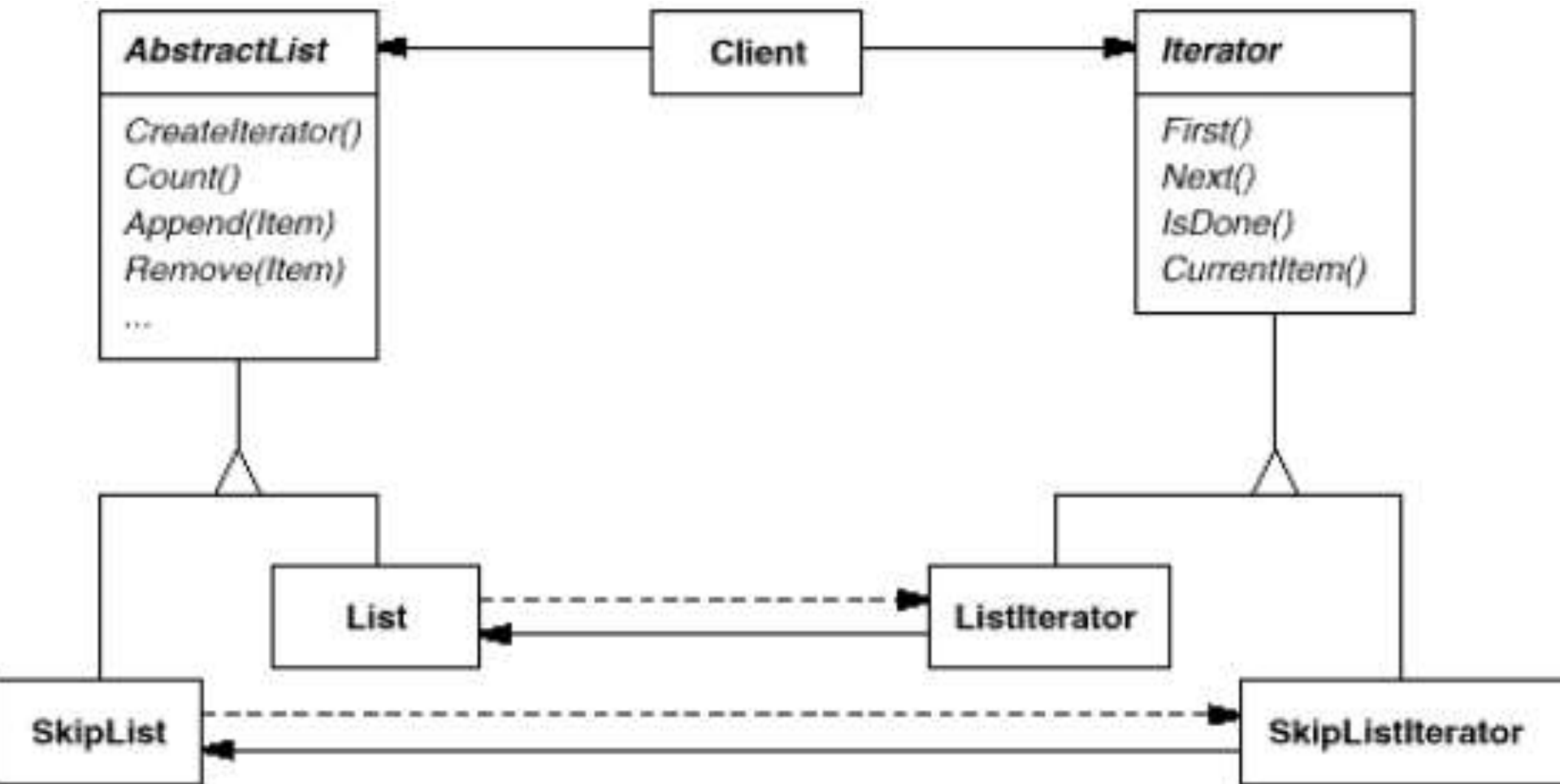

Interpreter – Java 2

```
public class NotesProducer {  
  
    private Frequency freq;  
  
    public NotesProducer() {  
        this.freq = freq;  
    }  
  
    public void playSound(Frequency freq) {  
    }  
}
```

Iterator

- ▶ **Intent** – Provide a way to access the elements of an aggregate object sequentially
- ▶ **Also Known As** – Cursor
- ▶ **Motivation** – *An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.*
Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish

Iterator – Structure



Iterator – Applicability

- ▶ to access an aggregate object's contents without exposing its internal representation
- ▶ to support multiple traversals of aggregate objects
- ▶ to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)

