Ingineria programării

Curs 9 – 2-3 Mai

Recapitulare

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

Recapitulare – CP

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

Recapitulare – SP

- **Adapter** – socket-plug
- **Bridge** – drawing API
- **Composite** – menus
- **Decorator** – Christmas tree

Structural Patterns

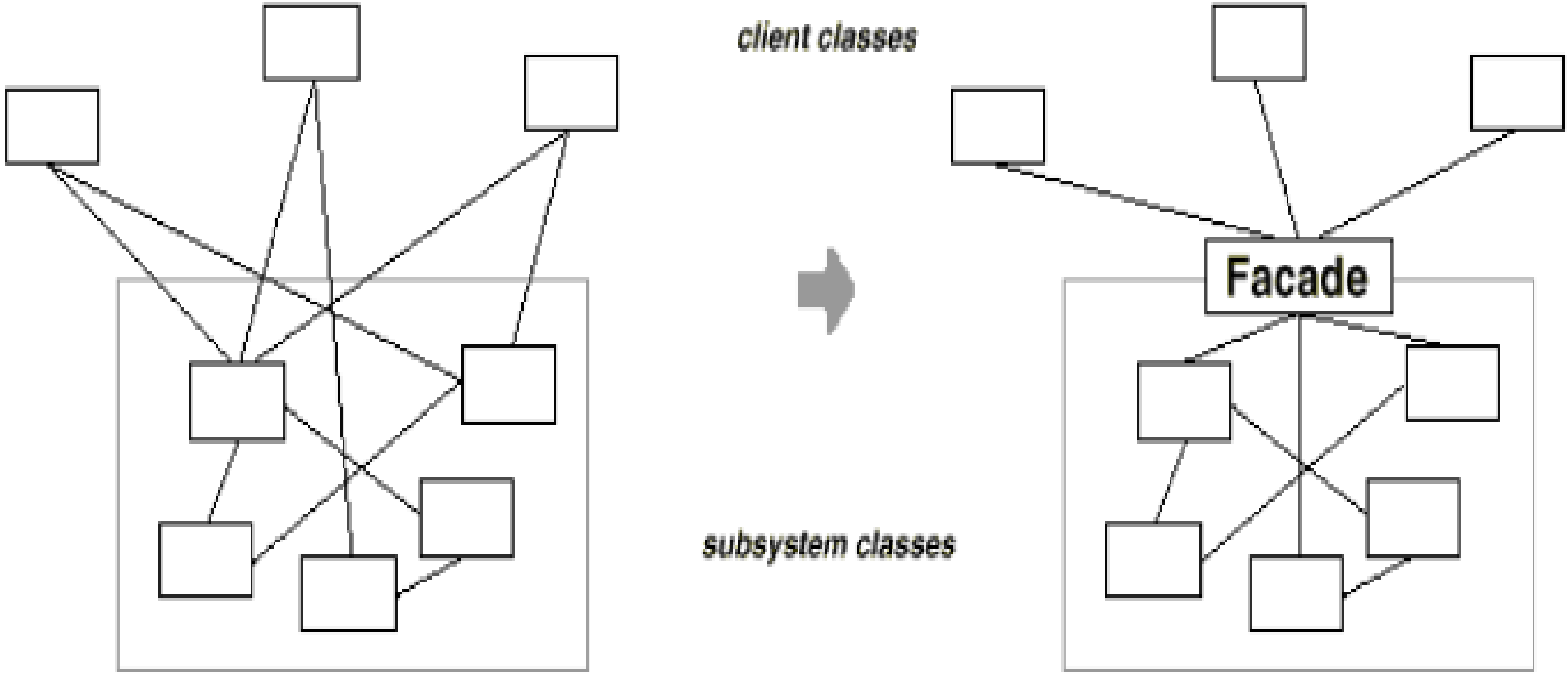
- **Façade** – store keeper
- **Flyweight** – FontData
- **Proxy** – ATM access

Structural Patterns – Façade

- **Intent – Provide a unified interface to a set of interfaces in a subsystem**
- **Motivation** – Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as *Scanner, Parser, ProgramNode, BytecodeStream,* and *ProgramNodeBuilder* that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler want to compile some code

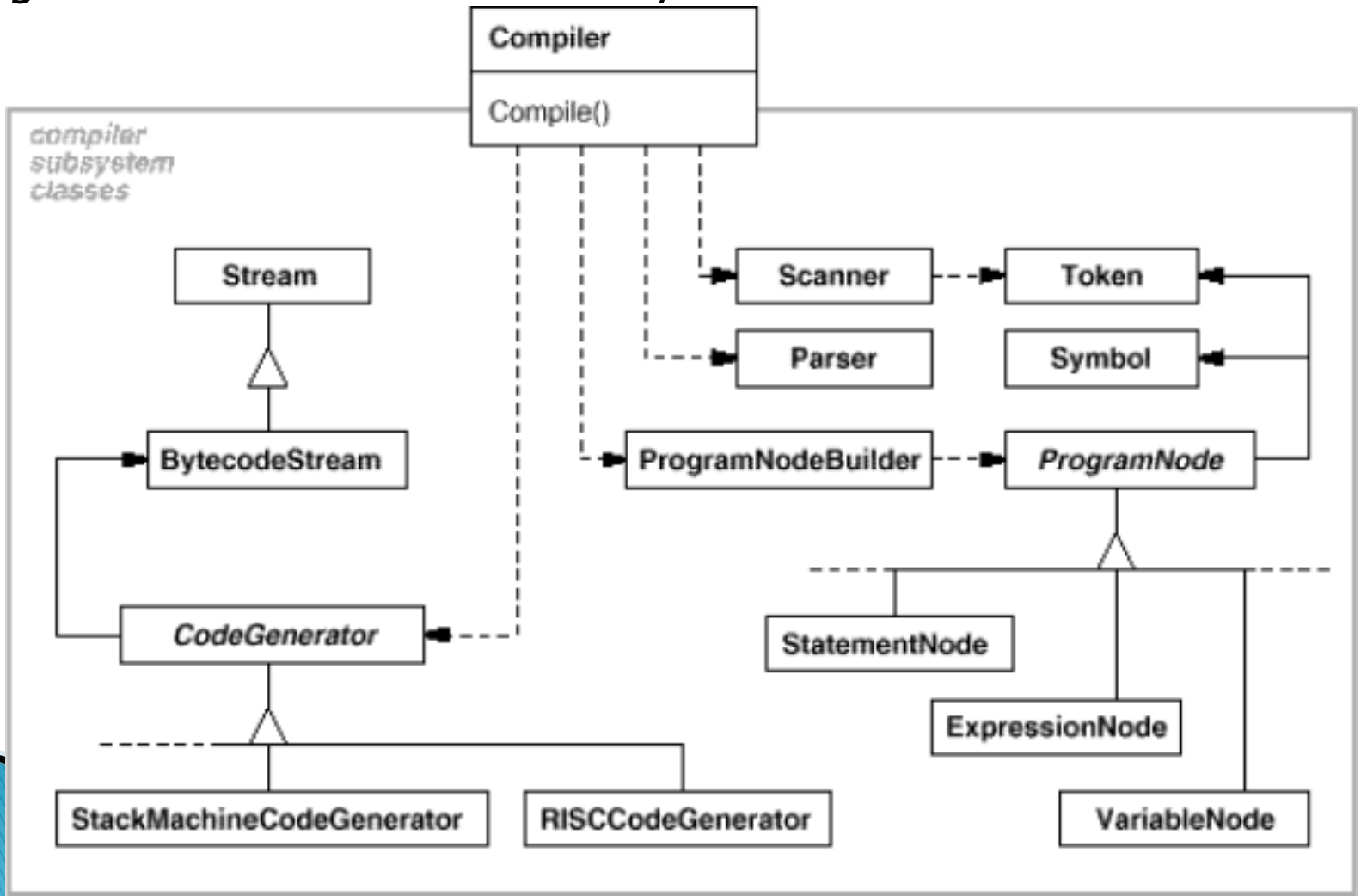
Façade 1

- A common design goal **is to minimize the communication and dependencies between** subsystems



Façade 2

- The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it



Façade 3

- **Applicability** – Use the Facade pattern when
  - you want to provide a simple interface to a complex subsystem
  - there are many dependencies between clients and the implementation classes of an abstraction
  - you want to layer your subsystems

Façade – Example 1

- Facade as the name suggests means the face of the building. The people walking past the road can only see this glass face of the building. The face hides all the complexities of the building and displays a friendly face.
- Facade hides the complexities of the system and provides an interface to the client from where the client can access the system. In Java, the interface JDBC can be called a façade
- Other examples?

