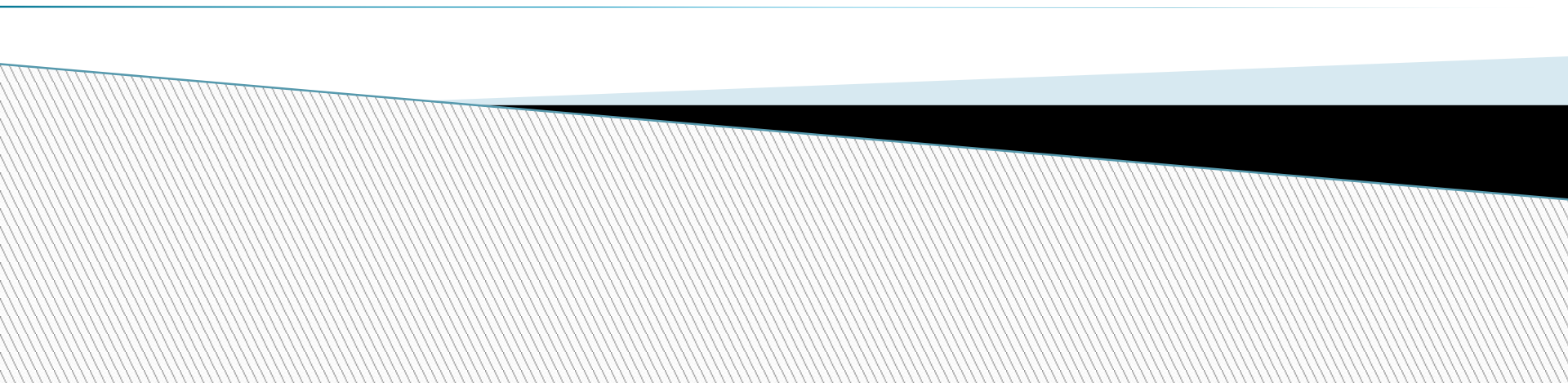


# Design Patterns - Behavioral



# Behavioral Patterns

# Behavioral Patterns

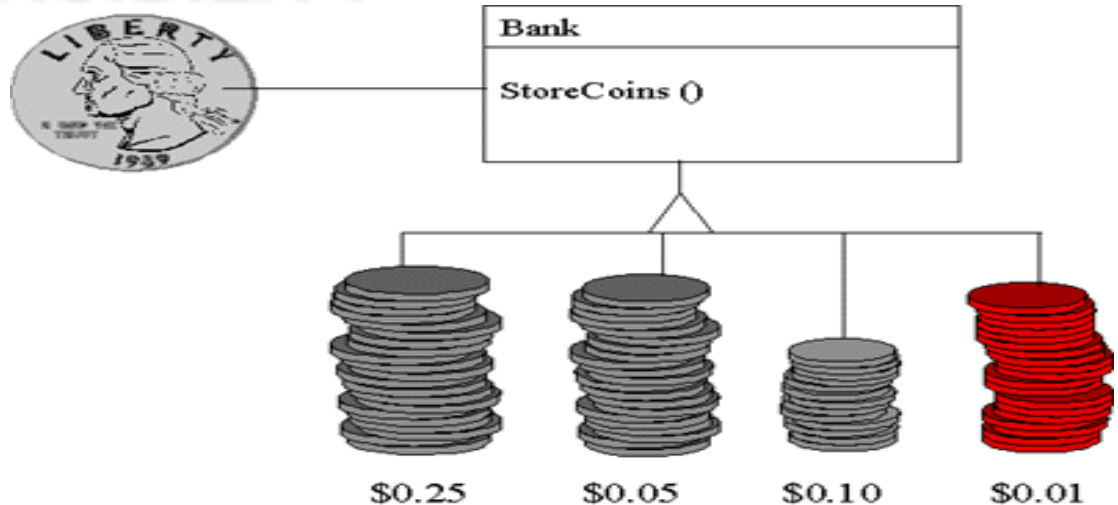
- ▶ Characterize ways in which classes or objects interact and distribute responsibility

• Chain of Responsibility	• State
• Command	• Strategy
• Iterator	• Template Method
• Mediator	• Visitor
• Memento	• Interpreter
• Observer	

# Chain of Responsibility

# CHAIN OF RESPONSIBILITY

## Object Diagram for *Chain of Responsibility* using Coin Sorting Example



The *Chain of Responsibility* pattern avoids coupling the sender of a request to the receiver, by giving more than one object a chance to handle the request. Mechanical coin sorting banks use the *Chain of Responsibility*. Rather than having a separate slot for each coin denomination coupled with receptacle for the denomination, a single slot is used. When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank.