Iterator – Example

- ▶ For example, remote control of TV. Any remote control we use, either at home/hotel or at a friend's place, we just pick up the TV remote control and start pressing Up and Down or Forward and Back keys to iterate through the channels



Iterator – Java 1

```
public interface Iterator {  
    public Channel nextChannel(int currentChannel);  
    public Channel prevChannel(int currentChannel);  
}
```

```
public ChannelSurfer implements Iterator {  
    public Channel nextChannel (int currentChannel) {  
        Channel channel = new Channel(currentChannel+1);  
        return channel;  
    }  
    public Channel prevChannel (int currentChannel) {  
        Channel channel = new Channel(currentChannel-1);  
        return channel;  
    }  
}
```

Iterator – Java 2

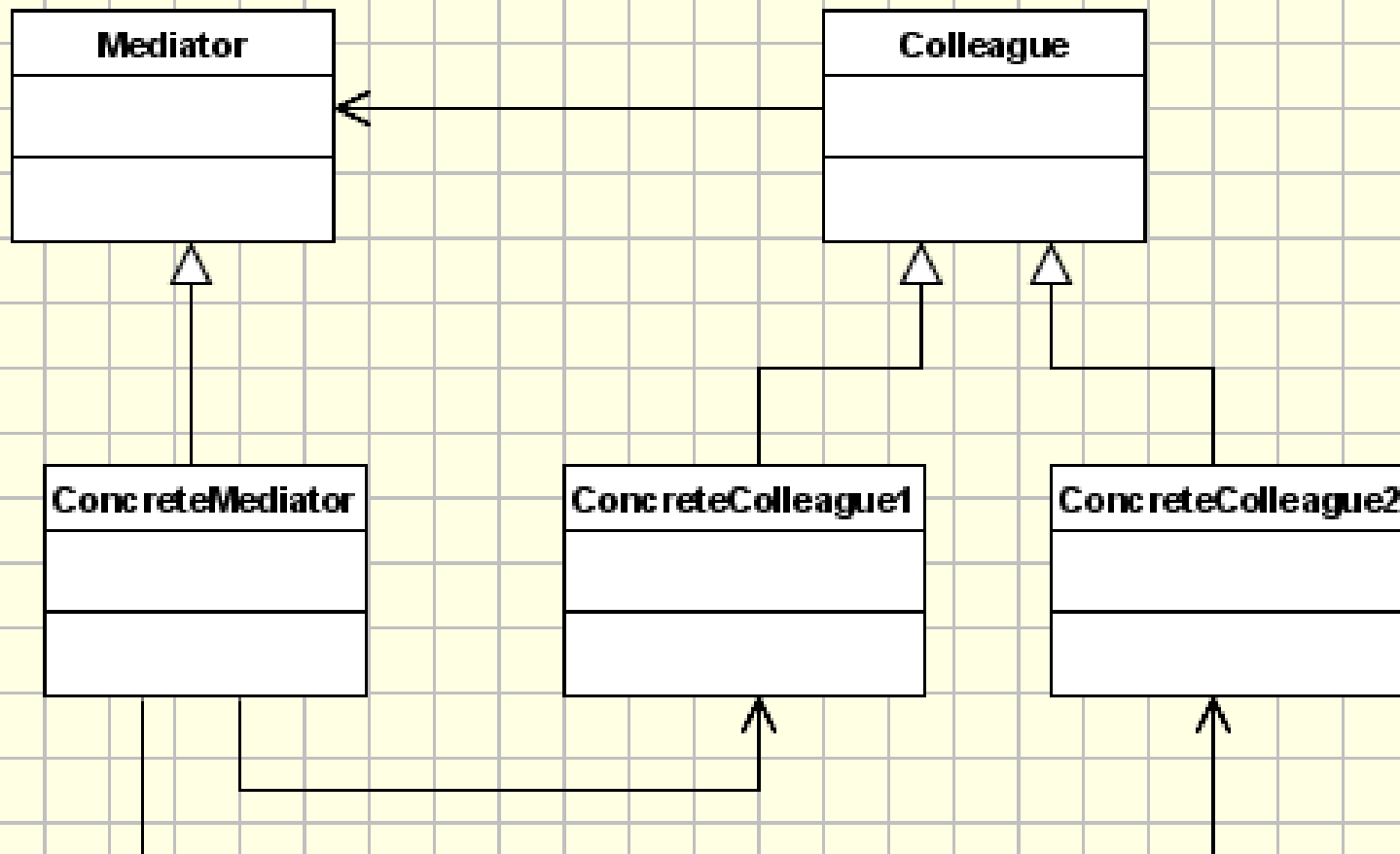
```
public class RemoteControl {  
    private ChannelSurfer surfer;  
    private Settings settings;  
  
    public RemoteControl() {  
        surfer = new ChannelSurfer();  
        settings = new Settings();  
    }  
    public getProgram(ChannelSurfer surfer) {  
        return new Program(surfer.nextChannel());  
    }  
}
```