Façade – Example 2

- Let's consider a **store**. This store has a store keeper. In the storage, there are a lot of things stored e.g. **packing material, raw material and finished goods**.
- You, as client want access to different goods. You do not know where the different materials are stored. You just have access to store keeper who knows his store well. Here, the store keeper acts as the facade, as he hides the complexities of the system Store.

Façade – Java 1

```
public interface Store {
  public Goods getGoods();
}

public class FinishedGoodsStore implements Store
{
  public Goods getGoods() {
      FinishedGoods finishedGoods = new FinishedGoods();
      return finishedGoods;
  }
}
```

Façade – Java 2

```
public class StoreKeeper {
  public RawMaterialGoods getRawMaterialGoods() {
      RawMaterialStore store = new RawMaterialStore();
      RawMaterialGoods rawMaterialGoods =
  (RawMaterialGoods)store.getGoods();
      return rawMaterialGoods;
  }
  …
}
```

Façade – Java 3

```
public class Client {
  public static void main(String[] args) {
      StoreKeeper keeper = new StoreKeeper();
      RawMaterialGoods rawMaterialGoods =
      keeper.getRawMaterialGoods();
  }
}
```

Façade – The Good, The Bad …

- Isolates and mask system complexity from the user
- The façade class runs the risk of being coupled to everything

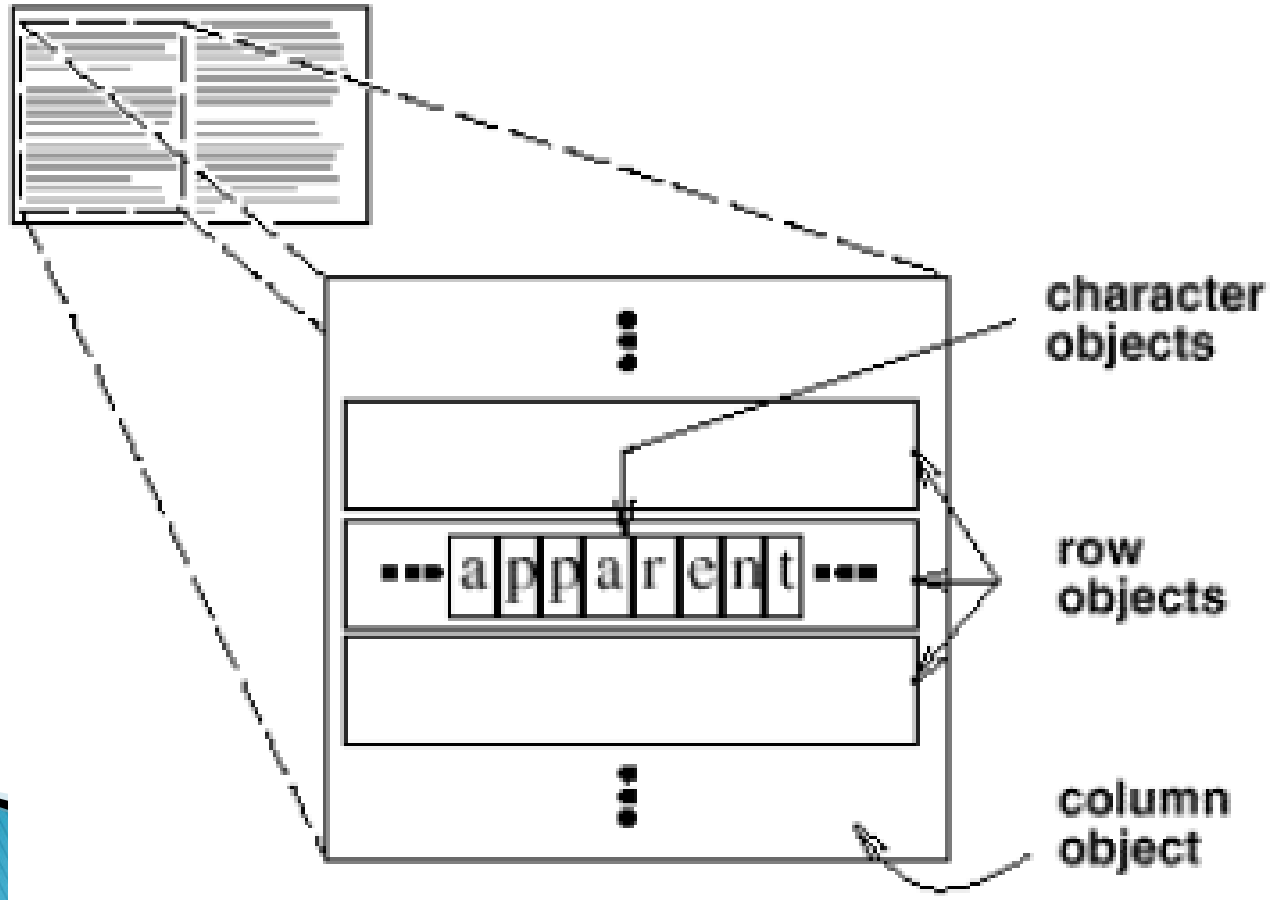
Structural Patterns – Flyweight

- **Intent – Use sharing to support large numbers of fine-grained objects efficiently**

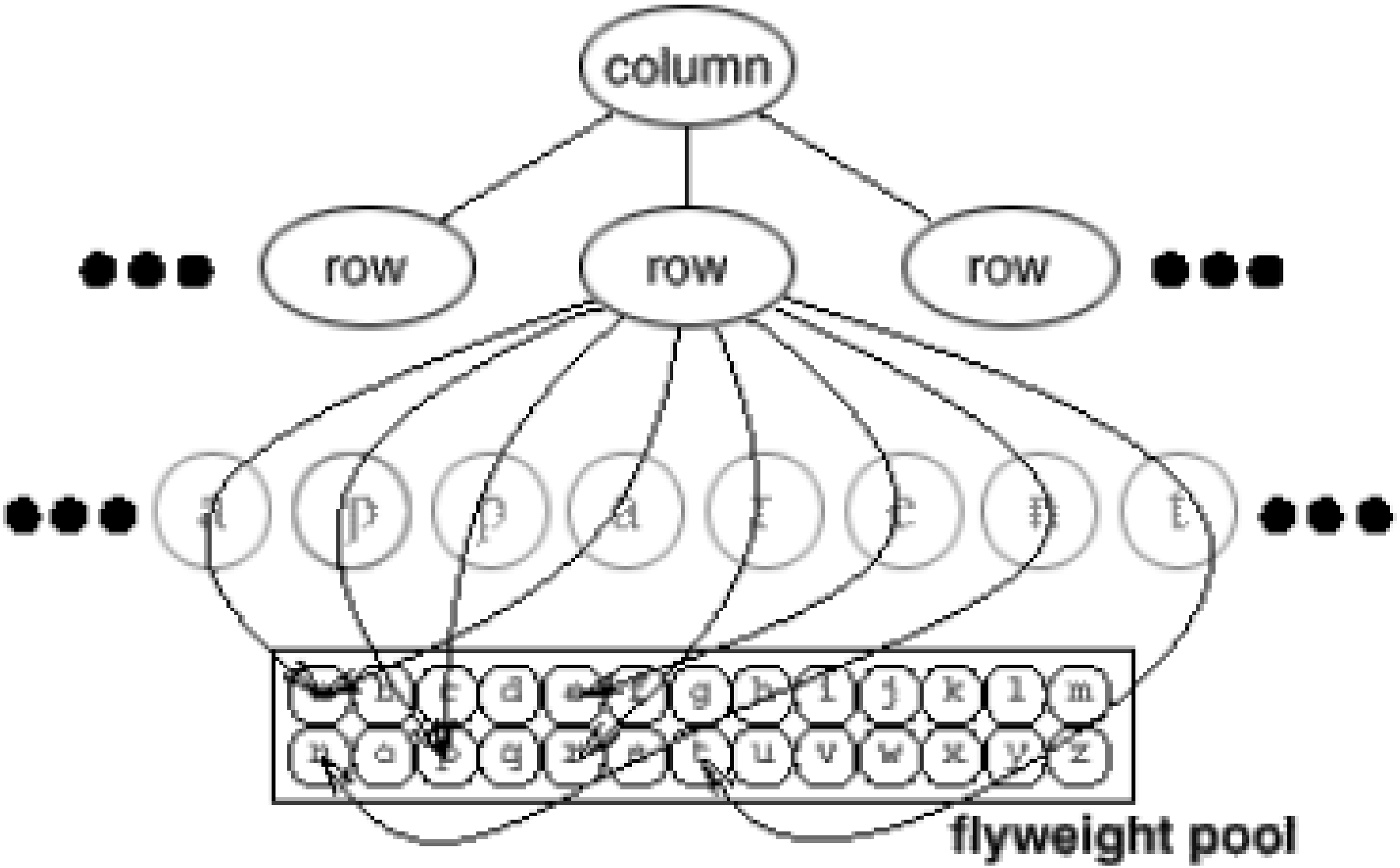
- **Motivation** – Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.
- For example, most document editor implementations have text formatting and editing facilities that are modularized to some extent.

