7. Behavioral Patterns – 2 of 3

***SW Design Patterns*,**
by Boyan Bontchev,
FMI - Sofia University
2006/2016

Annotation

- Behavioral patterns
- Definitions
- Properties
- Intent, motivation, structure, participants, collaborations, consequences, implementation issues about:
  - Iterator
  - Mediator
  - Observer
  - State
- Examples in Java

References

1. Gamma, Helm, Johnson, Vlissides **("Gang of Four“ - GoF)** *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995
2. *Design Patterns Explained*, by Allan Shalloway and James Trott, Prentice Hall, 2001
3. THE DESIGN PATTERNS JAVA COMPANION, by JAMES W. COOPER, Adison-Wesley, October 2, 1998

Design pattern catalog - GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Template Method • Memento • Observer • State • Strategy • Visitor

Behavioral Design Pattern

- Behavioral patterns are *concerned with algorithms and the assignment of responsibilities* between objects.

- Behavioral patterns describe not just patterns of objects or classes but also the *patterns of communication* between them. 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.

Let follow a paper…

- **Non-Software Examples of Software Design Patterns,** by Michael Duell, in AG Communication Systems e-zine:

  http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm

The 11 Behavioral Design Pattern 1/3

Behavioral class patterns _use inheritance to distribute behavior between classes_:

- **Template Method (GoF325) - concerns an abstract definition of an algorithm; defines the algorithm step by step, where each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations.**
- **Interpreter (GoF243) - represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.**
- **Memento (GoF381) - without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later**

11 Behavioral Design Pattern 2/3

Behavioral object patterns _use object composition_ rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is _how peer objects know about each other_. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other.

- **Mediator (GoF273) - avoids this by introducing a mediator object between peers; provides the indirection needed for loose coupling.**
- **Chain of Responsibility (GoF223) - provides even looser coupling. It lets you send requests to an object implicitly through a chain of candidate objects. Any candidate may fulfill the request depending on runtime conditions. The number of candidates is open-ended, and you can select which candidates participate in the chain at run-time.**
- **Observer (GoF293) - pattern defines and maintains a dependency between objects. The classic example of *Observer is in Smalltalk Model/View/Controller*, where all views of the model are notified whenever the model's state changes.**

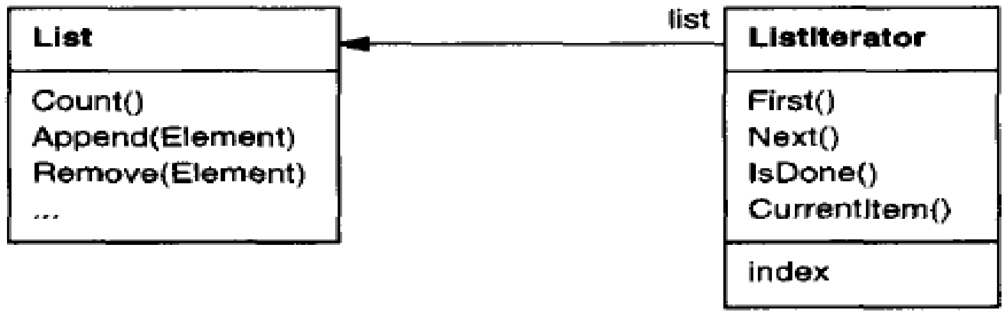
11 Behavioral Design Pattern 3/3

Other behavioral object patterns are concerned with encapsulating behavior in an object and delegating requests to it:

- Strategy (GoF315) - encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses;
- Command (GoF233) - encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways.
- State (GoF305) - encapsulates the states of an object so that the object can change its behavior when its state object changes;
- Visitor (GoF331) encapsulates behavior that would otherwise be distributed across classes;
- Iterator (GoF257) - abstracts the way you access and traverse objects in an aggregate.

The Iterator Pattern

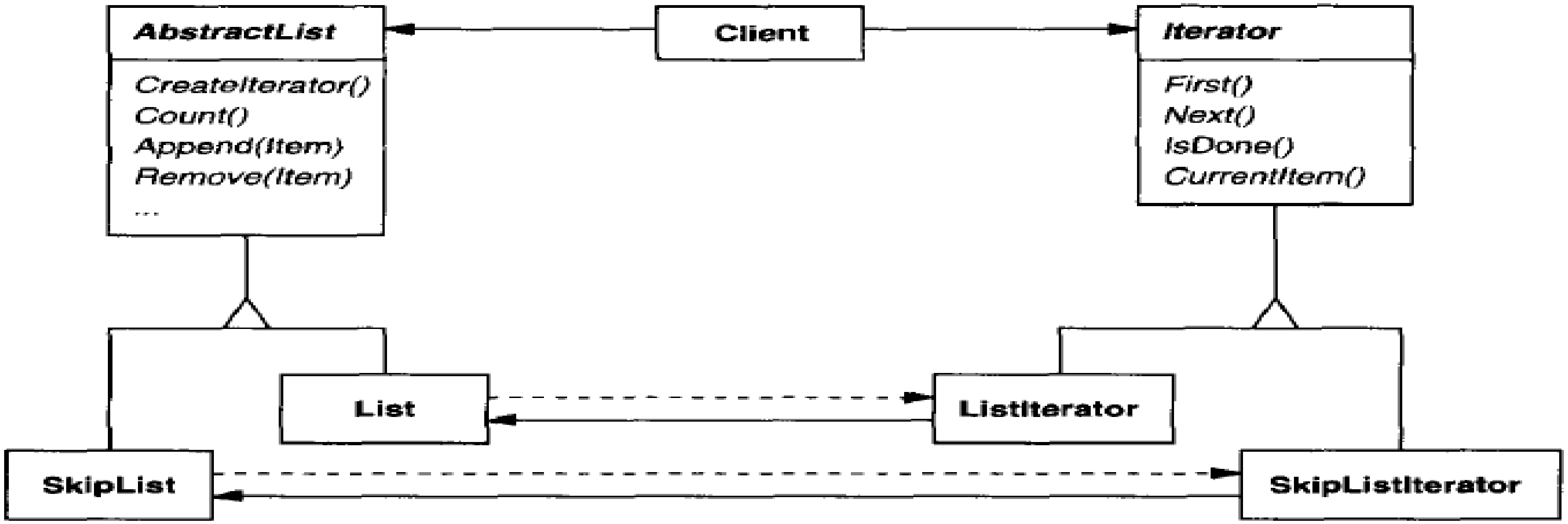
- **Intent** - provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Also Known As** - Cursor
- **Motivation** - an aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
  - Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish.
  - But you probably don't want to extend the List interface with operations for different traversals, even if you could anticipate the ones you will need.
  - You might also need to have more than one traversal pending on the same list.



The Solution

- The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an **iterator** object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.
- Before you can instantiate ListIterator, you must supply the List to traverse. Once you have the ListIterator instance, you can access the list's elements sequentially:
  - The CurrentItem operation returns the current element in the list.
  - First initializes the current element to the first element.
  - Next advances the current element to the next element.
  - IsDone tests whether we've advanced beyond the last element
- *Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface*. For example, FilteringListIterator might provide access only to those elements that satisfy specific filtering constraints.

- The client should not know that it is a *list* that's traversed as opposed to some other aggregate structure. We have to be able to change the aggregate class without changing clients => **polymorphic iteration.**
- We define an ***AbstractList*** class that provides a common interface for manipulating lists. Similarly, we need an abstract ***Iterator*** class that defines a common iteration interface. Then we can define concrete **Iterator** subclasses for the different list implementations => the iteration mechanism independent of concrete aggregate classes.
- **CreateIterator** is an example of a factory method (GoF107). We use it here to let a client ask a list object for the appropriate iterator.

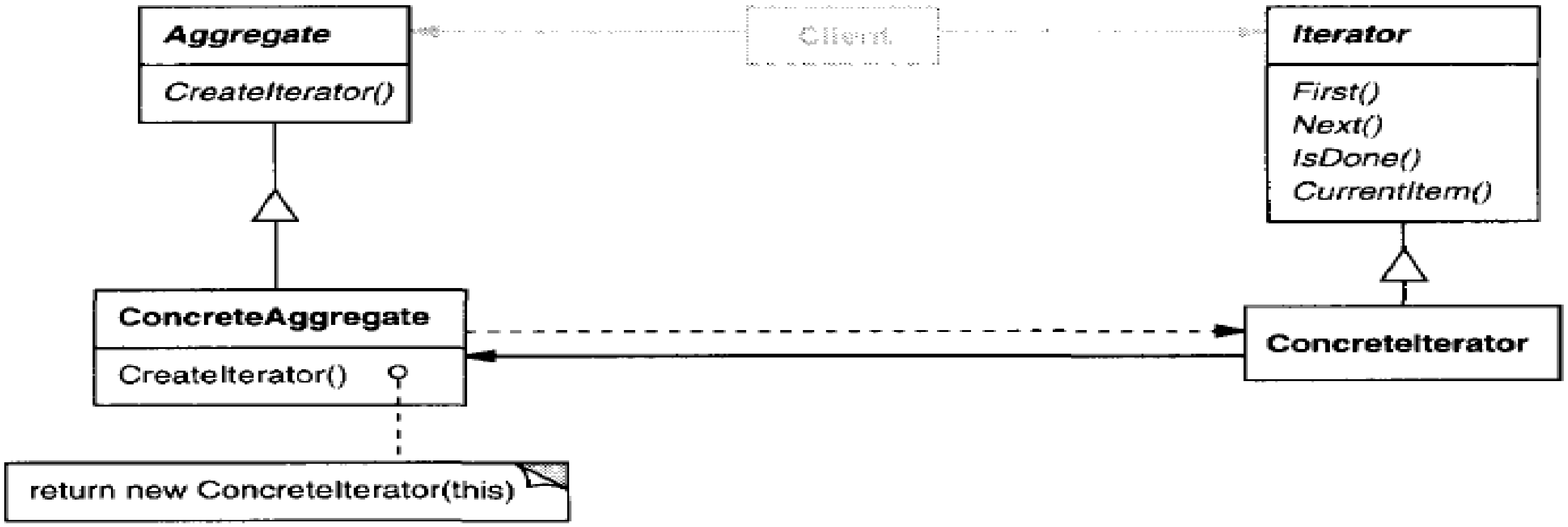


Applicability

Use the Iterator pattern:

- 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).