Mediator

- ▶ **Intent** – Define an object that encapsulates how a set of objects interact
- ▶ **Motivation** – Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other

Mediator – Structure

cd: Mediator Implementation - UML Class Diagram



Mediator – Applicability

- ▶ According to (Gamma et al), the Mediator pattern should be used when:
 - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
 - reusing an object is difficult because it refers to and communicates with many other objects.
 - a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Mediator – Examples

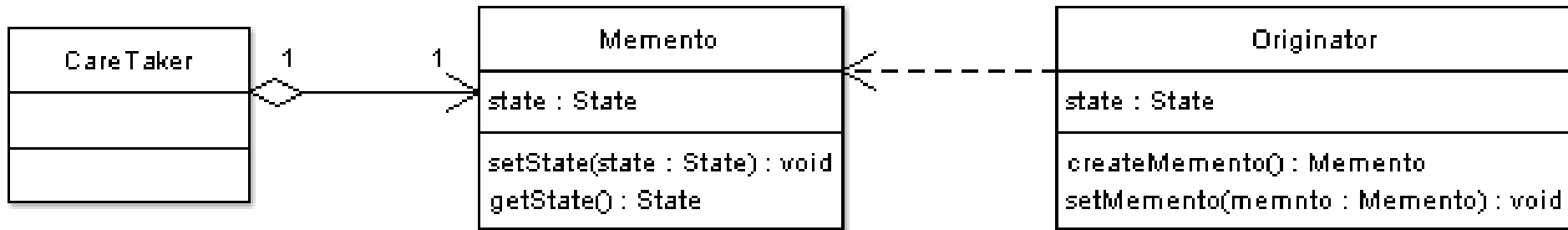
- ▶ A very common example can be airplanes interacting with the control tower and not among themselves
- ▶ Another popular example is Stock exchange
- ▶ The chat application is another example of the mediator pattern
- ▶ Other examples?



Memento

- ▶ **Intent** – Without violating encapsulation, capture and externalize an object's internal state so that the **object can be restored to this state later**
- ▶ **Also Known As** – Token
- ▶ **Motivation** – Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors. **You must save state information somewhere so that you can restore objects to their previous states**

Memento – Structure



▶ Memento

- Stores internal state of the Originator object
- Allows the originator to restore previous state

▶ Originator

- Creates a memento object capturing it's internal state
- Use the memento object to restore its previous state.