Flyweight 1

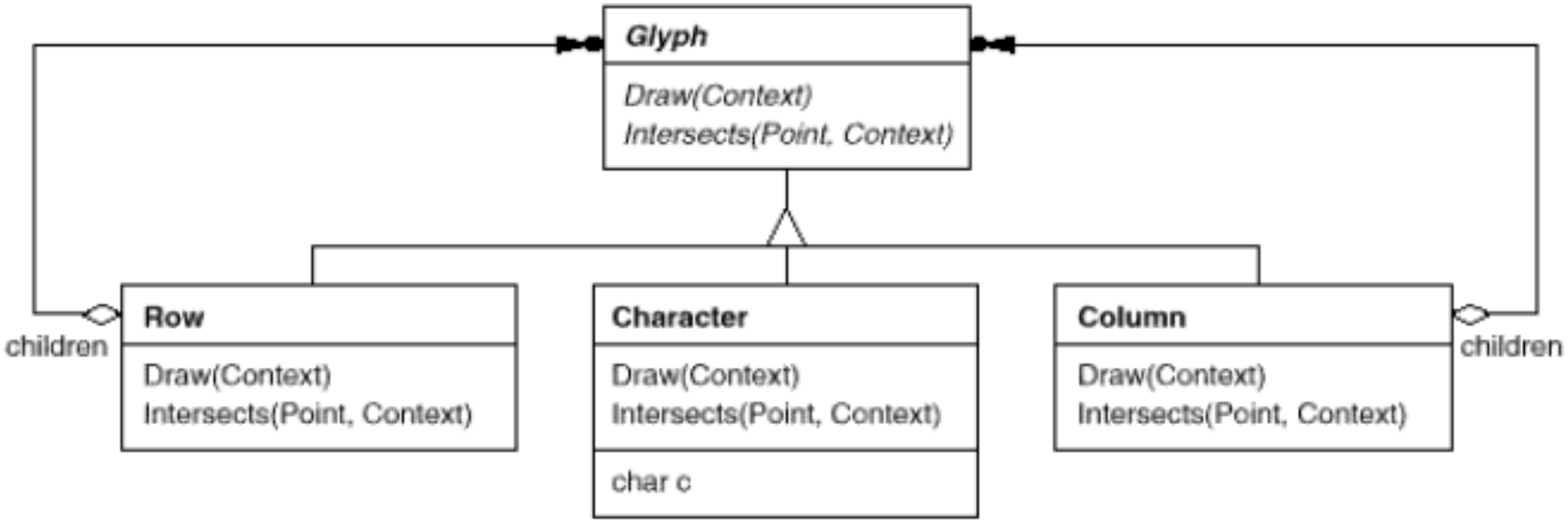
- The following diagram shows how a document editor can use objects to represent characters



Flyweight 2



Flyweight 3


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
children

Flyweight 4

- **Applicability** – Use the Flyweight pattern when
  - Supporting a large number of objects that:
    - Are similar
    - Share at least some attributes
    - Are too numerous to easily store whole in memory

Flyweight – Example

- **A Flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects**

- A classic example usage of the flyweight pattern are the data structures for graphical representation of characters in a word processor. It would be nice to have, for each character in a document, a glyph object containing its *font outline, font metrics*, and other formatting data, but it would amount to hundreds or thousands of bytes for each character. Instead, are used the flyweights called **FontData**

Flyweight – Java 1

```
public enum FontEffect {
   BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}
public final class FontData {
   private static final WeakHashMap<FontData,
   WeakReference<FontData>> FLY_WEIGHT_DATA = new
   WeakHashMap<FontData, WeakReference<FontData>>();
   private final int pointSize;
   private final String fontFace;
   private final Color color;
   private final Set<FontEffect> effects;

   private FontData(int pointSize, String fontFace, Color color,
   EnumSet<FontEffect> effects) {
         this.pointSize = pointSize;
         this.fontFace = fontFace;
         this.color = color;
         this.effects = Collections.unmodifiableSet(effects);
   }
```

Flyweight – Java 2

```
public static FontData create(int pointSize, String
  fontFace, Color color, FontEffect... effects) {
  EnumSet<FontEffect> effectsSet =
  EnumSet.noneOf(FontEffect.class);
  for (FontEffect fontEffect : effects) {
  effectsSet.add(fontEffect); }
  FontData data = new FontData(pointSize, fontFace,
  color, effectsSet);
  if (!FLY_WEIGHT_DATA.containsKey(data)) {
       FLY_WEIGHT_DATA.put(data, new
       WeakReference<FontData> (data));
  }
  return FLY_WEIGHT_DATA.get(data).get();
}
```

Flyweight – The Good, The Bad …

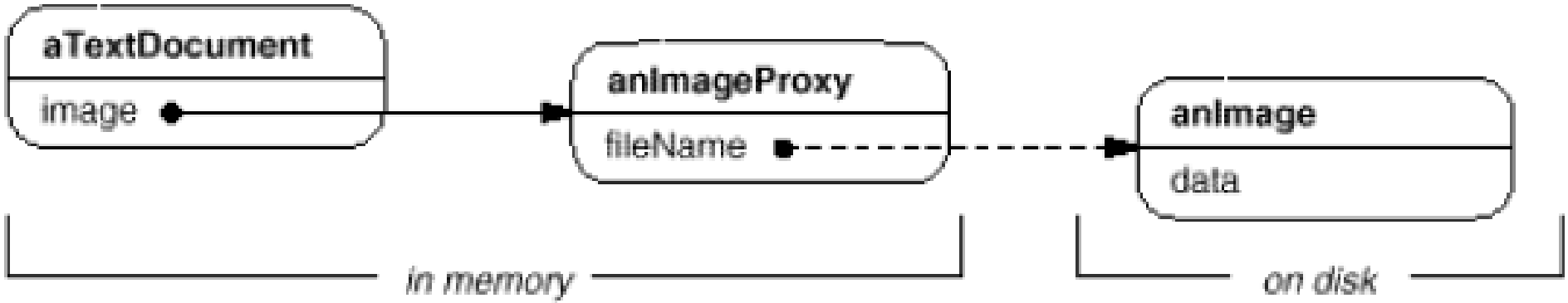
- Saves on memory in the case of large numbers of objects
- Becomes costly in processing time
- The code is complicated and not intuitive

Structural Patterns – Proxy

- **Intent – Provide a surrogate or placeholder for another object to control access to it.**
- **Also Known As** – Surrogate
- **Motivation** – Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time