# Intent

- ▶ Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

## Problem

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

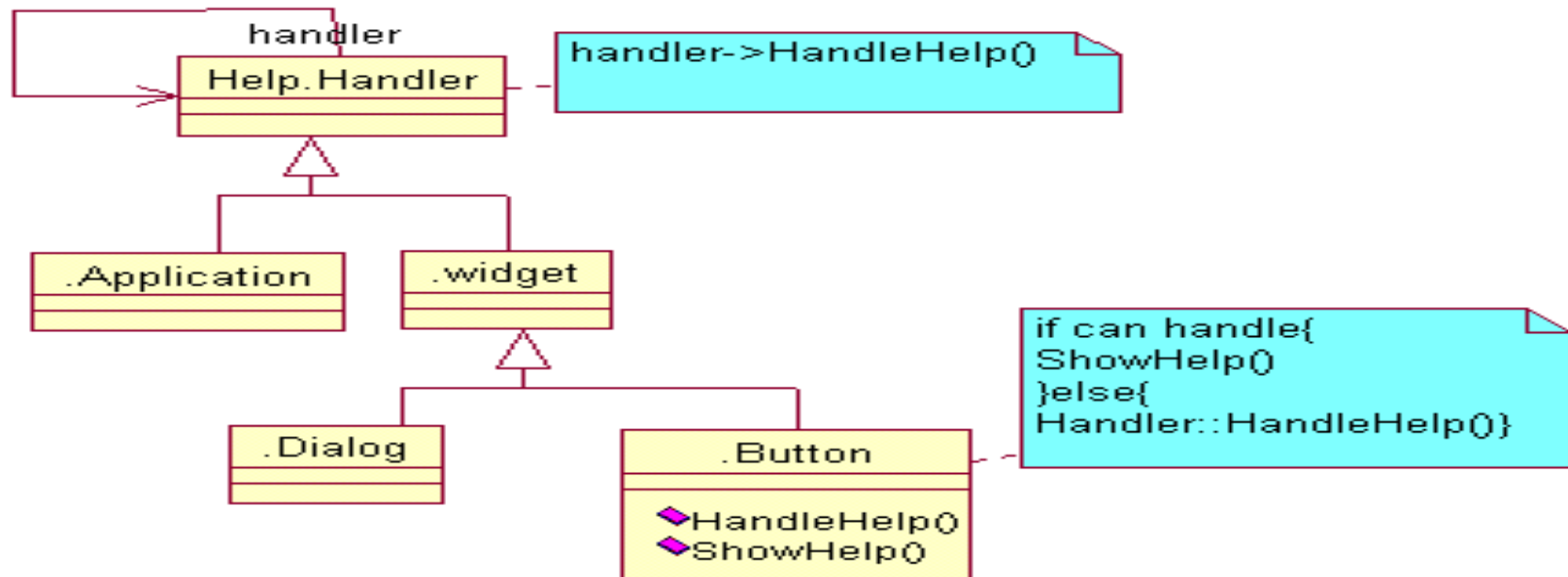
## Illustration: Problem

In a context-sensitive help facility for a graphical user interface. For example : If button on the dialog widget is selected as **Part of the Interface** - help specific to dialog will be displayed

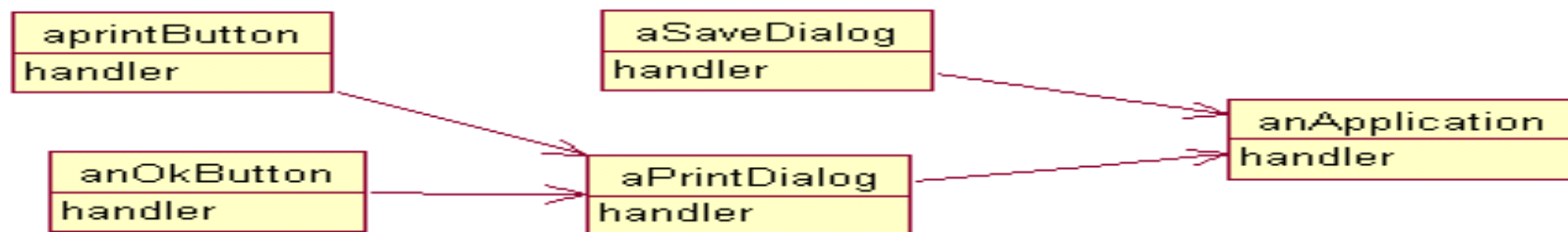
**Note** : If specific information on a widget is not present then help system should display more general help.