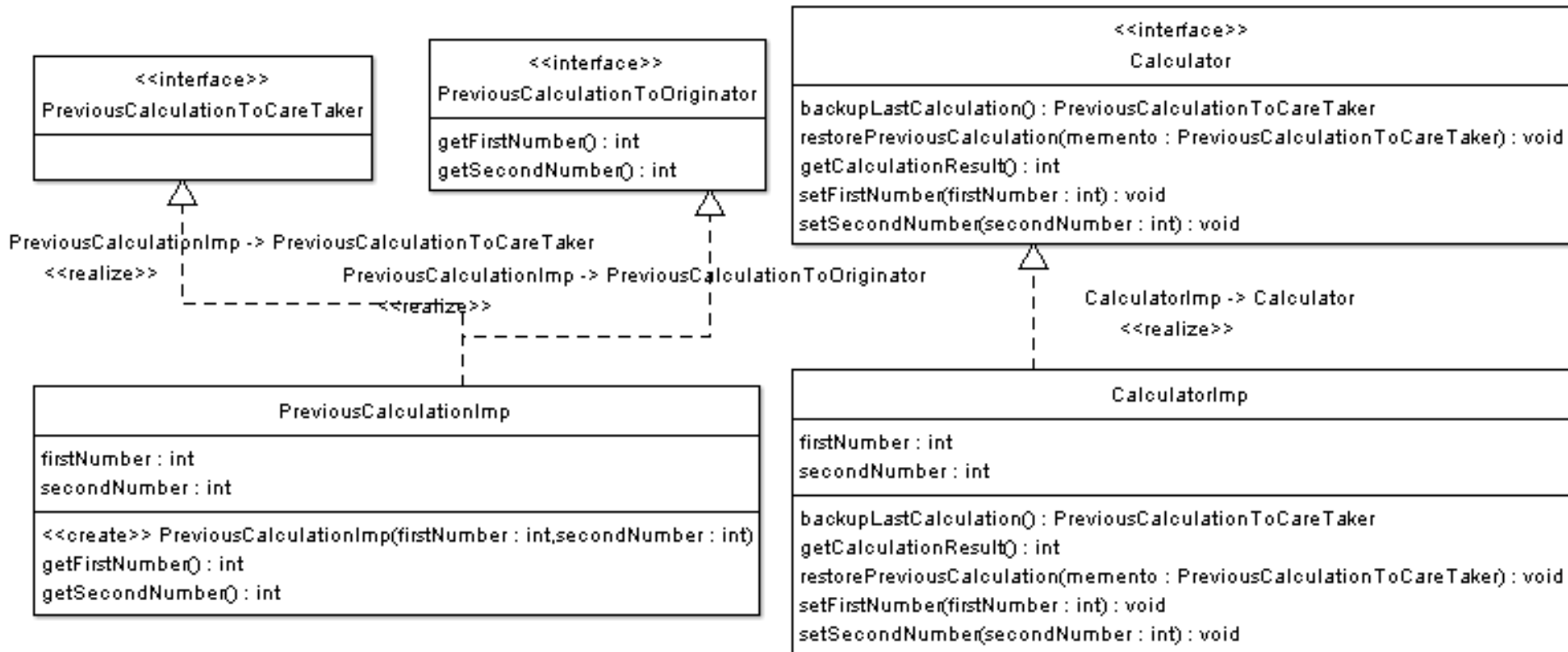
▶ Caretaker

- Responsible for keeping the memento.

The memento is opaque to the caretaker, and the caretaker must not operate on it.

Memento – Example

► Simple Calculator with Undo Operation



Memento – Database Transactions

- ▶ Transactions are operations on the database that occur in an atomic, consistent, durable, and isolated fashion
- ▶ If all operations succeed, the transaction would commit and would be final
- ▶ And if any operation fails, then the transaction would fail and all operations would rollback and leave the database as if nothing has happened
- ▶ This mechanism of rolling back uses the memento design pattern



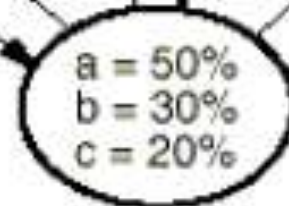
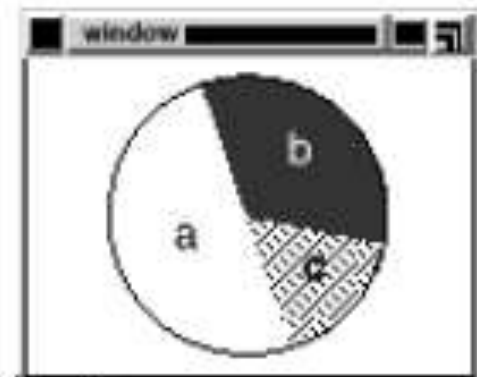
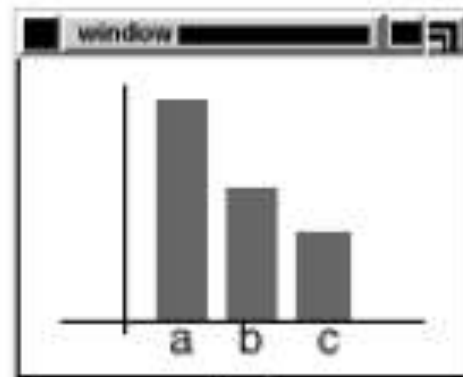
Observer

- ▶ **Intent** – Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- ▶ **Also Known As** – Dependents, Publish-Subscribe
- ▶ **Motivation** – A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects