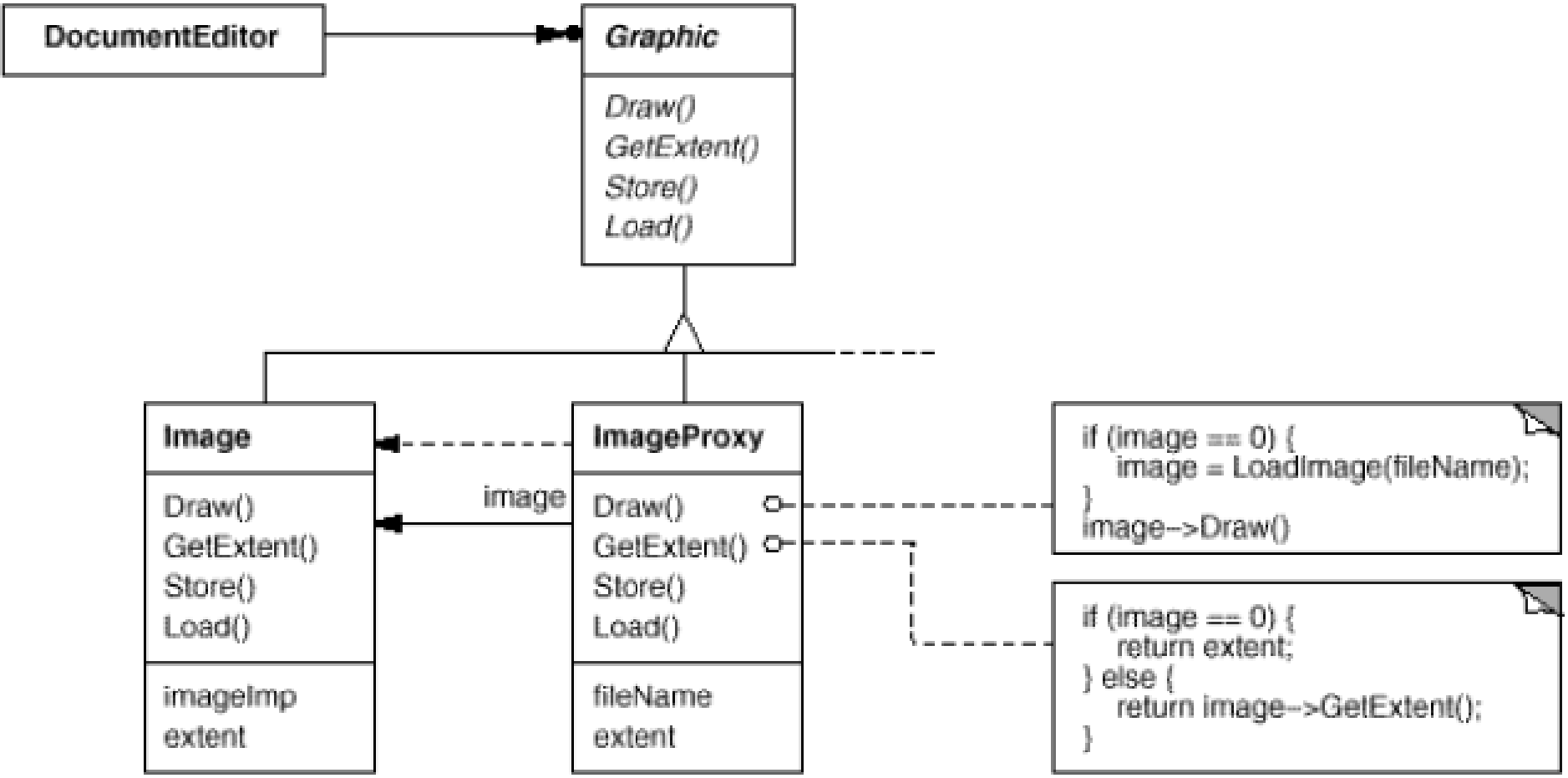
Proxy 1

- The solution is to use another object, an image **proxy, that acts as a stand-in** for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



Proxy 2

- The following class diagram illustrates this example in more detail



Proxy 3

- **Applicability** – Use the Proxy pattern when
  - You need to provide some interposed service between the application logic and the client
  - Provide some lightweight version of a service or resource
  - Screen or restrict user access to a resource or service

Proxy – Example

- Let’ say we need to withdraw money to make some purchase. The way we will do it is, go to an ATM and get the money, or purchase straight with a cheque.
- In old days when ATMs and cheques were not available, what used to be the way??? Well, get your passbook, go to bank, get withdrawal form there, stand in a queue and withdraw money. Then go to the shop where you want to make the purchase.
- In this way, we can say that ATM or cheque in modern times act as proxies to the Bank.

Proxy – Java 1

```
public class Bank {
   private int numberInQueue;

   public double getMoneyForPurchase(double amountNeeded) {
         You you = new You("Prashant");
         Account account = new Account();
         String accountNumber = you.getAccountNumber();
         boolean gotPassbook = you.getPassbook();
         int number = getNumberInQueue();

         while (number != 0) {number--; }

         boolean isBalanceSufficient =
   account.checkBalance(accountNumber, amountNeeded);
         if(isBalanceSufficient)
                  return amountNeeded;
         else
                  return 0;
   }

   private int getNumberInQueue() {
         return numberInQueue;}
}
```

Proxy – Java 2

```
public class ATMProxy {
  public double getMoneyForPurchase(double amountNeeded){
    You you = new You("Prashant");
    Account account = new Account();
    boolean isBalanceAvailable = false;
    if(you.getCard()) {
        isBalanceAvailable =
        account.checkBalance(you.getAccountNumber(),
        amountNeeded);
   }

   if(isBalanceAvailable)
        return amountNeeded;
   else
        return 0;
   }
}
```

Proxy – The Good, The Bad …

- The provided service can be changed without affecting the client
- The proxy is available even if the base service or resource may be unavailable
- Preserves O (from SOLID) – you can add new proxies without changing the service or client
- It usually delays the response to the client
- The code is complicated because of increased number of classes

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

Patterns

- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

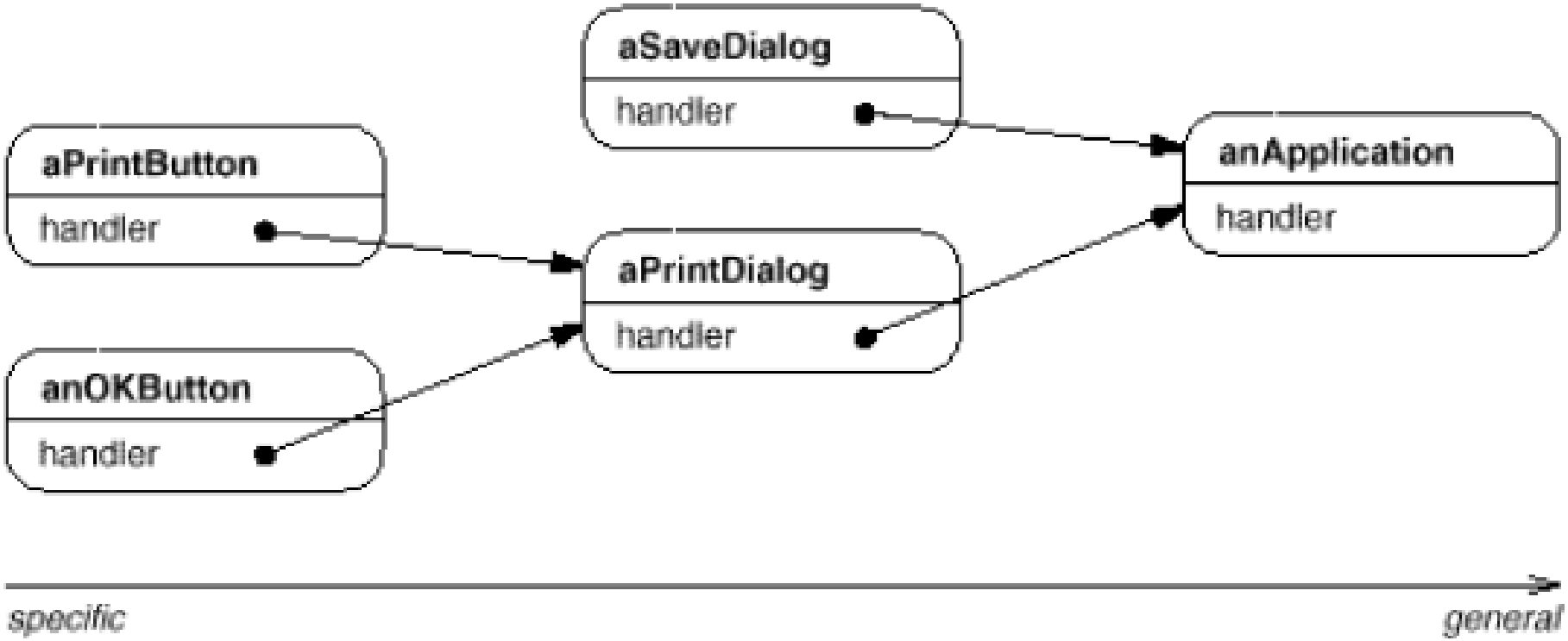
Chain of Responsibility

- **Intent** – Chain the receiving objects and **pass the request along the chain until an object handles it**