Hence it's natural to organize help information according to its generality-from the most specific to the most general.

The **problem** here is that the object that ultimately *provides* the help isn't known explicitly to the object (e.g., the button) that *initiates* the help request. What we need is a way to decouple the button that initiates the help request from the objects that might provide help information.



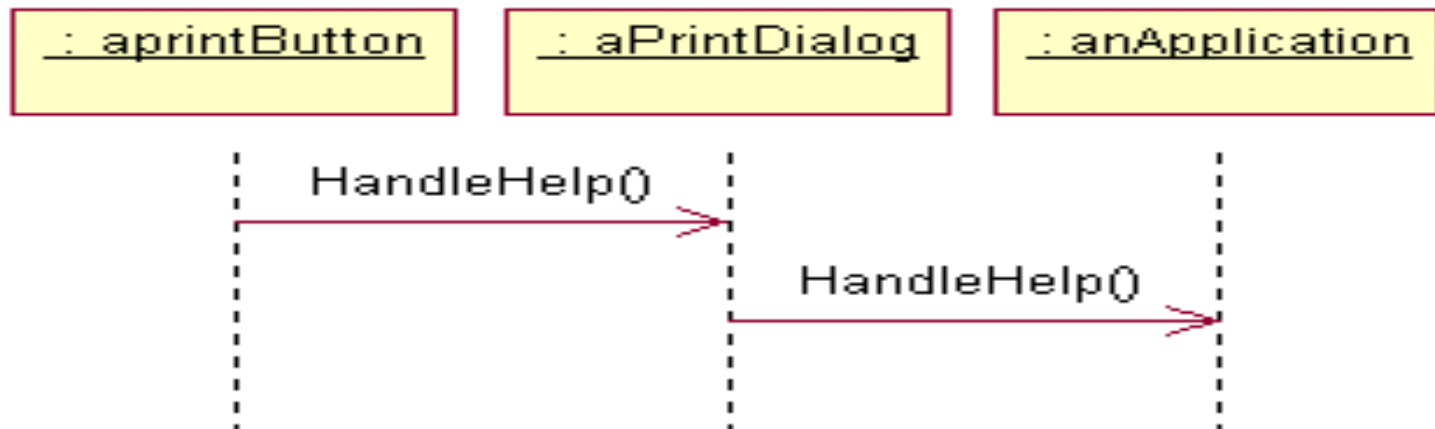
- ▶ The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it.





## Illustration: solution

The first object in the chain receives the request and either handles it or forwards it to the next candidate on the chain, which does likewise. The object that made the request has no explicit knowledge of who will handle it - we say the request has an implicit receiver.