Structure and Participants



- **Iterator**
  - - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - - implements the Iterator interface.
  - - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations and Consequences

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.
- Three important consequences:
  - 1. *It supports variations in the traversal of an aggregate.* Complex aggregates may be traversed in many ways. Iterators make it easy to change the traversal algorithm: Just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.
  - 2. *Iterators simplify the Aggregate interface.* Iterator's traversal interface avoids the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
  - 3. *More than one traversal can be pending on an aggregate.* An iterator keeps track of its own traversal state. Therefore you can have <u>*more than one traversal in progress at once*</u>.

Implementations 1/2

- *Who controls the iteration? A* fundamental issue is deciding which party controls the iteration, the *iterator or the client that uses the iterator*. When the client controls the iteration, the iterator is called an **external iterator,** and when the iterator controls it, the iterator is an **internal iterator**
- *Who defines the traversal algorithm?* The aggregate might define the traversal algorithm and *use the iterator to store just the state of the iteration*. We call this kind of iterator a **cursor,** to point to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.
- 3. *How robust is the iterator?* It can be dangerous to modify an aggregate while you're traversing it. If elements are added or deleted from the aggregate, you might access element twice or missing it completely. A simple solution is *to copy the aggregate and traverse the copy*, but that's too expensive to do in general. **A robust iterator** ensures that insertions and removals won't interfere with traversal, without copying the aggregate.

Implementations 2/2

- *4. Additional Iterator operations* - SkipTo operation is useful for sorted or indexed collections. SkipTo positions the iterator to an object matching specific criteria.

- *5. Iterators for composites.* External iterators can be difficult to implement over recursive aggregate structures like those in the Composite (GoF163) pattern, because *a position in the structure may span many levels of nested aggregates*. Therefore an *external iterator has to store a path through the Composite to keep track of the current object*. Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack. If the nodes in a Composite have an interface for moving from a node to its siblings, parents, and children, then a cursor-based iterator may offer a better alternative.

Example - Enumerations in Java [3]

- The Enumeration type is built into the Vector and Hashtable classes. Rather than the Vector and Hashtable implementing the two methods of the Enumeration directly, both classes contain an *elements* method that returns an Enumeration of that class's data:

```
public Enumeration elements();
```

- This *elements()* method is really a kind Factory method that produces instances of an Enumeration class.
- Then, you move through the list with the following simple code:

```
Enumeration e = vector.elements();
while (e.hasMoreElements()) {
   String name = (String)e.nextElement();
   System.out.println(name);
}
```

- In addition, the Hashtable also has the *keys* method, which returns an enumeration of the keys to each element in the table:

```
public Enumeration keys();
```

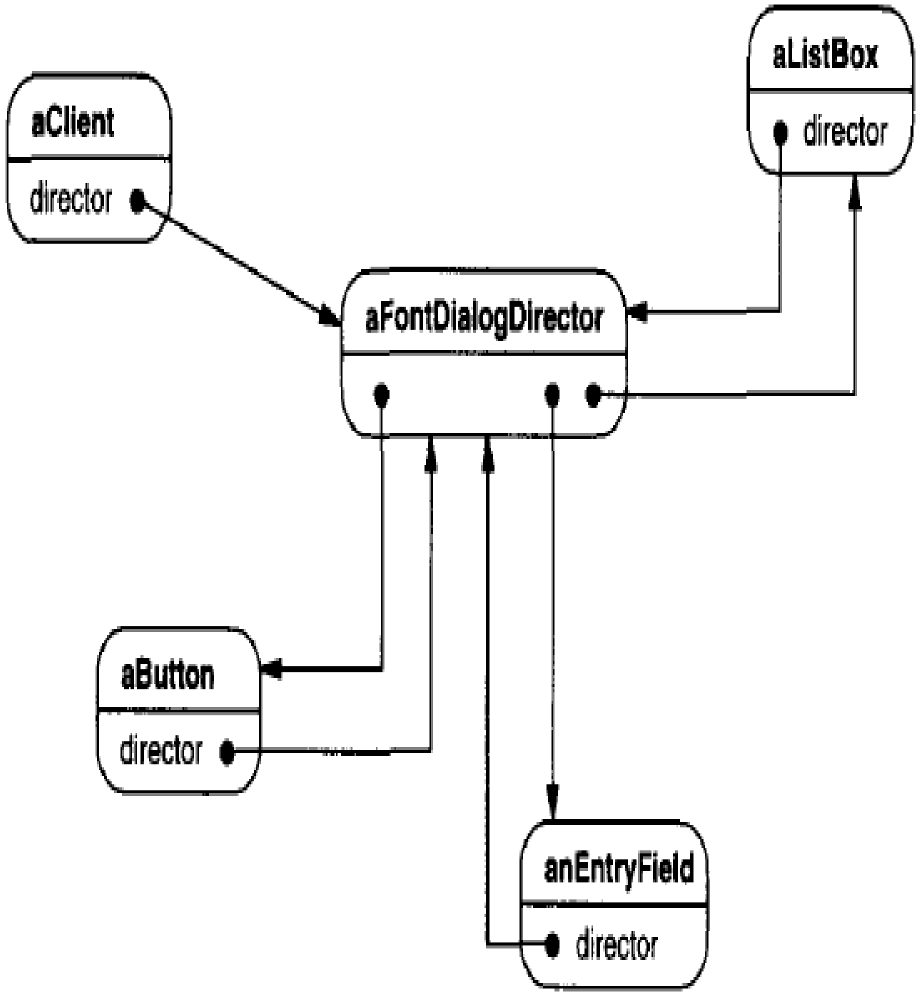
- This is the preferred style for implementing Enumerations in Java and has the advantage that you can have any number of simultaneous active enumerations of the same data.

The Mediator Pattern [1]

- **Intent** - defines 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.
- **Motivation – (1)** partitioning a system into *many objects generally enhances reusability* but lots of interconnections make it less likely that an object can *work without the support of others* - the system acts as though it were monolithic. **(2)** it can be *difficult to change the system's behavior* in any significant way, since behavior is distributed among many objects.
- **Example** – dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields, often with dependencies between the widgets - a *button* gets disabled when a certain *entry field* is empty; selecting an entry in a list of choices called a *list box* might change the contents of an *entry field*, etc. So, *widget classes have to be customized to reflect dialog-specific dependencies*. Customizing them individually *by subclassing will be tedious*, since many classes are involved.

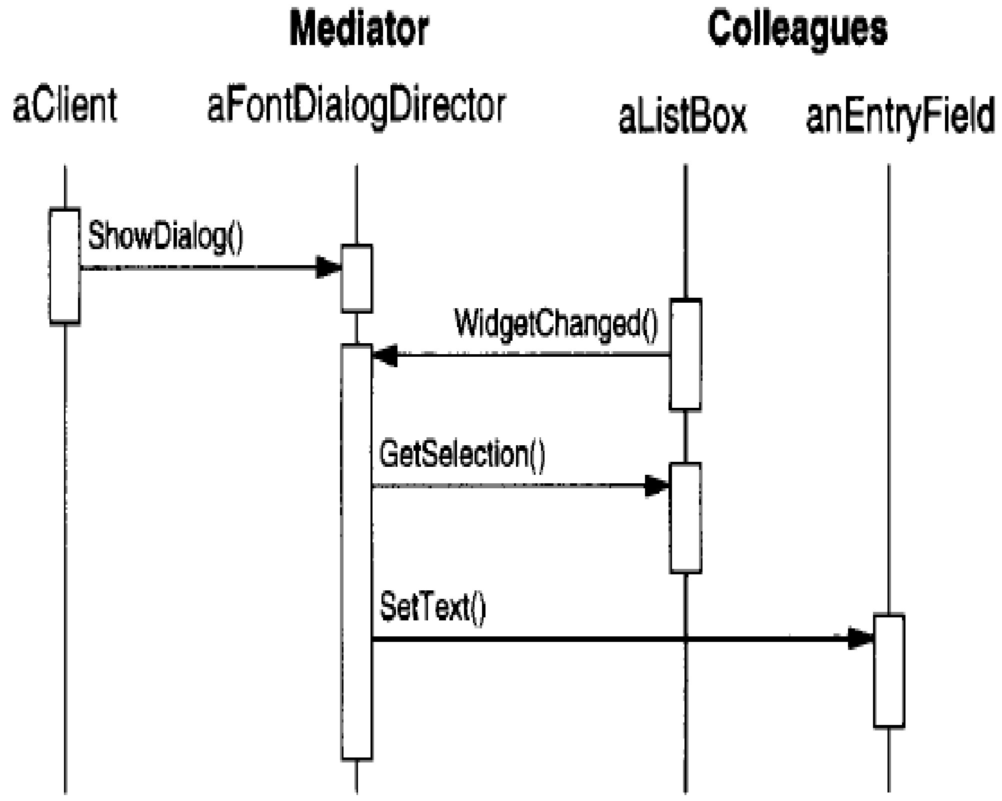
The Solution

- You can avoid these problems by encapsulating collective behavior in a separate **mediator** object. *A mediator is responsible for controlling and coordinating the interactions of a group of objects.*
- The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby *reducing the number of interconnections*.
- For example, **FontDialogDirector** can be the mediator between the widgets in a dialog box. A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction. It acts as a *hub of communication* for widgets ->