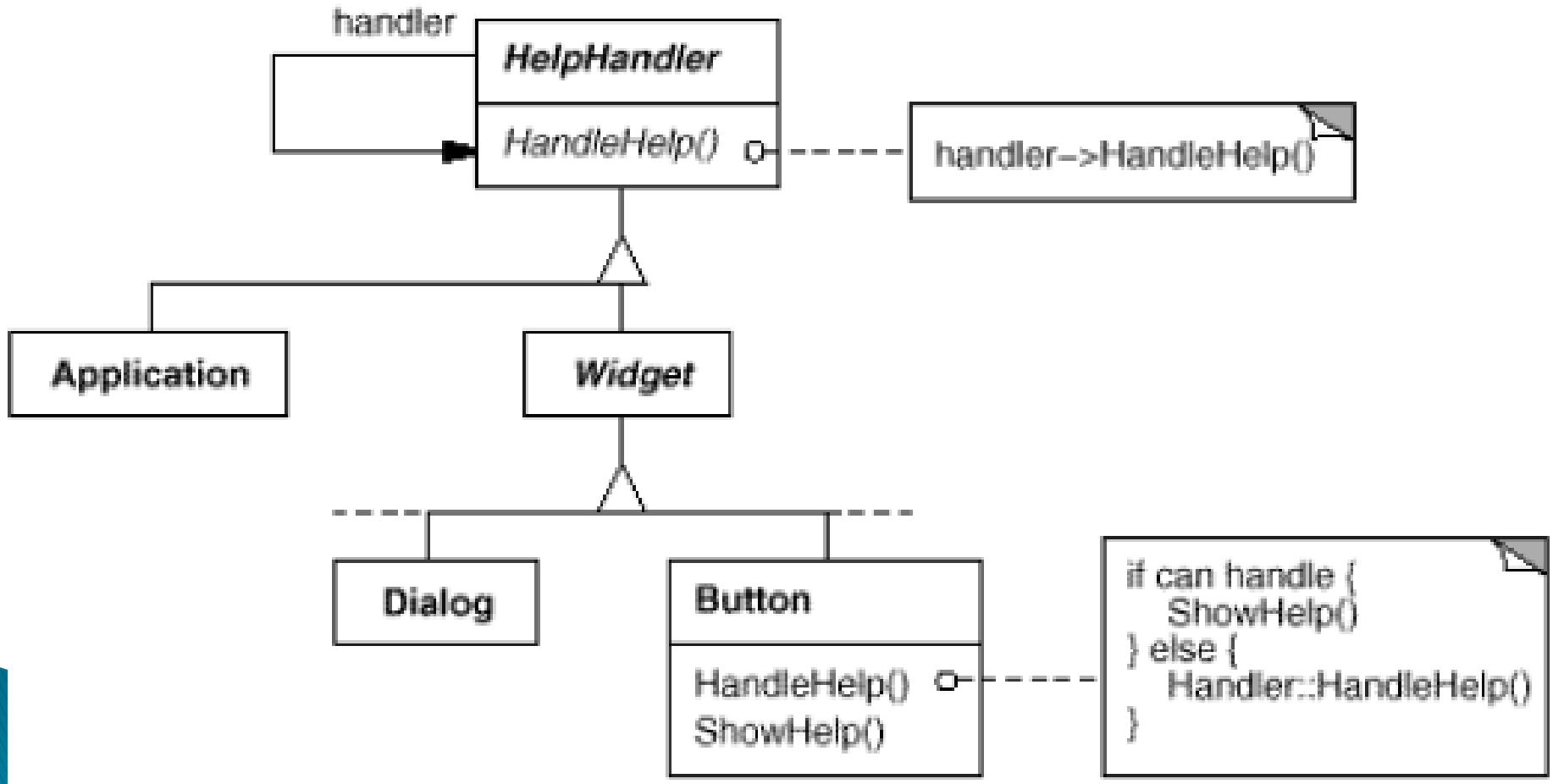
- **Motivation** – Consider a context-sensitive help facility for a graphical user interface. The help that's provided depends on the part of the interface that's selected and its context. If no specific help information exists for that part of the interface, then the help system should display a more 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
         }
   }}}
```

Chain of Responsability– The Good, The Bad …

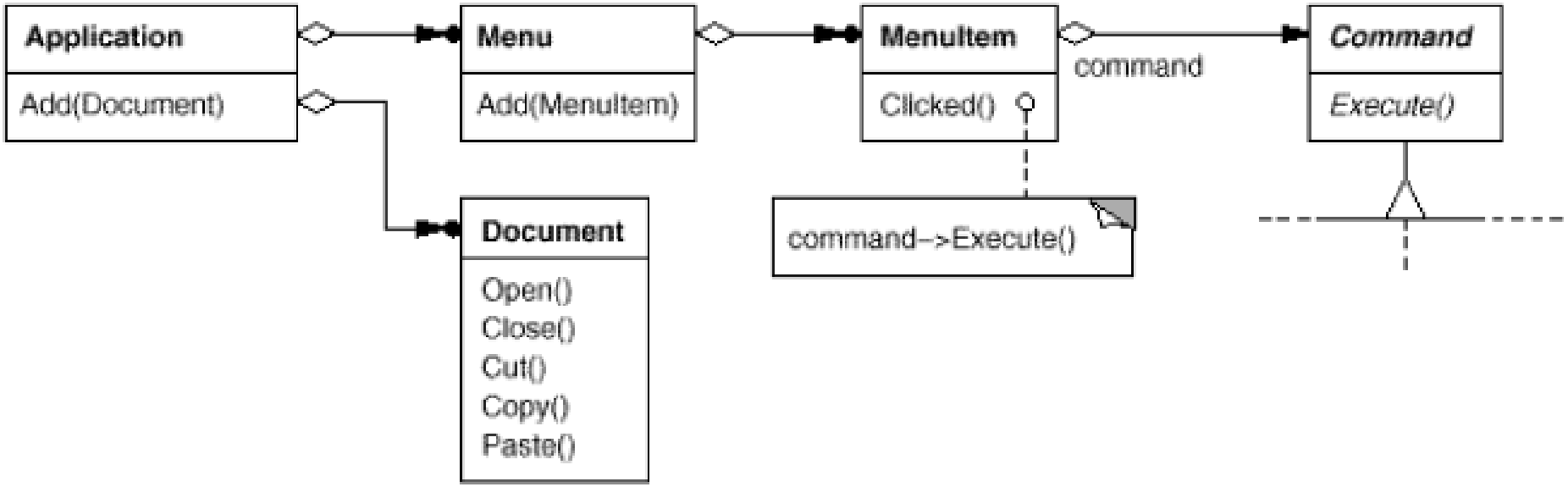
- Control the sequence of handler calls
- Preserves S and O (from SOLID)
- Some requests may not be handled by any class

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

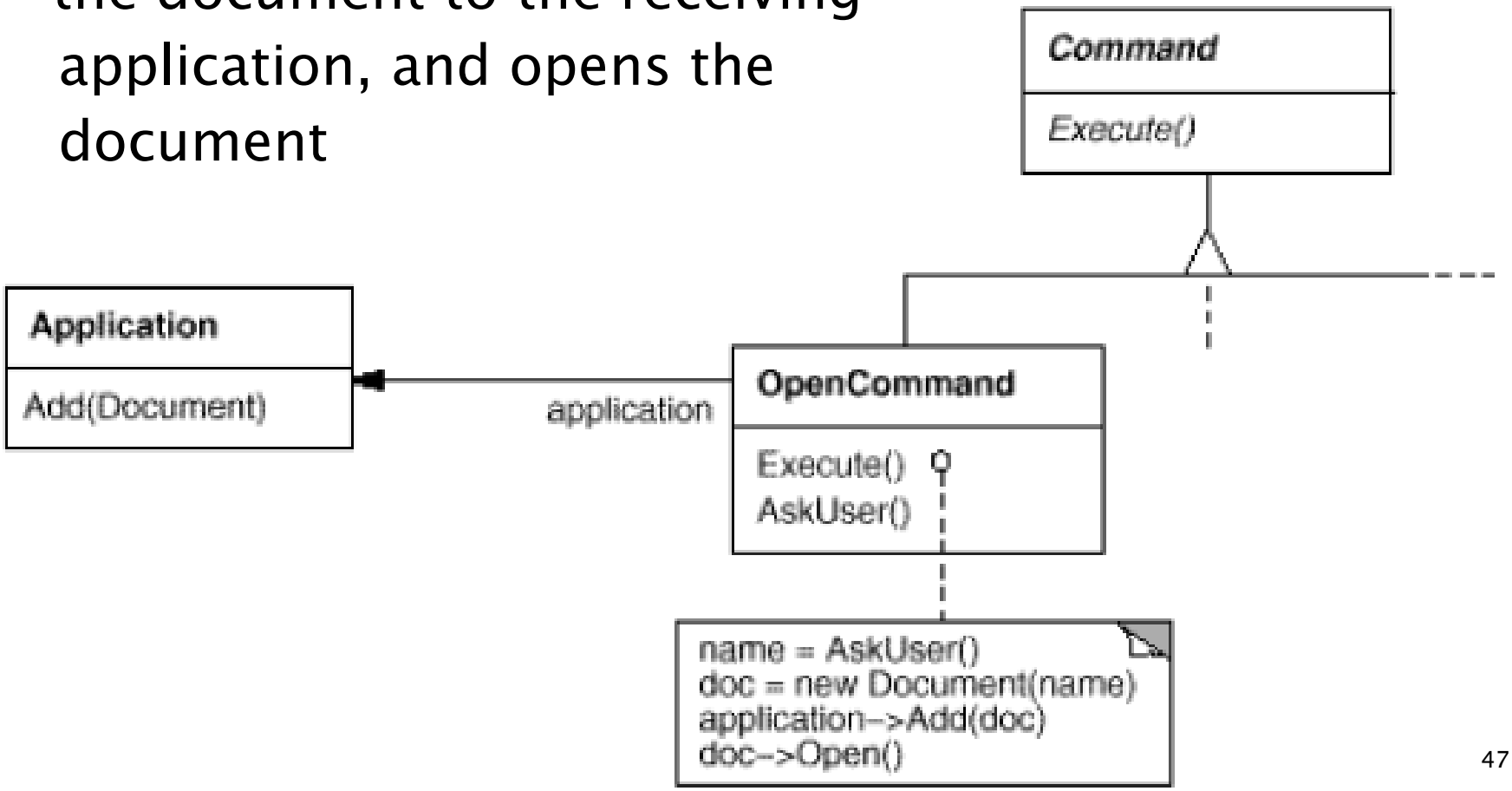
Command 2

- 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;
  }
}
```