Observer – Example

observers

	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10



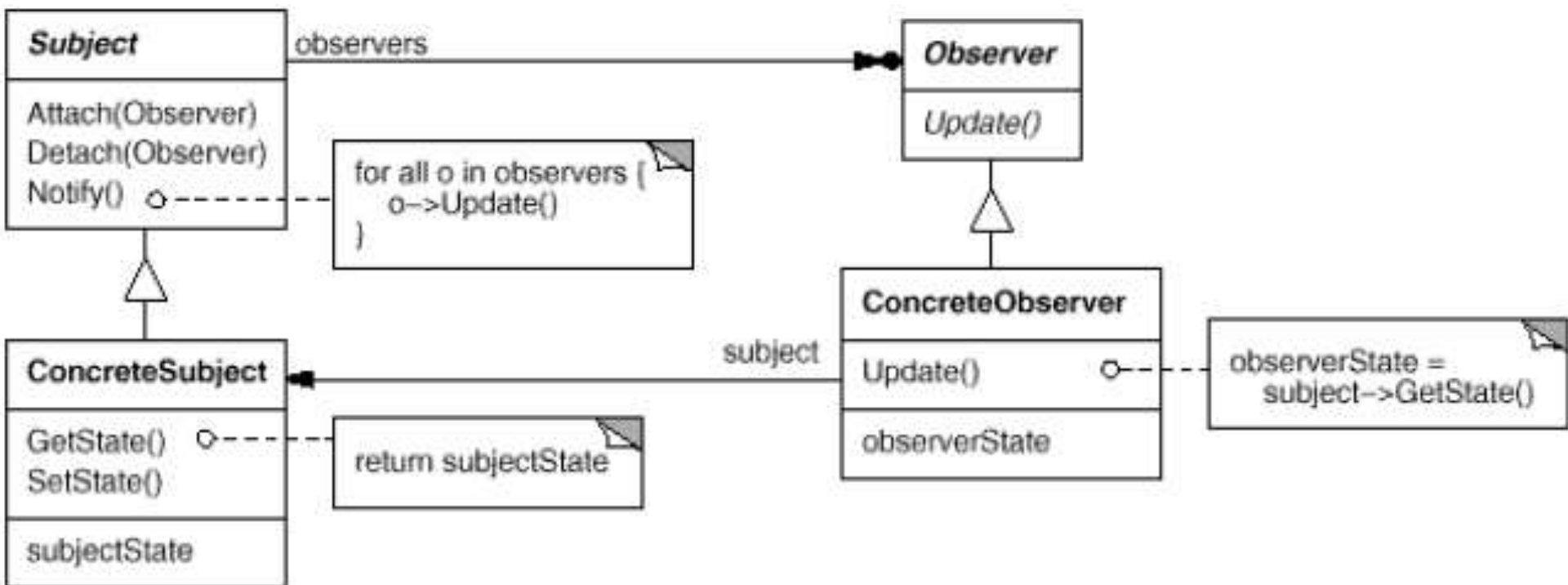
subject

—————> change notification
-----> requests, modification

Observer – Applicability

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

Observer – Structure



Observer – Example

- ▶ Below is an example that takes keyboard input and treats each input line as an event. The example is built upon the library classes **java.util.Observer** and **java.util.Observable**
- ▶ When a string is supplied from System.in, the method `notifyObservers` is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods – in our example, `ResponseHandler.update(...)`.
- ▶ *The Java Swing library makes extensive use of the observer pattern for event management*

Observer – Java 1

```
public class EventSource extends Observable
    implements Runnable {
    public void run() {
        try {
            final InputStreamReader isr = new
InputStreamReader( System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true ) {
                final String response = br.readLine();
                setChanged();
                notifyObservers( response ); }
        }
        catch (IOException e) { e.printStackTrace(); } }
}
```

Observer – Java 2

```
public class ResponseHandler implements Observer {  
    private String resp;  
  
    public void update (Observable obj, Object arg) {  
        if (arg instanceof String) {  
            resp = (String) arg;  
            System.out.println("\nReceived Response: "+ resp );  
        }  
    }  
}
```