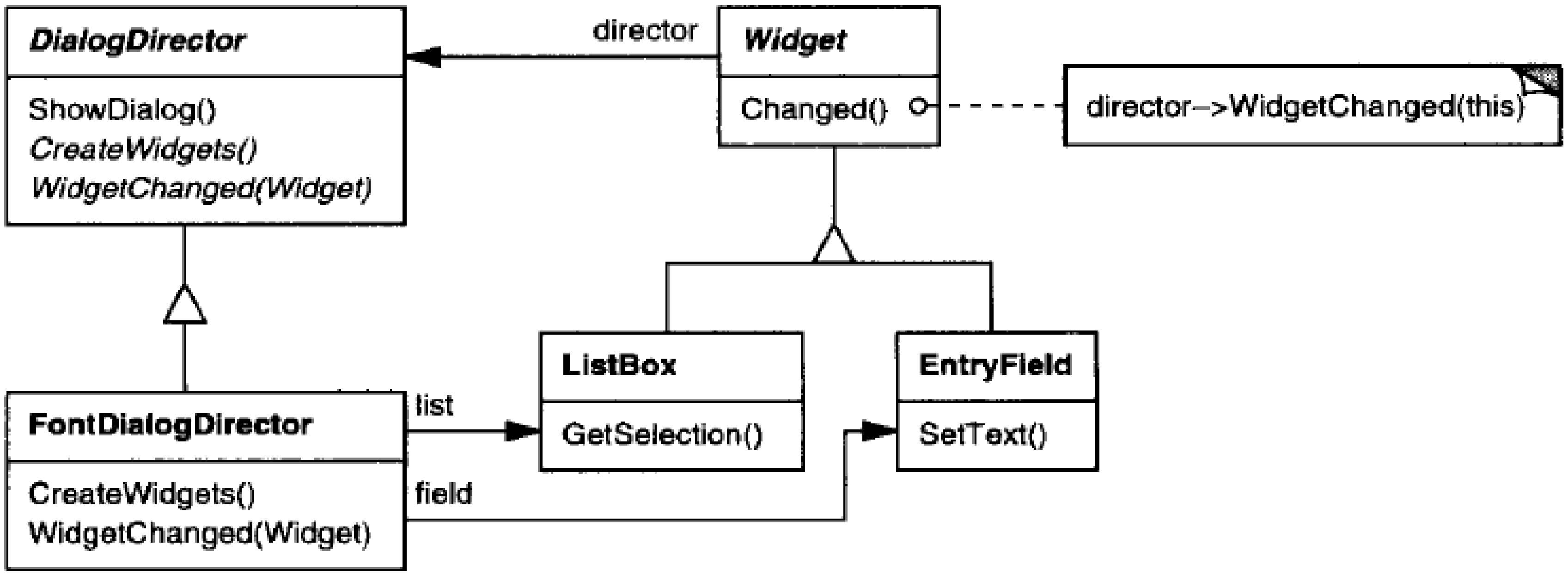
Interactions

1. The list box tells its director that it's changed.
2. The director gets the selection from the list box.
3. The director passes the selection to the entry field.
4. Now that the entry field contains some text, the director enables button(s) for initiating an action.



The director mediates between the list box and the entry field. <u>Widgets communicate with each other only indirectly, through the director</u>. They don't have to know about each other; all they know is the director. Furthermore, because the behavior is localized in one class, it can be changed or replaced by extending or replacing that class.

- ***DialogDirector*** is an abstract class that defines the overall behavior of a dialog. Clients call the **ShowDialog** operation to display the dialog on the screen.
  - ***CreateWidgets*** is an abstract operation for creating the widgets of a dialog.
  - ***WidgetChanged*** is another abstract operation; widgets call it to inform their director that they have changed.
- ***DialogDirector*** subclasses override ***CreateWidgets*** to create the proper widgets, and they override ***WidgetChanged*** to handle the changes.

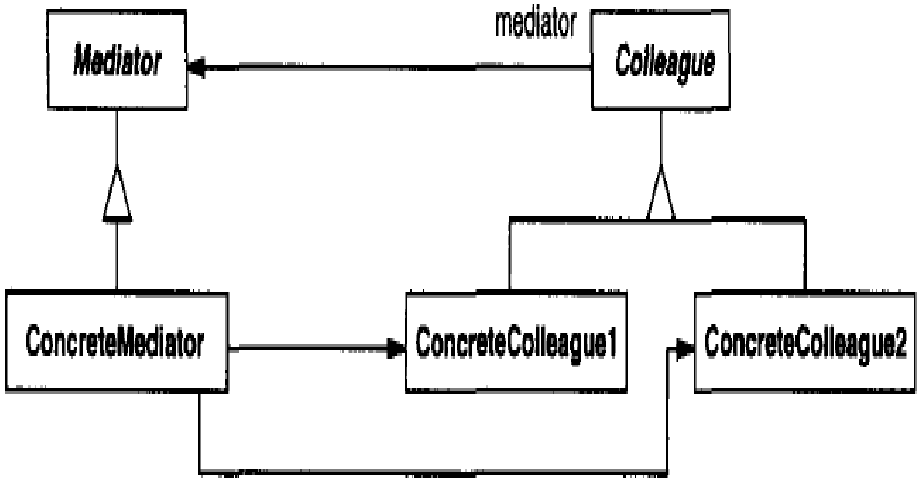


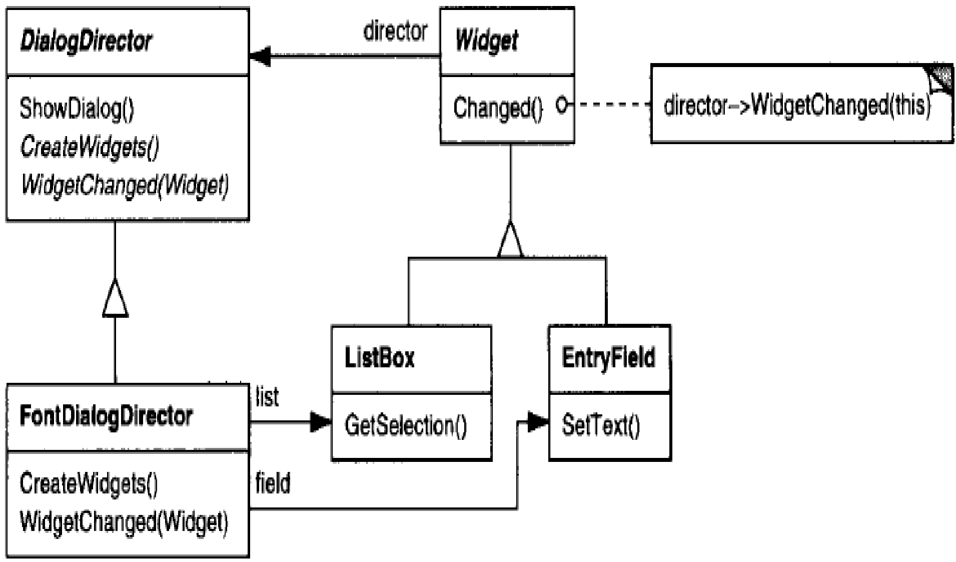
Applicability

Use the Mediator pattern 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*.