Command– The Good, The Bad …

- Supports undo/redo types of operations
- Preserves S and O (from SOLID)
- Combine simple commands into a single complex one
- Allows delaying execution
- Code becomes complicated because of an extra layer of code between caller and service

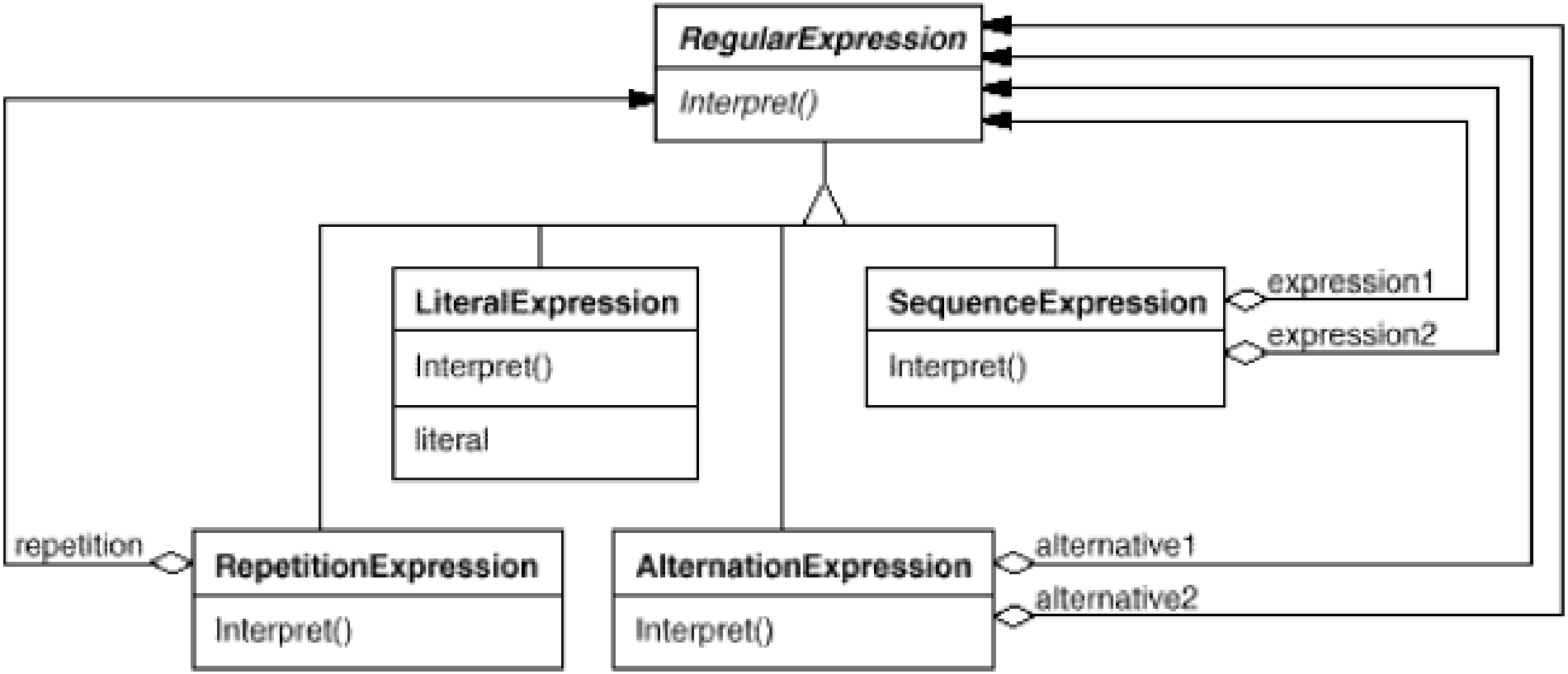
Interpreter

- **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:
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

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, Mi… ” 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) {
  }
}
```

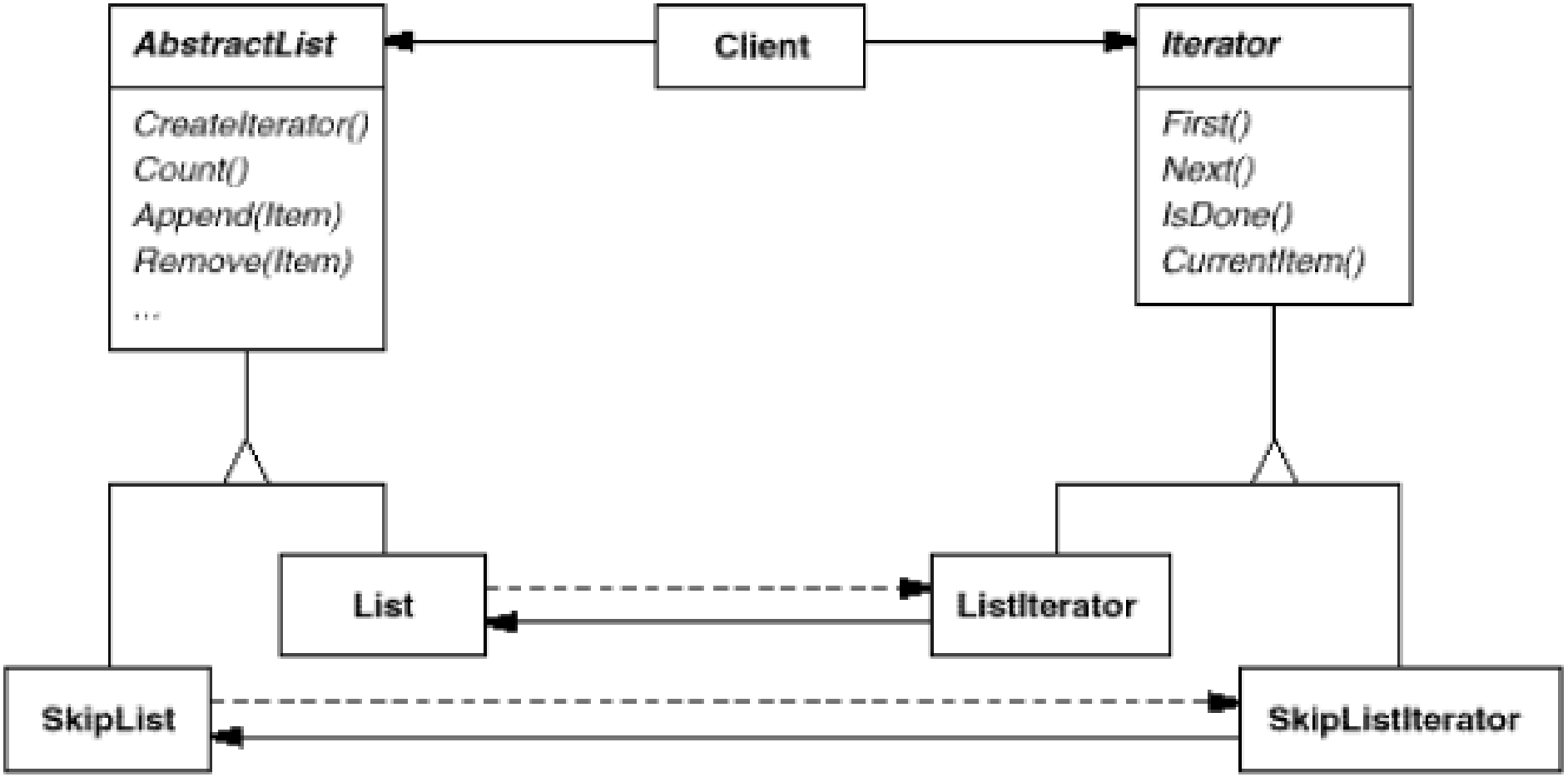
Interpreter – The Good, The Bad …

- Detaches user classes from model classes
- Preserves S and O (from SOLID)
- Code becomes complicated because of lots of extra classes

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());
  }
}
```