Observer – Java 3

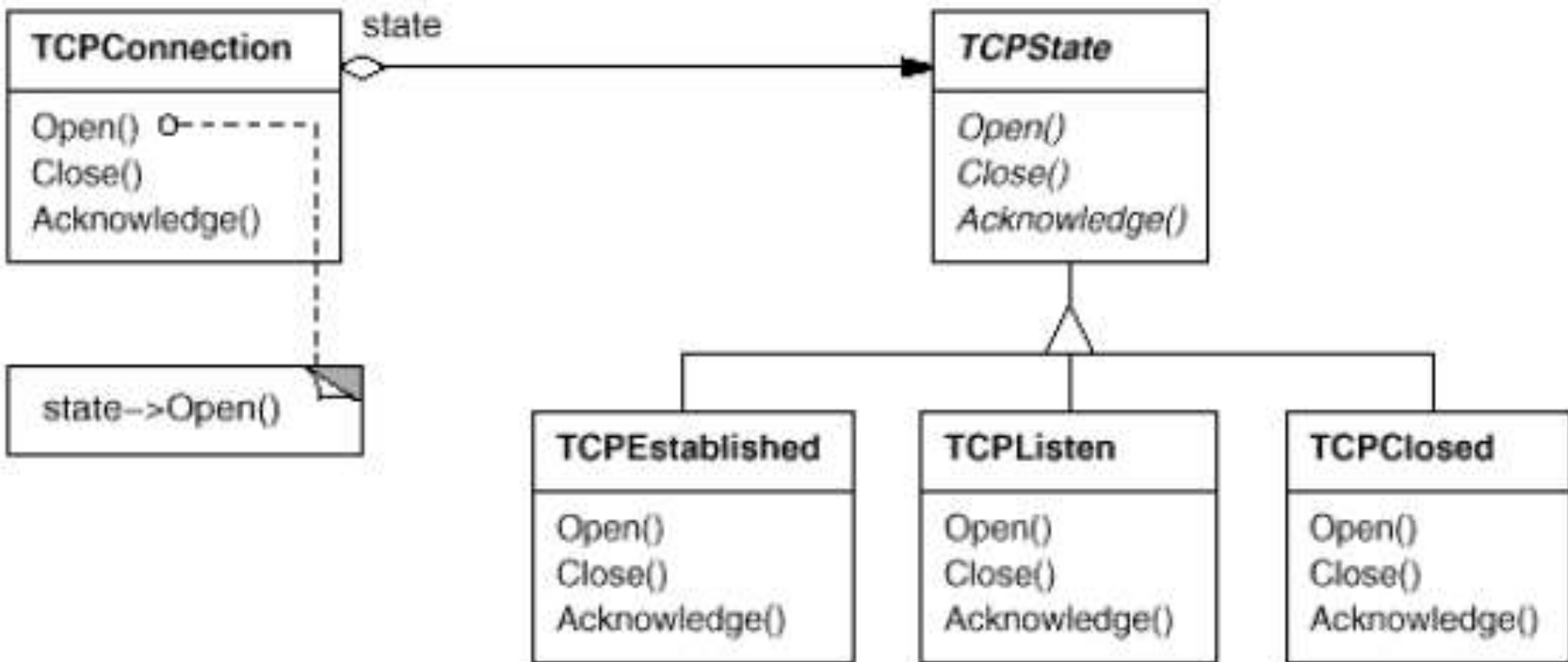
```
public class MyApp {  
    public static void main(String args[]) {  
        System.out.println("Enter Text >");  
        // create an event source – reads from stdin  
        final EventSource evSrc = new EventSource();  
        // create an observer  
        final ResponseHandler respHandler = new  
            ResponseHandler();  
        // subscribe the observer to the event source  
        evSrc.addObserver( respHandler );  
        // starts the event thread  
        Thread thread = new Thread(evSrc);  
        thread.start();  
    }  
}
```

State

- ▶ **Intent** – Allow an object to alter its behavior when its internal state changes
- ▶ **Also Known As** – Objects for States
- ▶ **Motivation** – Consider a class `TCPConnection` that represents a network connection. A `TCPConnection` object can be in one of several different states: Established, Listening, Closed. When a `TCPConnection` object receives requests from other objects, it responds differently depending on its current state

State – Idea

- ▶ The key idea in this pattern is to introduce an abstract class called `TCPState` to represent the states of the network connection.



State – Applicability

- ▶ Use the State pattern in either of the following cases:
 - An object's behavior depends on its state
 - Operations have large, multipart conditional statements that depend on the object's state