Structure and Participants





- **Mediator** (DialogDirector)
  - - defines an interface for communicating with Colleague objects.
- **ConcreteMediator** (FontDialogDirector)
  - - implements cooperative behavior by coordinating Colleague objects.
  - - knows and maintains its colleagues.
- **Colleague classes** (ListBox, EntryField)
  - - each Colleague class knows its Mediator object.
  - - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Consequences (benefits and drawbacks)

- 1. *The Mediator limits subclassing. A* mediator *localizes behavior that otherwise would be distributed* among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.
- 2. *It decouples colleagues.* A mediator promotes *loose coupling between colleagues*. You can vary and reuse Colleague and Mediator classes independently.
- 3. *It simplifies object protocols.* A mediator *replaces many-to-many interactions with one-to-many* interactions between the mediator and its colleagues.
- 4. *It abstracts how objects cooperate.* Making mediation an independent concept and encapsulating it in an object lets you focus *on how objects interact apart from their individual behavior*.
- 5. *It centralizes control.*

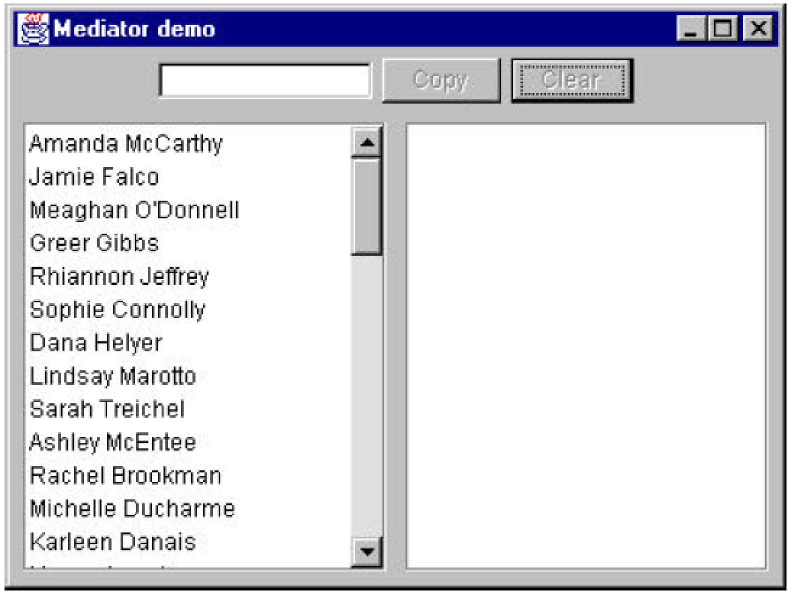
Implementation

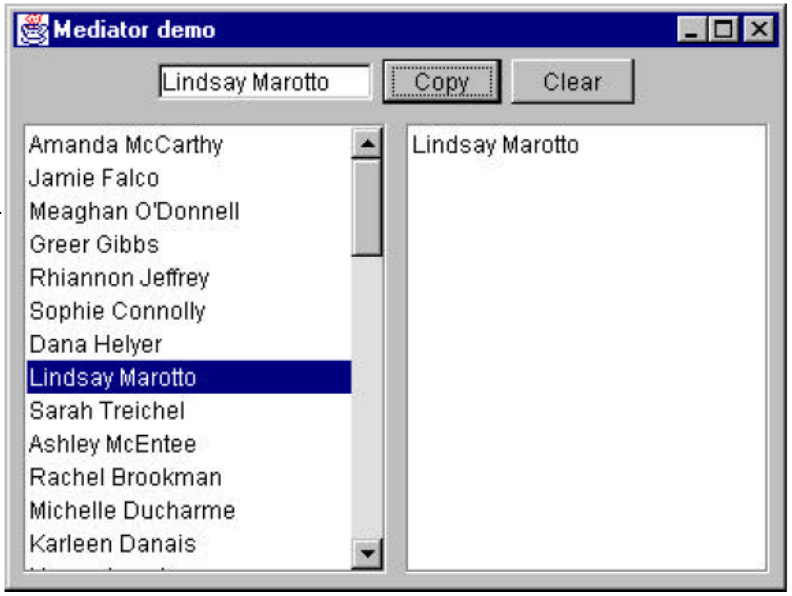
- 1. *Omitting the abstract Mediator class.* There's no need to define an abstract Mediator class when colleagues work with *only one mediator*. The abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa.

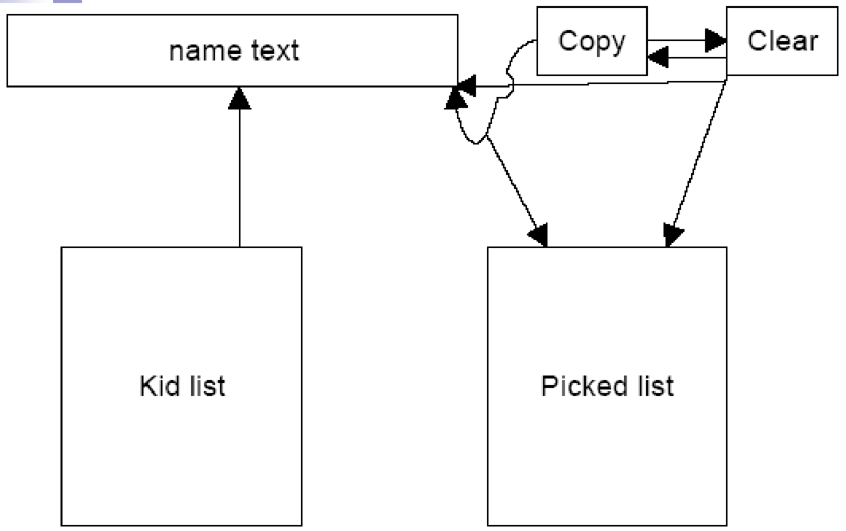
- 2. *Colleague-Mediator communication.* Colleagues have to communicate with their mediator when an event of interest occurs. One approach is to implement the Mediator as an Observer using the Observer (GoF293) pattern. Colleague classes act as *Subjects, sending notifications to the mediator whenever they change state*. The mediator responds by propagating the effects of the change to other colleagues.

Java Example [3]

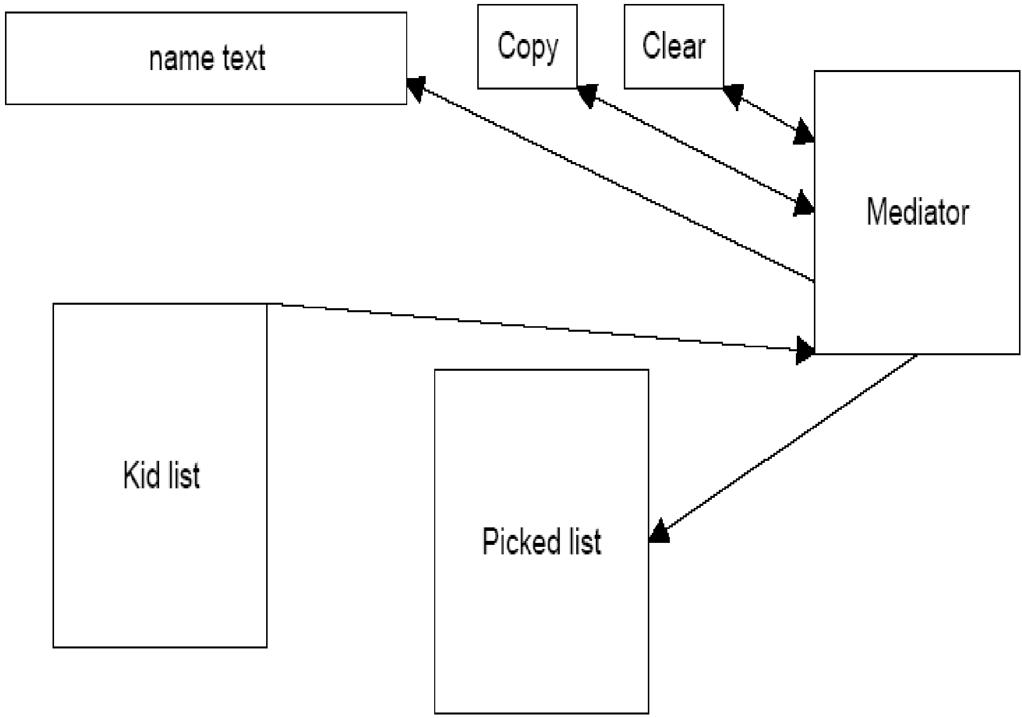
- When the program starts, the Copy and Clear buttons are disabled.
- When you select one of the names in the left-hand list box, it is copied into the text field for editing, and the *Copy* button is enabled.
- When you click on *Copy,* that text is added to the right hand list box, and the *Clear* button is enabled.
- If you click on the *Clear* button, the right hand list box and the text field are cleared, the list box is deselected and the two buttons are again disabled.







← Two Interactions' Designs ↓



The Mediator pattern simplifies this system by being the only class that is aware of the other classes in the system.

Each of the controls that the Mediator communicates with is called a Colleague. Each Colleague informs the Mediator when it has received a user event, and the Mediator decides which other classes should be informed of this event.