Iterator – The Good, The Bad …

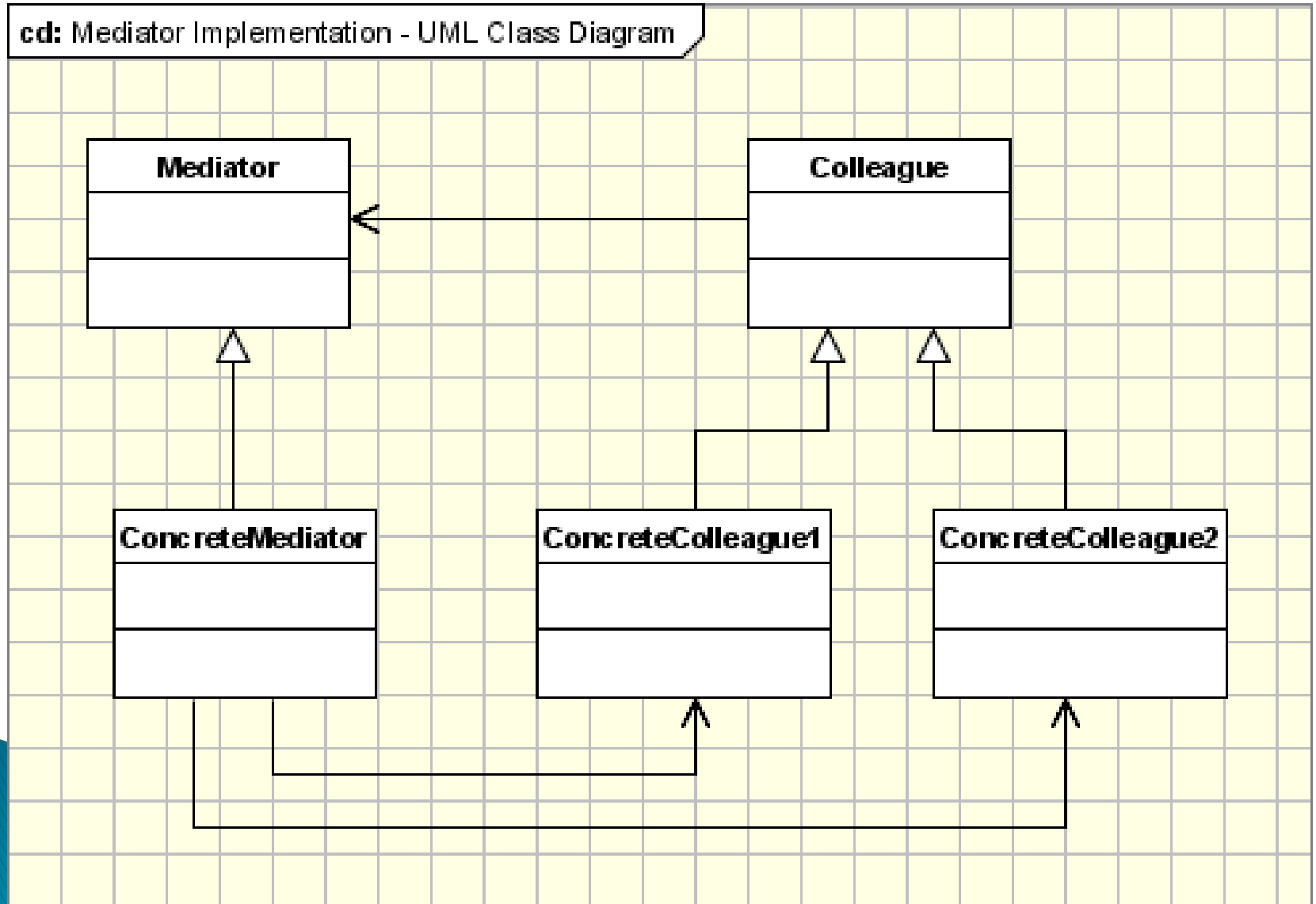
- Allows the use of multiple iterators at the same time on the same collections
- Iteration can be stopped and resumed at will, as each iterator conserves its state
- Preserves S and O (from SOLID)
- May lead to unnecessary complexity for simple collections
- Slows iteration over particular types of collections

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



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?

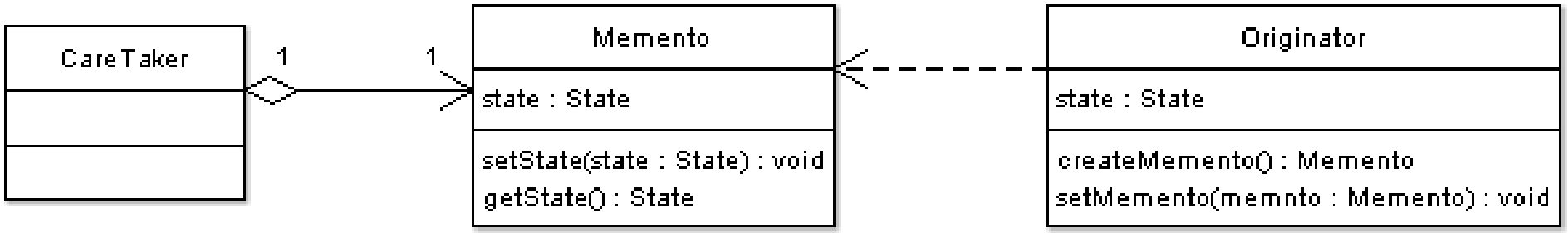
Mediator – The Good, The Bad …

- Reduces coupling
- Allows for easy reuse of classes
- Preserves S and O (from SOLID)
- The Mediator may become a God Object (knows too much, does too many things)