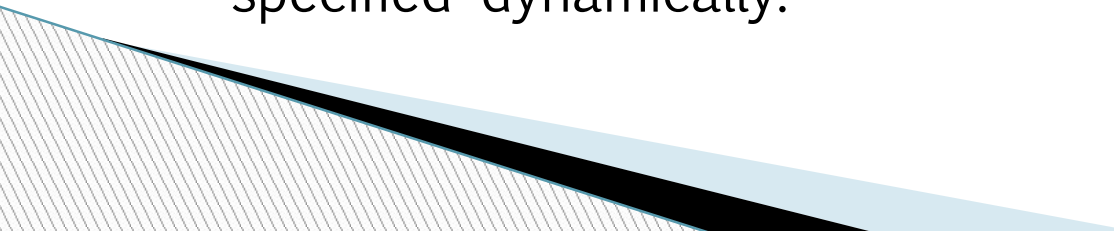


# Solution

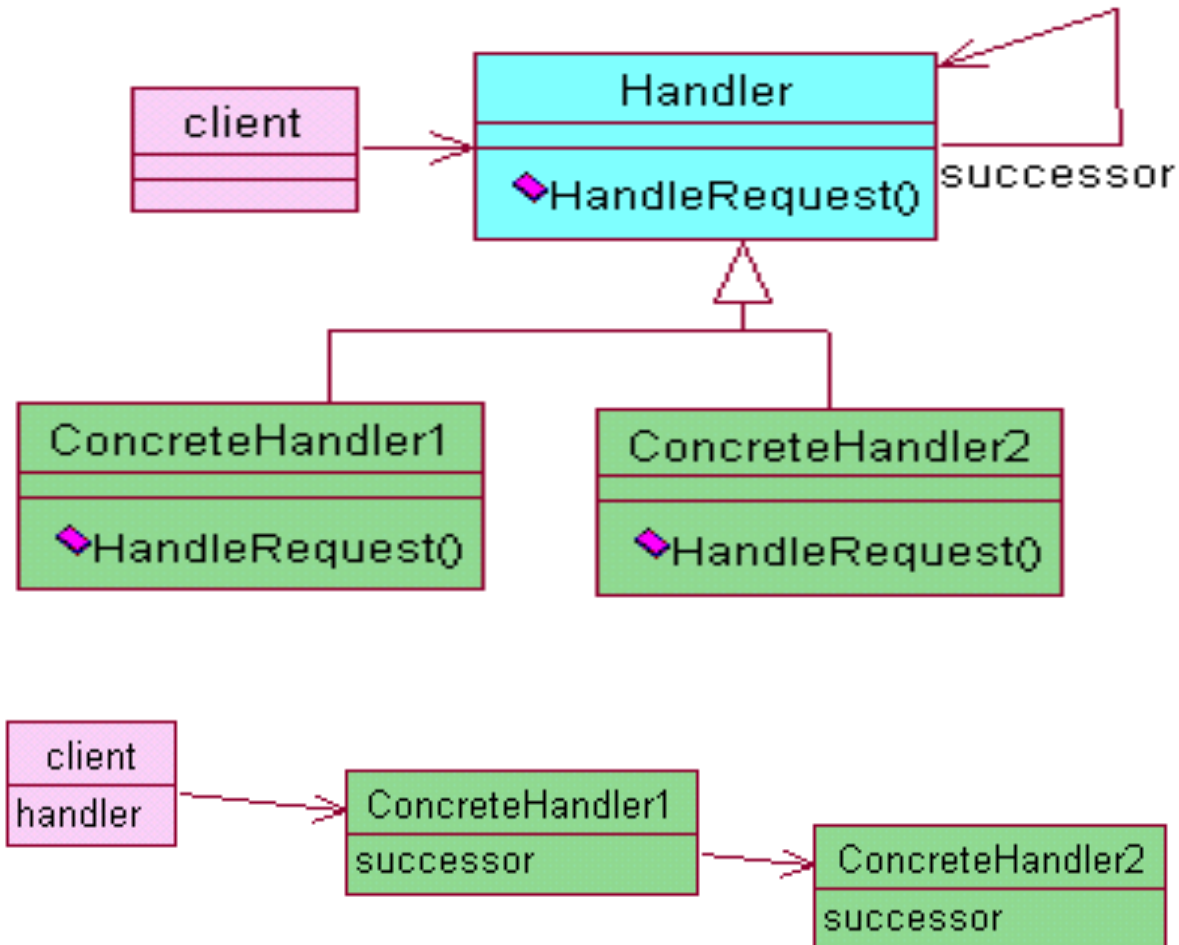
**Chain of Responsibility** deals with traversing an object tree from a node towards the root, in need to fulfill certain responsibility. If none of the objects in the chain provide the solution then the root will give a default solution.

## Applicability

### Use Chain of Responsibility when

- More than one object may handle a request, and the handler isn't known a priori, The handler should be ascertained automatically
  - 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.
- 