Sample Code

```
Mediator med = new Mediator();
kidList = new KidList(med);
tx = new KTextField(med);
copy = new CopyButton(this, med);
clear = new ClearButton(this, med);
med.init();

    //the buttons use the Command pattern and register themselves
    //with  the Mediator during their initialization.
    public class CopyButton extends JButton implements
    Command {
        Mediator med; //copy of the Mediator
        public CopyButton(ActionListener fr, Mediator md) {
            super("Copy"); //create the button
            addActionListener(fr); //add its listener
            med = md; //copy in Mediator instance
            med.registerMove(this); //register with the Mediator
        }
        public void Execute() { //execute the copy
            med.copy();
        }
    }
```

```
public class KidList extends JawtList implements ListSelectionListener {
        KidData kdata; //reads the data from the file
        Mediator med; //copy of the mediator
        public KidList(Mediator md) {
                super(20); //create the JList
                kdata = new KidData ("50free.txt");
                fillKidList(); //fill the list with names
                med = md; //save the mediator
                med.registerKidList(this);
                addListSelectionListener(this);
        }
        public void valueChanged(ListSelectionEvent ls) {
                //if an item was selected pass on to mediator
                JList obj = (JList)ls.getSource();
                if (obj.getSelectedIndex() >= 0)
                                med.select();
        }
        private void fillKidList() {
                Enumeration ekid = kdata.elements();
                while (ekid.hasMoreElements()) {
                  Kid k =(Kid)ekid.nextElement();
                  add(k.getFrname()+" "+k.getLname());
                }
        }
}
```

```
//The text field is simple, since all it does is register itself with the mediator.
public class KTextField extends JTextField {
        Mediator med;
        public KTextField(Mediator md) {
                super(10);
                med = md;
                med.registerText(this);
        }
}
```

•The general point of all these classes is that each knows about the Mediator and tells the Mediator of its existence so the Mediator can send commands to it when appropriate.

•The Mediator itself is very simple. It supports the Copy, Clear and Select methods, and has register methods for each of the controls:

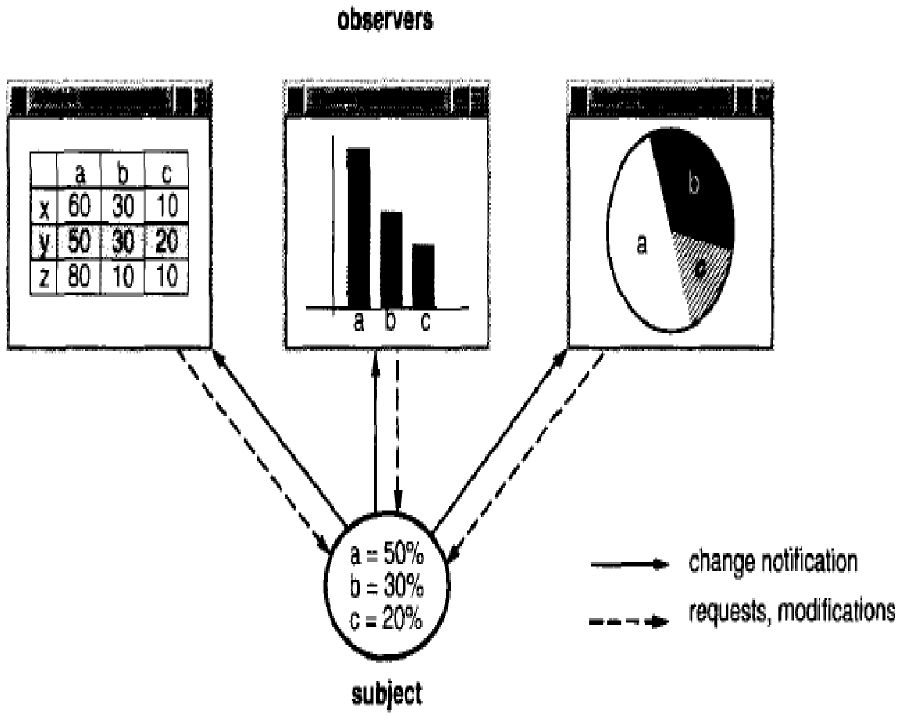
```
public class Mediator {
    private ClearButton clearButton;
    private CopyButton copyButton;
    private KTextField ktext;
    private KidList klist;
    private PickedKidsList picked;
    public void copy() {
        picked.add(ktext.getText());
        //copy text
        clearButton.setEnabled(true);
        //enable Clear
    }
    //------------------------------------
    public void clear() {
        ktext.setText(""); //clear text
        picked.clear(); //and list
        //disable buttons
        copyButton.setEnabled(false);
        clearButton.setEnabled(false);
        klist.clearSelection(); //deselect list
    }
public void select() {
    String s = (String)klist.getSelectedValue();
    ktext.setText(s); //copy text
    copyButton.setEnabled(true); //enable Copy
}
//-----------copy in controls--------------------
public void registerClear(ClearButton cb) {
    clearButton = cb;
}
public void registerCopy(CopyButton mv){
    copyButton = mv;
}
public void registerText(KTextField tx) {
    ktext = tx;
}
public void registerPicked(PickedKidsList pl) {
    picked = pl;
}
public void registerKidList(KidList kl) {
  klist = kl; }
}
```

The Observer Pattern [1]

- **Intent -** defines 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. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
- For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data

Motivating Example

- Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, but when the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



This behavior implies that *<u>the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state</u>*. And there's *<u>no reason to limit the number of dependent objects</u>* to two; there may be any number of different user interfaces to the same data.

The Solution

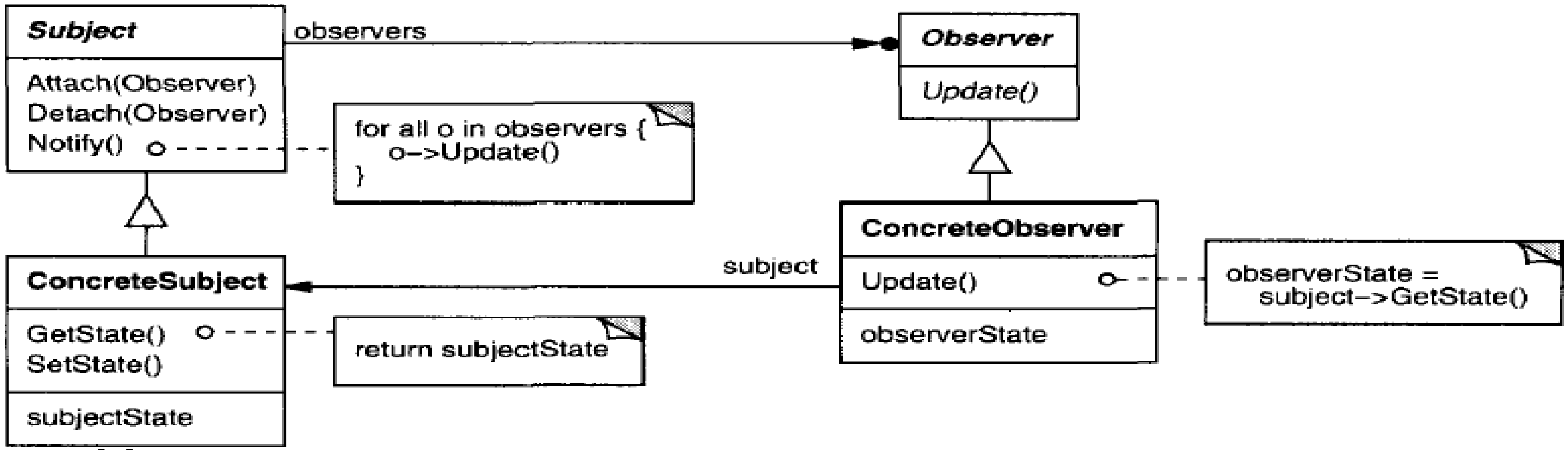
- The Observer pattern describes how to establish these relationships. The key objects in this pattern are the observed **subject** and the **observer.**
  - **A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state.**
  - **In response, each observer will query the subject to synchronize its state with the subject's state.**
- This kind of interaction is also known as **publish-subscribe.**
  - The *subject is the publisher of notifications*. It sends out these notifications without having to know who its observers are.
  - Any number of *observers can subscribe to receive notifications*.

Applicability

Use the Observer pattern in any of the following situations:

- When *an abstraction has two aspects, one dependent on the other*. *Encapsulating these aspects in separate objects lets you vary and reuse them independently*.
- 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. In other words, you don't want these objects to be tightly coupled.