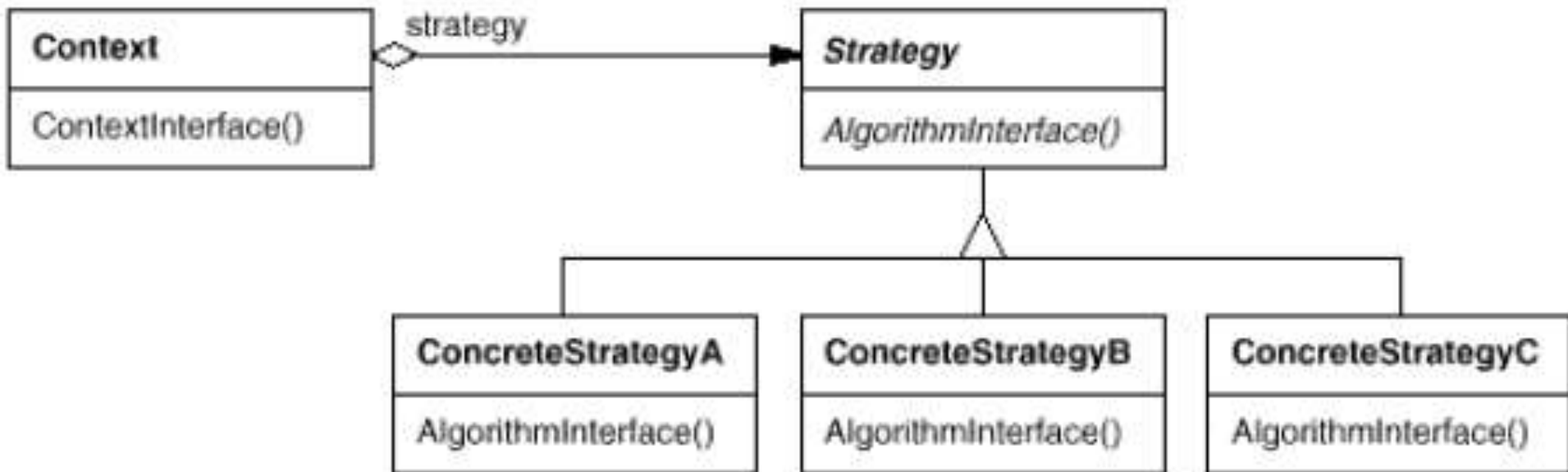
Strategy



- ▶ **Intent** – Define a family of algorithms, encapsulate each one, and make the minter changeable
- ▶ **Also Known As** – Policy
- ▶ **Motivation** – Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons

Strategy – Structure

- ▶ With Strategy pattern, we can define classes that encapsulate different line breaking algorithms



Strategy – Example

- ▶ In the strategy pattern algorithms can be selected at runtime.
- ▶ A standard calculator that implements basic operations: $+$, $-$, $*$



Strategy – Java 1

```
interface Strategy {  
    int execute(int a, int b);  
}  
class ConcreteStrategyAdd implements Strategy {  
    public int execute(int a, int b) {  
        System.out.println("Called ConcreteStrategyA's execute()");  
        return (a + b);  
    }  
}  
class ConcreteStrategySub implements Strategy {  
    public int execute(int a, int b) {  
        System.out.println("Called ConcreteStrategyB's execute()");  
        return (a - b);  
    }  
}  
class ConcreteStrategyMul implements Strategy {  
    public int execute(int a, int b) {  
        System.out.println("Called ConcreteStrategyC's execute()");  
        return a * b;  
    }  
}
```

Strategy – Java 2

```
class Context {  
    Strategy strategy;  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }  
    public int execute(int a, int b) {  
        return this.strategy.execute(a, b);  
    }  
}
```

Strategy – Java 3

```
class StrategyExample {  
  
    public static void main(String[] args) {  
        Context context;  
        context = new Context(new ConcreteStrategyAdd());  
        int resultA = context.execute(3,4);  
        context = new Context(new ConcreteStrategySub());  
        int resultB = context.execute(3,4);  
        context = new Context(new ConcreteStrategyMul());  
        int resultC = context.execute(3,4);  
    }  
}
```


Template Method



- ▶ **Intent** – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- ▶ **Motivation** – Consider an application framework that provides Application and Document classes.
- ▶ The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file

Template Method – Example

- ▶ The template pattern is often referred to as the **Hollywood Principle**: "*Don't call us, we'll call you.*" Using this principle, the template method in a parent class controls the overall process by calling subclass methods as required
- ▶ This is shown in several games in which players play against the others, but only one is playing at a given time

Template Method – Java 1

```
abstract class Game {  
    protected int playersCount;  
    abstract void initializeGame();  
    abstract void makePlay(int player);  
    abstract boolean endOfGame();  
    abstract void printWinner();  
  
    final void playOneGame(int playersCount) {  
        this.playersCount = playersCount;  
        initializeGame(); int j = 0;  
        while (!endOfGame()) {  
            makePlay(j); j = (j + 1) % playersCount; }  
        printWinner();  
    }  
}
```

Template Method – Java 2

```
class Monopoly extends Game {  
    // Implementation of necessary concrete methods  
  
    void initializeGame() { // ... }  
    void makePlay(int player) { // ... }  
    boolean endOfGame() { // ... }  
    void printWinner() { // ... }  
  
    // Specific declarations for the Monopoly game.  
}  
  
class Chess extends Game {  
    ...}
```