# Structure



# Collaborations

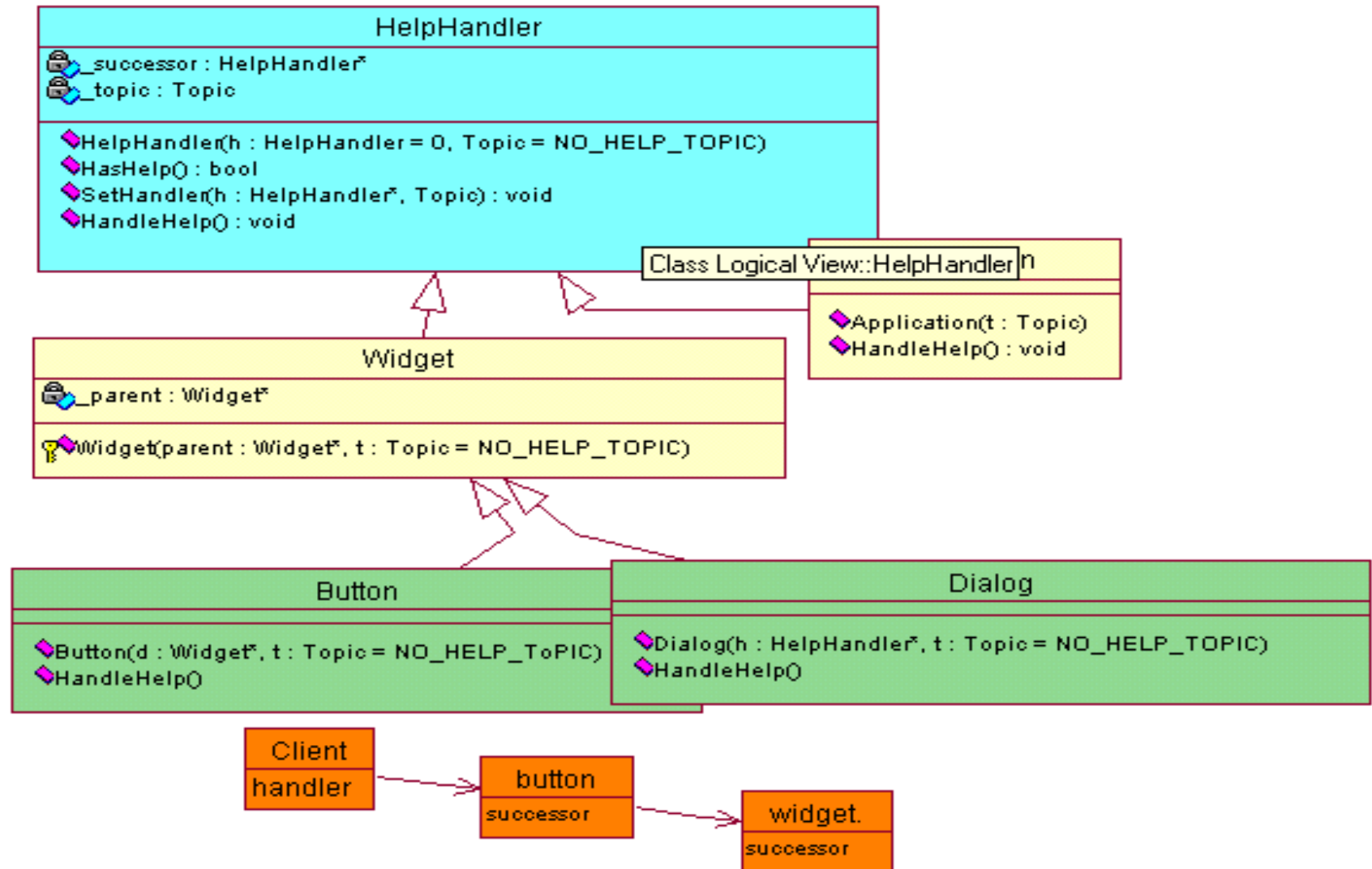
When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

## Consequences

Chain of Responsibilities has the following benefits and liabilities:

- **Reduced coupling**
- **Added flexibility in assigning responsibilities to objects**
- **Receipt is not guaranteed** - since there is no explicit receiver, there's no guarantee it will be handled - the request can fall off the end of the chain without ever being handled.