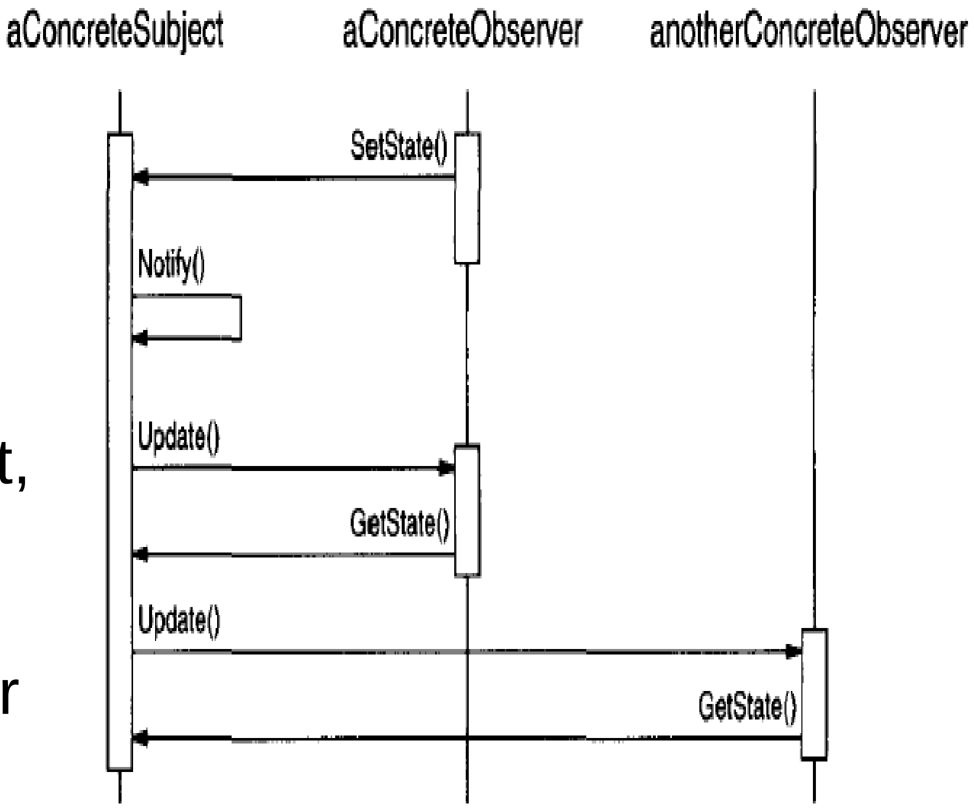
Structure and Participants



- **Subject**
  - - knows its observers. Any number of Observer objects may observe a subject.
  - - provides an interface for attaching and detaching Observer objects.
- **Observer**
  - - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - - stores state of interest to ConcreteObserver objects.
  - - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - - maintains a reference to a ConcreteSubject object.
  - - stores state that should stay consistent with the subject's.
  - - implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object ***may*** query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.



Consequences

- 1. The Observer pattern lets you ***vary subjects and observers independently***. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.
- 2. ***Abstract coupling between Subject and Observer***. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact.
- 3. ***Support for broadcast communication***. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it.
- 4. ***Unexpected updates***. The simple update protocol provides no details on what changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Implementation Issues 1/3

- *Mapping subjects to their observers.* The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject, or an associative look-up (a hash table) to maintain the subject-to-observer mapping.
- *Observing more than one subject.* It might make sense in some situations for an observer to depend on more than one subject. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
- *Dangling references to deleted subjects.* Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is to be deleted so that they can reset their reference to it.

Implementation Issues 2/3

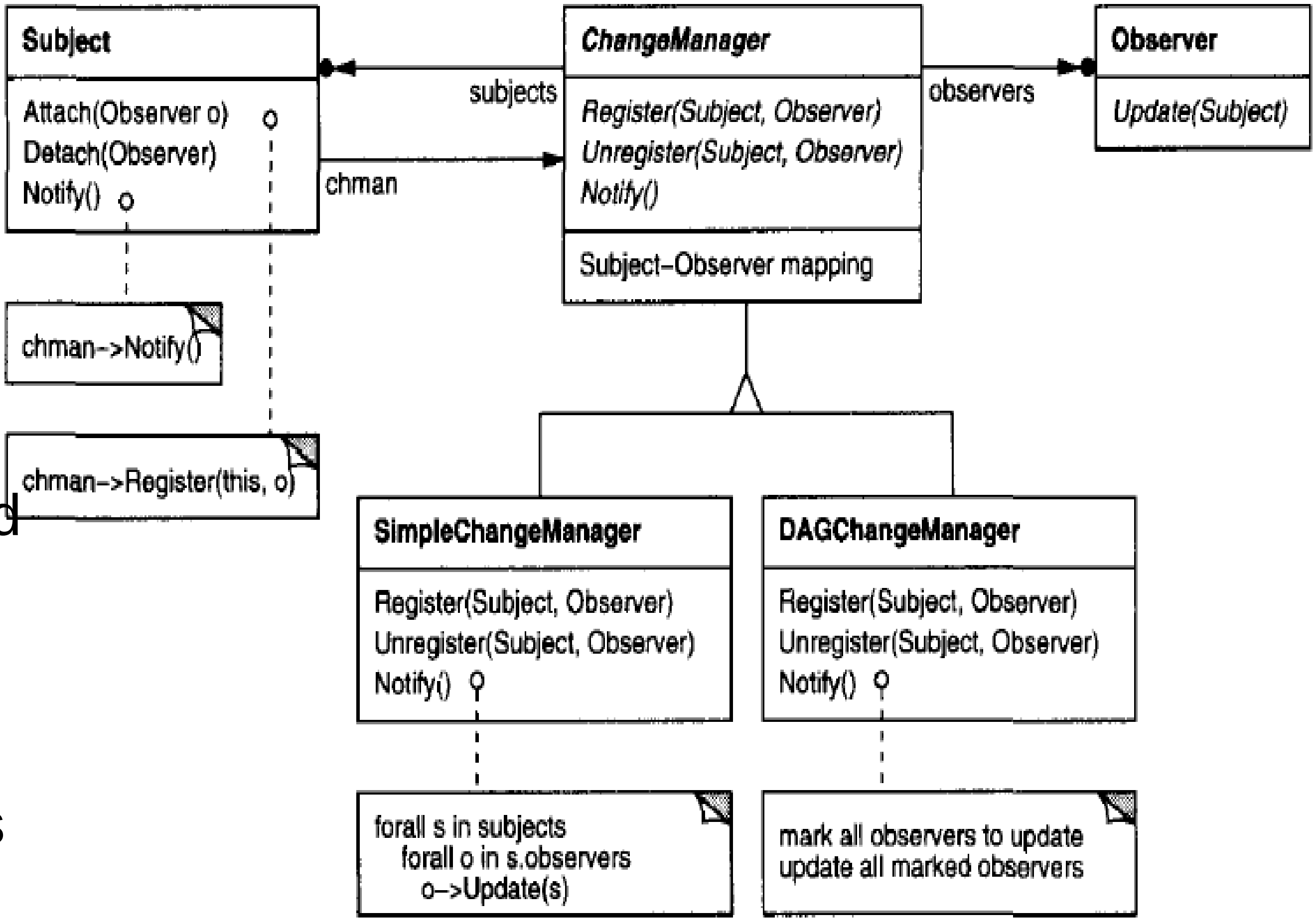
- *Who triggers the update?* The subject and its observers rely on the notification mechanism to stay consistent. But *what object actually calls Notify to trigger the update*?
  - (a) Have *state-setting operations on Subject calling Notify after they change the subject's state* – good as **clients don't have to remember to call Notify on the subject**, BUT … **several consecutive operations will cause several consecutive updates**, which may be inefficient.
  - (b) Make *clients responsible for calling Notify at the right time* – good as **the client can wait to trigger the update until after a series of state changes has been made**, thereby avoiding needless intermediate updates, BUT …. **clients have an added responsibility to trigger the update**. That makes errors more likely, since clients might forget to call Notify.
- *Making sure Subject state is self-consistent before notification (i.e.,* before calling Notify), because observers query the subject for its current state in the course of updating their own state.

Implementation Issues 3/3

- *Avoiding observer-specific update protocols: the push and pull models.* Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely.
  - **push model -** the <u>*subject sends to observers detailed information about the change*</u>, whether they want it or not.
  - **pull model -** the <u>*subject sends nothing but the most minimal notification*</u>, and <u>*observers MAY ask for details explicitly thereafter*</u>.
- *Specifying modifications of interest explicitly.* You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. One way to support this uses the notion of **aspects** for Subject objects.
- *Encapsulating complex update semantics.* When the dependency relationship between (many) subjects and (many) observers is particularly complex, an object (**Change-Manager)** that maintains these relationships might be required.

Example – usage of Change Manager

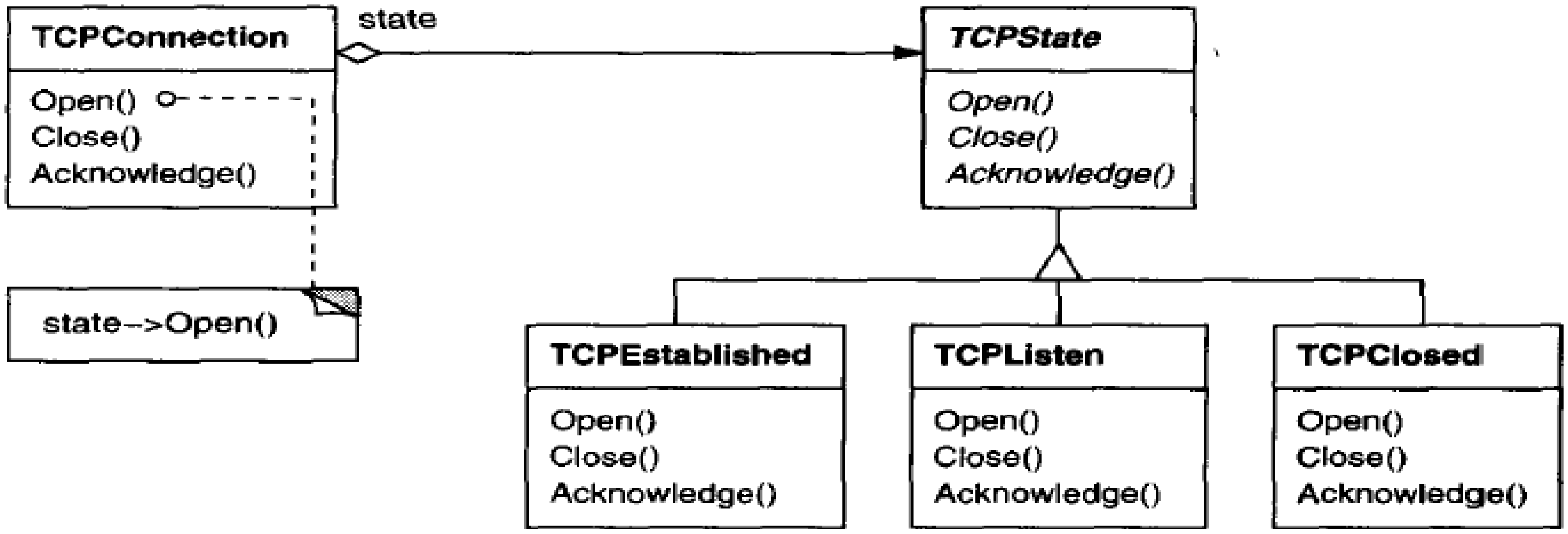
- ***ChangeManager*** maps a subject to its observers and provides an interface to maintain this mapping – it eliminates the need for subjects to maintain references to their observers and vice versa.
- it defines a particular update strategy.
- it updates all dependent observers at the request of a subject.