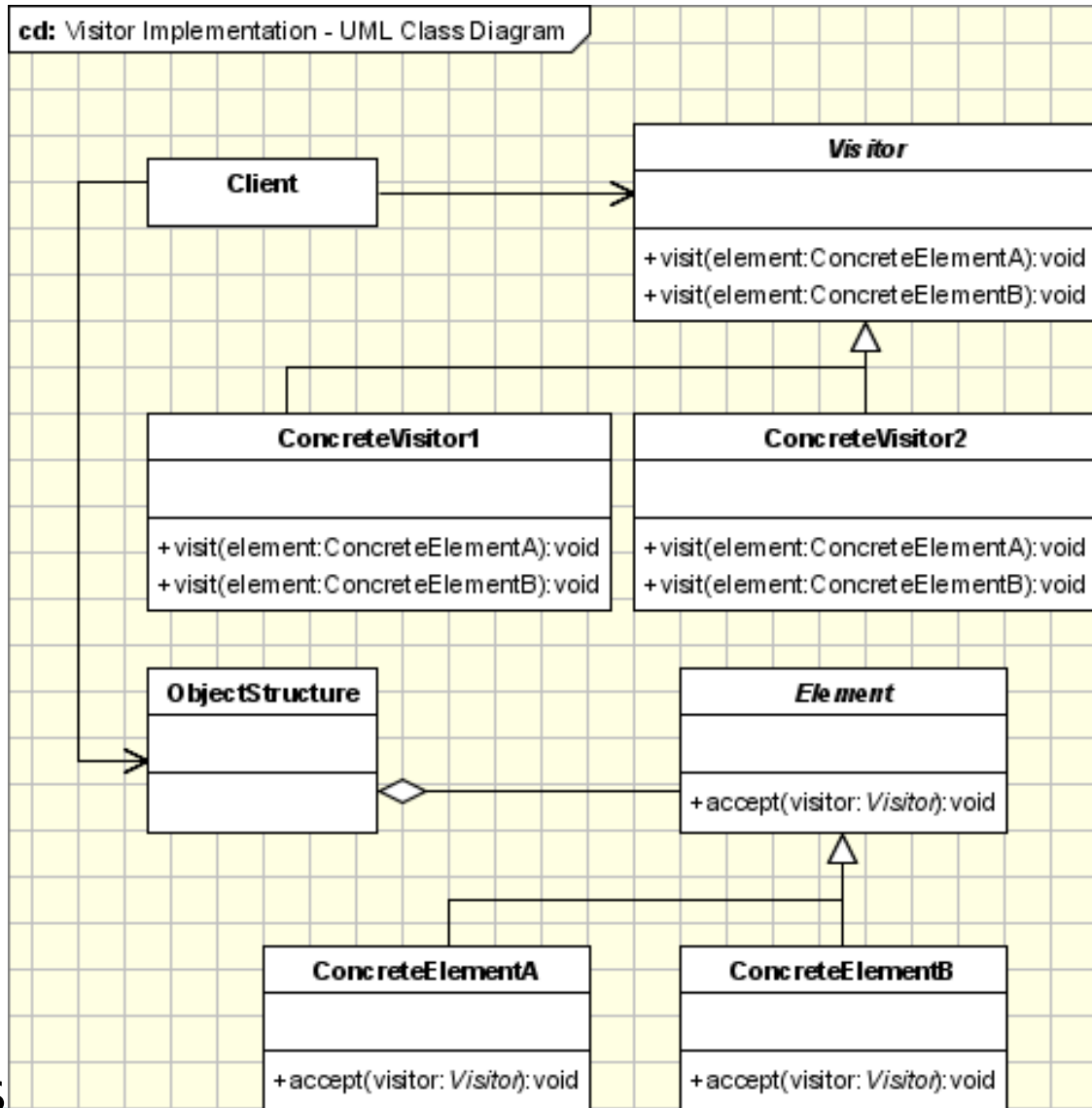
Visitor



- ▶ **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.
- ▶ **Motivation** – Collections are data types widely used in object oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type

Visitor – Structure Example

- ▶ We want to create a reporting module in our application to make statistics about a group of customers
- ▶ The statistics should be made very detailed so all the data related to the customer must be parsed
- ▶ All the entities involved in this hierarchy must accept a visitor so the CustomerGroup, Customer, Order and Item are visitable objects



Visitor– Applicability

- ▶ The visitor pattern is used when:
 - Similar operations have to be performed on objects of different types grouped in a structure
 - There are many distinct and unrelated operations needed to be performed
 - The object structure is not likely to be changed but is very probable to have new operations which have to be added

Visitor Pattern using Reflection

- ▶ Reflection can be used to overcome the main drawback of the visitor pattern
- ▶ When the standard implementation of visitor pattern is used the method to invoke is determined at runtime
- ▶ **Reflection is the mechanism used to determine the method to be called at compile-time**

Bibliografie

- ▶ Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)

Links

- ▶ Structural Patterns: <http://www.oodesign.com/structural-patterns/>
- ▶ Gang-Of-Four: <http://c2.com/cgi/wiki?GangOfFour>,
<http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf>
- ▶ Design Patterns Book:
<http://c2.com/cgi/wiki?DesignPatternsBook>
- ▶ About Design Patterns:
<http://www.javacamp.org/designPattern/>
- ▶ Design Patterns – Java companion:
<http://www.patterndepot.com/put/8/JavaPatterns.htm>
- ▶ Java Design patterns:
http://www.allapplabs.com/java_design_patterns/java_design_patterns.htm
- ▶ Overview of Design Patterns:
http://www.mindspring.com/~mgrand/pattern_synopses.htm