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**

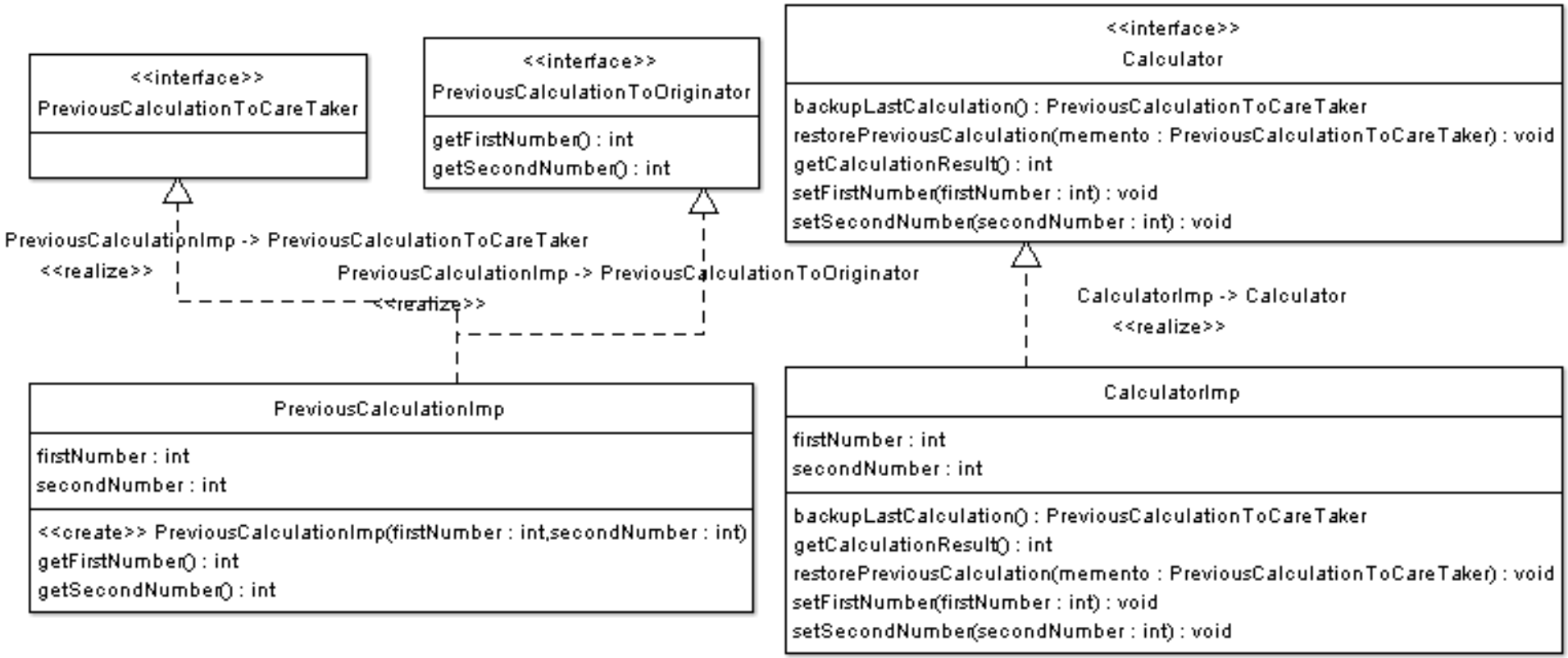
Memento – Structure



- **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

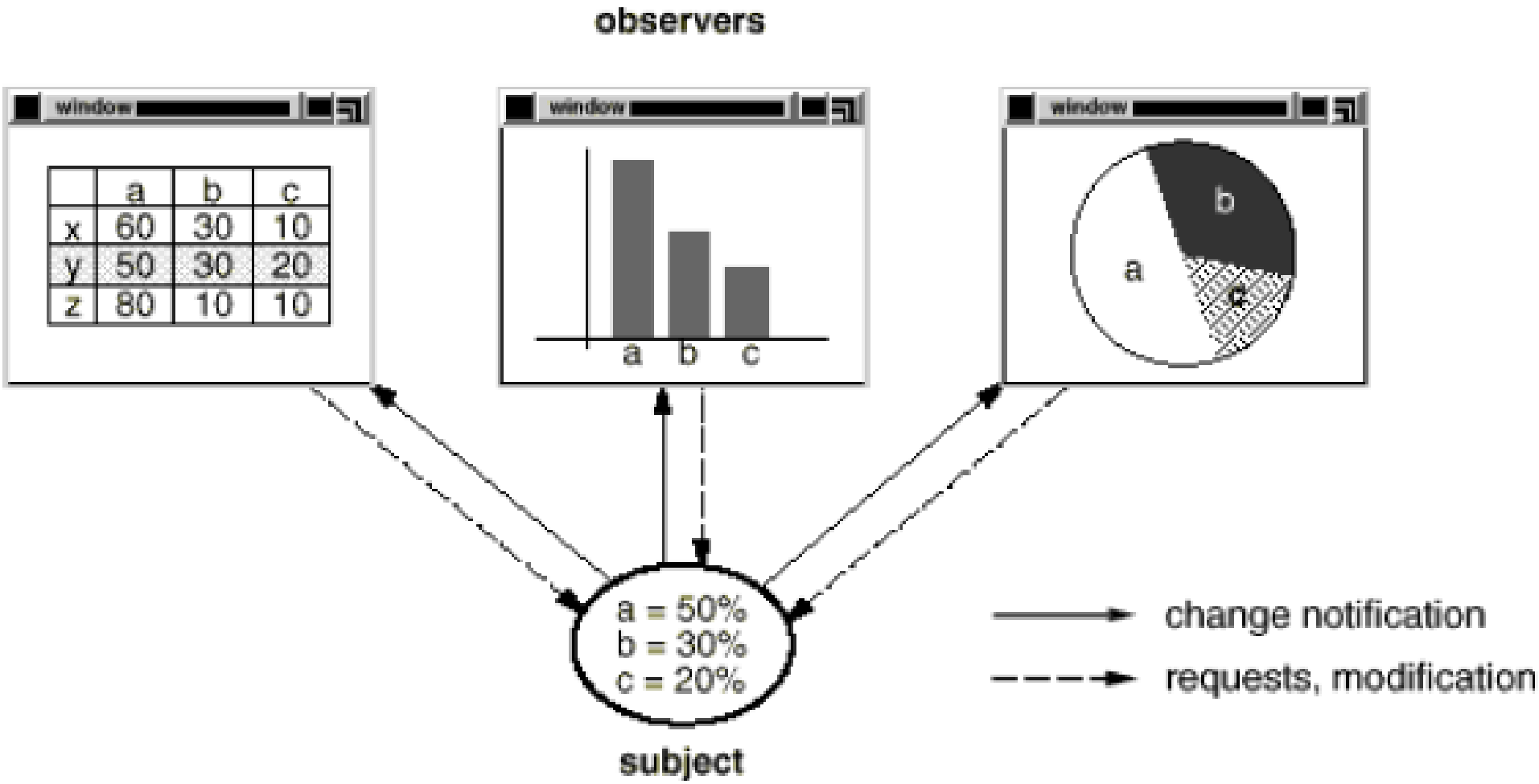
Memento – The Good, The Bad …

- Can produce saves of object states without breaking encapsulation
- Decreases responsibilities of the originator by managing mementos in the caretaker
- Numerous mementos use a lot of memory
- Overhead for the caretakers as they need to manage which mementos are obsolete and destroy them
- Some dynamic programming languages (JavaScript, Python, etc.) cannot guarantee an unchangeable state for the memento

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

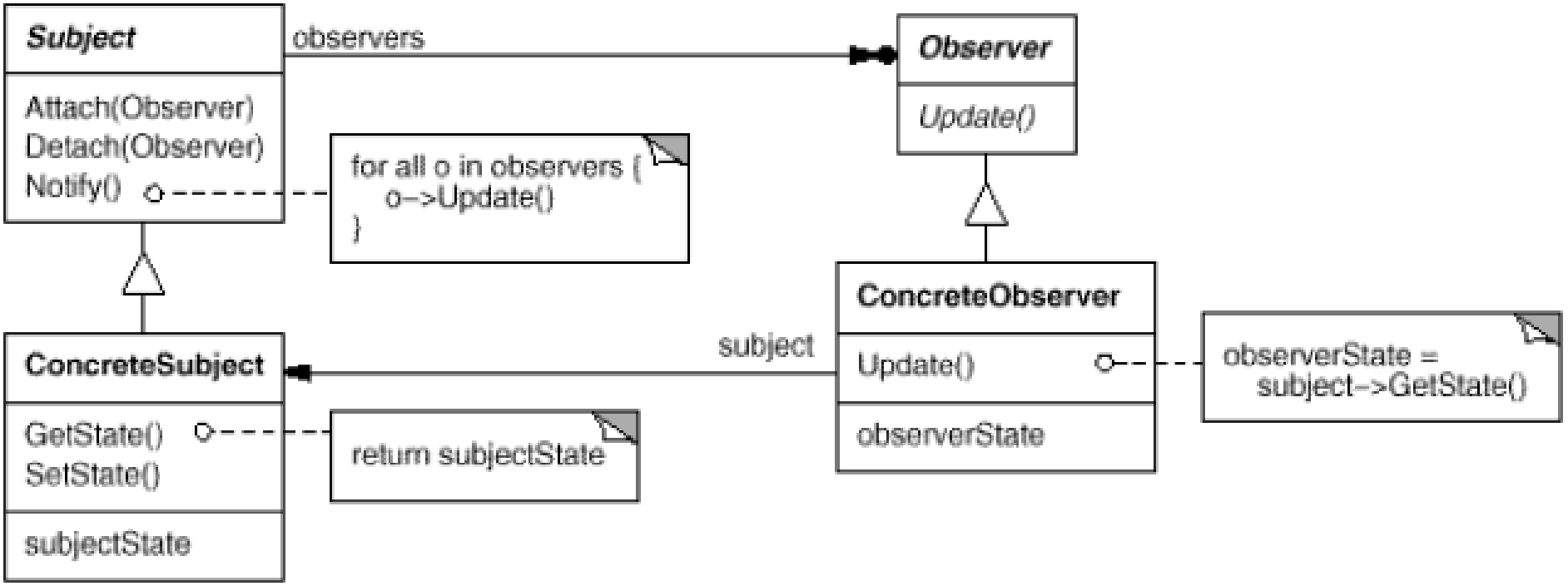
Observer – Example



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();
    }
}
```

Observer – The Good, The Bad …

- Can establish relations between objects (not classes) at runtime
- Preserves S and O (from SOLID)
- The order of notification of observers is random

Bibliography

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