The State Pattern [1]

- **Intent -** allows an object to alter its behavior when its internal state changes. The object will appear as an instance of different class.
- **Also Known As -** Objects for States
- **Motivation -** consider a class ***TCPConnection*** that represents a network connection. A TCP Connection 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: the effect of an Open request depends on whether the connection is in its Closed state or its Established state.
- The State pattern describes how TCPConnection can exhibit different behavior in each state. The key idea in this pattern is to *introduce an abstract class called* ***TCPState*** *to represent the states of the network connection*. The TCPState class declares an interface common to all classes that represent different operational states. Subclasses of TCPState implement state-specific behavior.

- For example, the classes TCPEstablished and TCPClosed implement behavior particular to the Established and Closed states of TCPConnection.
- The class TCPConnection maintains a state object (an instance of a subclass of TCPState) that represents the current state of the TCP connection.
- ***The class TCPConnection delegates all state-specific requests to this state object***. TCPConnection uses its TCPState subclass instance to perform operations particular to the state of the connection.
- ***Whenever the connection changes state, the TCPConnection object changes the state object it uses***. When the connection goes from established to closed, for example, TCPConnection will replace its TCPEstablished instance with a TCPClosed instance.

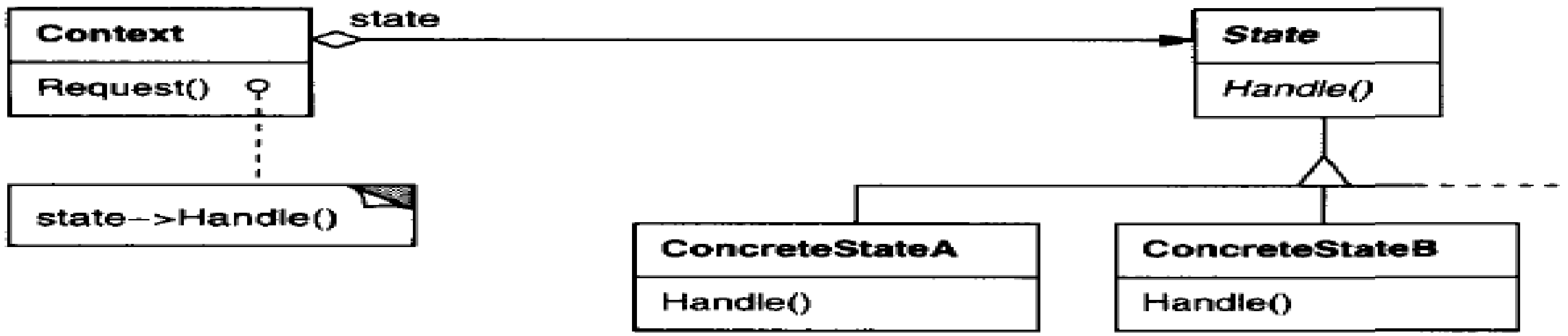


Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must *change its behavior at run-time depending on that state*.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. ***The State pattern puts each branch of the conditional in a separate class***. This lets you *treat the object's state as an object in its own right that can vary independently from other objects.*

Structure and Participants



- **Context** (TCPConnection)
  - - defines the interface of interest to clients.
  - - maintains an instance of a ConcreteState subclass that defines the current state.
- **State** (TCPState)
  - - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
  - - each subclass implements a behavior associated with a state of the Context.

Consequences

- *It localizes state-specific behavior and partitions behavior for different states.* The State pattern puts all behavior associated with a particular state into one object. Because all state specific code lives in a State subclass, *new states and transitions can be added easily by defining new subclasses*.
- *It makes state transitions explicit.* When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables. Introducing *separate objects for different states makes the transitions more explicit*.
- *State objects can be shared.* If State objects have *no instance variables* - that is, the state they represent is encoded entirely in their type - then *contexts can share a State object*. When states are shared in this way, they are essentially ***flyweights*** (GoF195) with no intrinsic state, only behavior.

Implementation 1/2

- *Who defines the state transitions?* The State pattern does not specify *which participant defines the criteria for state transitions*. If the criteria are fixed **(1)**, then **they can be implemented entirely in the Context**. It is generally more flexible and appropriate, however, **(2) let the State subclasses themselves to specify their successor state and when to make the transition**. This requires *adding an interface to the Context that lets State objects set the Context's current state explicitly*.
- *A table-based alternative* - main advantage: **regularity** - you can change the transition criteria by modifying data instead of changing program code. But:
  - A table look-up is often *less efficient* than a (virtual) function call.
  - Putting transition logic into a uniform, tabular format makes the transition criteria *less explicit* and therefore harder to understand.
  - It's usually *difficult to add actions* to accompany the state transitions.

Implementation 2/2

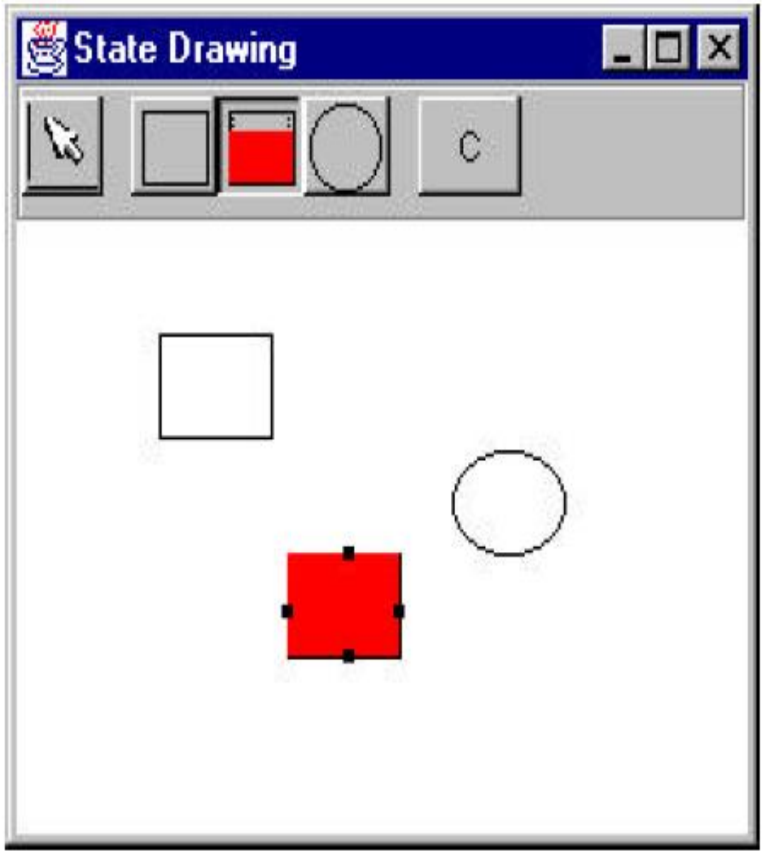
- *Creating and destroying State objects.* A common implementation trade-off worth considering is whether:
  - **(1) to create State objects *only when they are needed and destroy them thereafter* (preferable when the states that will be entered aren't known at run-time, *and* contexts change state infrequently; avoids creating objects that won't be used)**

  versus

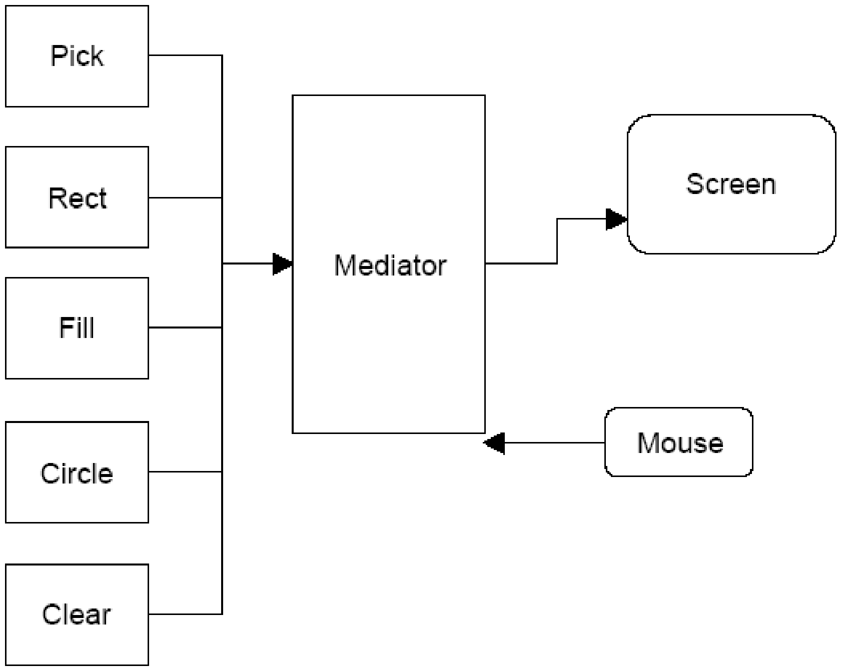
  - **(2) creating them *ahead of time and never destroying them* - better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly.**
- *Using dynamic inheritance.* Changing the behavior for a particular request could be accomplished by changing the object's class at run-time, but this is not possible in most OOP languages.

Sample Java Code [3]

- Our program will have toolbar buttons for Select, Rectangle, Fill, Circle and Clear.
- Each one of the tool buttons *does something rather different when it is selected and you click or drag your mouse across the screen.* Thus, the *state* of the graphical editor affects the behavior the program should exhibit. This suggests some sort of design using the State pattern.



- Initially we might design our program like this, with a Mediator managing the actions of 5 command buttons ->
- However, this initial design puts the entire burden of maintaining the state of the program on the Mediator, and we know that the main purpose of a Mediator is to coordinate activities between various controls, such as the buttons.
- Keeping the state of the buttons inside the Mediator can make it too complicated as well as leading to a set of *if* or *switch* tests making the program difficult to read and maintain.



Further, this ***set of large, monolithic conditional statements might have to be repeated for each action the Mediator interprets***, such as mouseUp, mouseDrag, rightClick and so forth.

Program Functionality

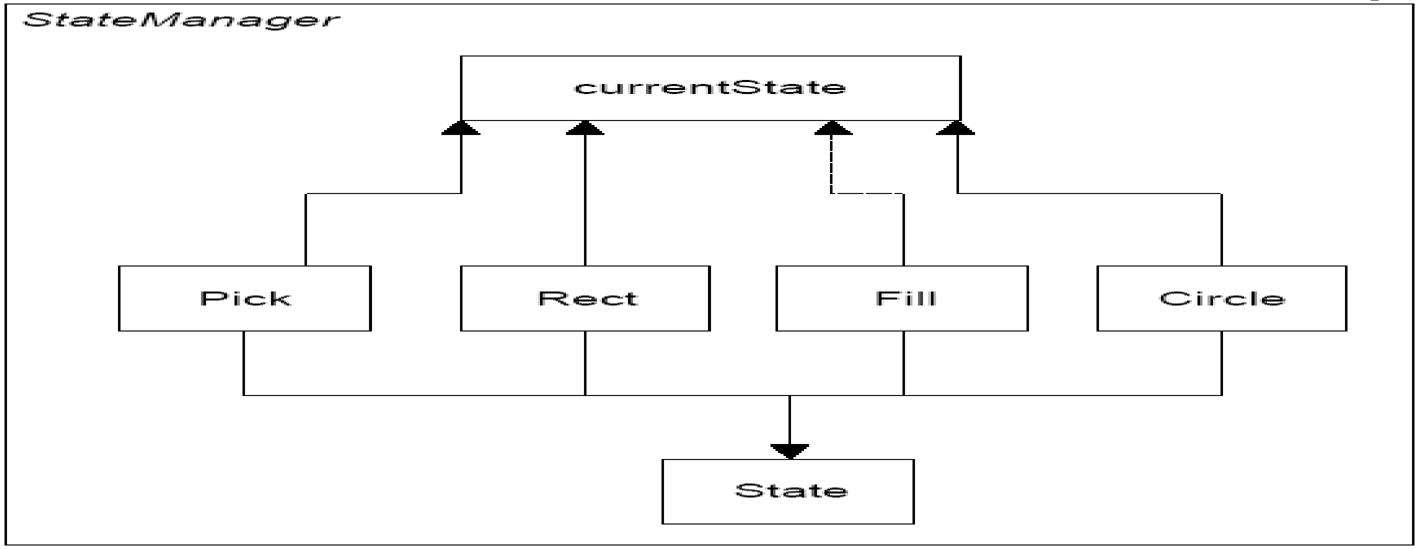
1. If the **Pick** button is selected, clicking inside a drawing element should cause it to be highlighted or appear with "handles". If the mouse is dragged and a drawing element is already selected, the element should move on the screen.
2. If the **Rect** button is selected, clicking on the screen should cause a new rectangle drawing element to be created.
3. If the **Fill** button is selected *<u>and a drawing element is already selected</u>*, that element should be filled with the current color. *<u>If no drawing is selected</u>*, then clicking inside a drawing should fill it with the current color.
4. If the **Circle** button is selected, clicking on the screen should cause a new circle drawing element to be created.
5. If the **Clear** button is selected, all the drawing elements are removed.

Common: our actions *<u>use the mouse click event to cause actions</u>*. One uses the mouse drag event to cause an action. Thus, **we really want to create a system that can help us redirect these events based on which button is currently selected**.

```
public class State {
        public void mouseDown(int x, int y) {}
        public void mouseUp(int x, int y) {}
        public void mouseDrag(int x, int y) {}
        public void select(Drawing d, Color c) {} //for Fill event
}
```

// none of the cases need all of these events, we gave our base class //empty methods rather than creating an abstract base class.

//now we create 4 derived *State* classes for *Pick*, *Rect*, *Circle* and *Fill* and //put instance of all of them inside a *StateManager* (*Context*) class which //sets the current state an executes methods on that state object.



A typical State object simply overrides those event methods that it must handle specially. For example, this is the complete Rectangle state object:

```
public class RectState extends State {
    private Mediator med; //save the Mediator
    public RectState(Mediator md) {
        med = md;
    }
    //-----------------------------------
    //create a new Rectangle where mouse clicks
    public void mouseDown(int x, int y) {
        med.addDrawing(new visRectangle(x, y));
    }
}
```

The RectState object simply tells the Mediator to add a rectangle drawing to the drawing list.

```
//the Circle state object tells the Mediator to add a circle to the draw in list:
public class CircleState extends State {
        private Mediator med; //save Mediator
        public CircleState(Mediator md) {
                med = md;
        }
        //Draw circle where mouse clicks
        public void mouseDown(int x, int y) {
                med.addDrawing(new visCircle(x, y));
        }
}
public class FillState extends State {
        private Mediator med; //save Mediator
        private Color color; //save current color
        public FillState(Mediator md) {
                med = md;
        }
        //Fill drawing if selected
        public void select(Drawing d, Color c) {
                        color = c;
                        if(d!= null)          { d.setFill(c); //fill that drawing }
                }
        //Fill drawing if you click inside one
        public void mouseDown(int x, int y) {
                Vector drawings = med.getDrawings();
                for(int i=0; i< drawings.size(); i++) {
                        Drawing d = (Drawing)drawings.elementAt(i);
                        if(d.contains(x, y)) d.setFill(color); //fill drawing
                }
        }
}
```

```
import java.awt.*;
public class StateManager {
        private State currentState;
        RectState rState; //states are kept here
        ArrowState aState;
        CircleState cState;
        FillState fState;
        public StateManager(Mediator med) {
                rState = new RectState(med); //create instances
                cState = new CircleState(med); //of each state
                aState = new ArrowState(med);
                fState = new FillState(med);
                currentState = aState;
        }
        //These methods are called when the tool buttons are selected
        public void setRect()     { currentState = rState; }
        public void setCircle()   { currentState = cState; }
        public void setFill()     { currentState = fState; }
        public void setArrow()  { currentState = aState; }
        public void mouseDown(int x, int y) { currentState.mouseDown(x, y); }
        public void mouseUp(int x, int y)     { currentState.mouseUp(x, y);}
        public void mouseDrag(int x, int y)  { currentState.mouseDrag(x, y);}
        public void select(Drawing d, Color c) { currentState.select(d, c); }
}
```

```
public Mediator() {
    startRect = false;
    dSelected = false;
    drawings = new Vector();
    undoList = new Vector();
    stMgr = new StateManager(this);
}
public void startRectangle() {
    stMgr.setRect(); //change to rectangle state
    arrowButton.setSelected(false);
    circButton.setSelected(false);
    fillButton.setSelected(false);
}
public void startCircle() {
    stMgr.setCircle(); //change to circle state
    rectButton.setSelected(false);
    arrowButton.setSelected(false);
    fillButton.setSelected(false);
}
```

<- The Mediator is the critical class, however, since it tells the StateManager when the current program state changes.