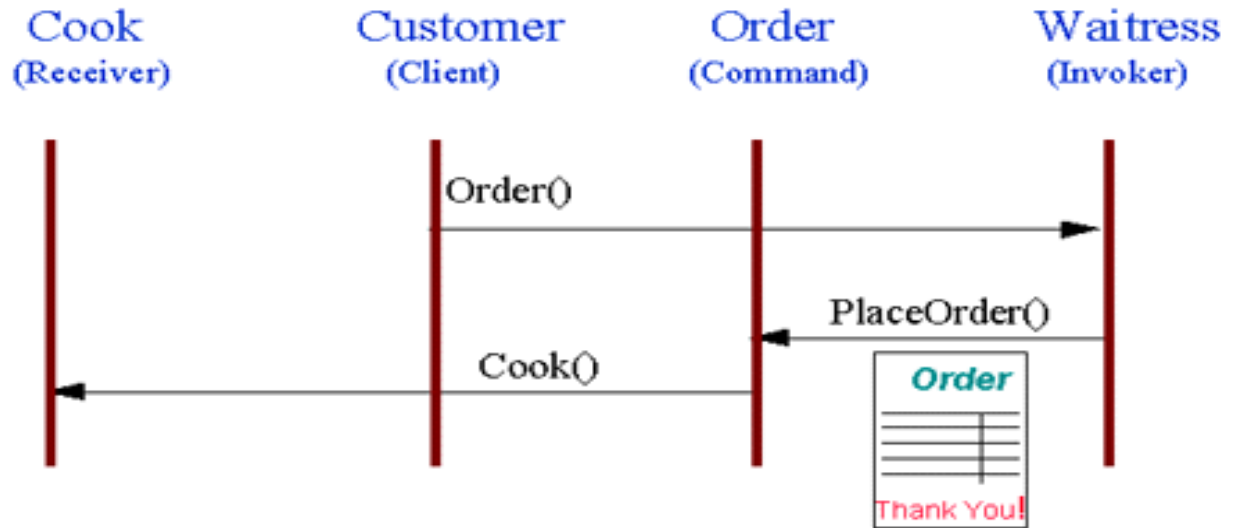
# Model



# Command

# COMMAND

Object Interaction  
Diagram for  
*Command* using  
Diner Example



The *Command* pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests.

The "check" at a diner is an example of a *Command* pattern. The waiter or waitress takes an order, or command from a customer, and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by different diners is not dependent on the menu, and therefore they can support commands to cook many different items.

# Intent

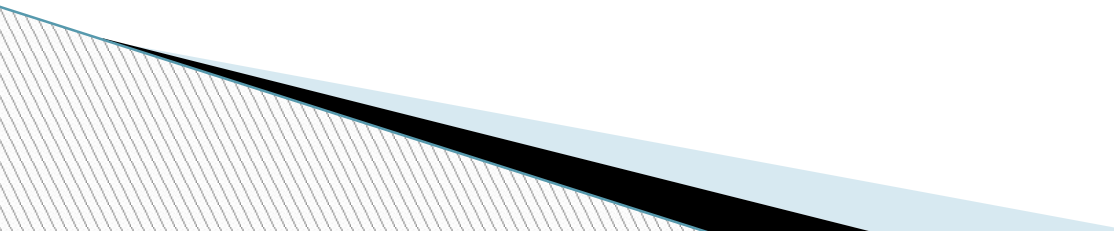
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

## Also Known As

Action, Transaction

## Problem:

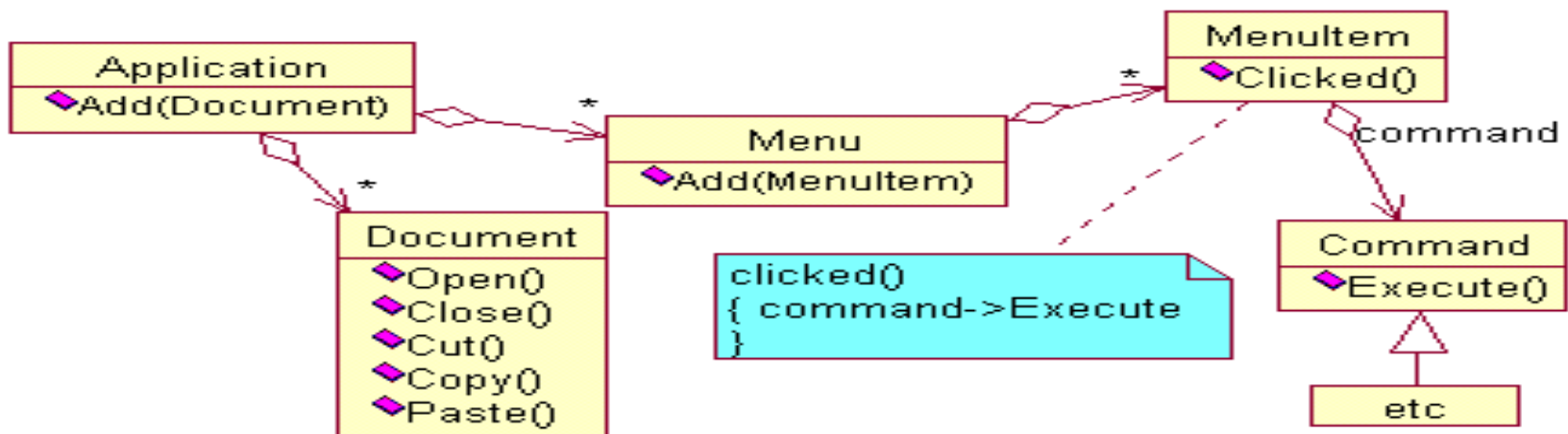
Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.





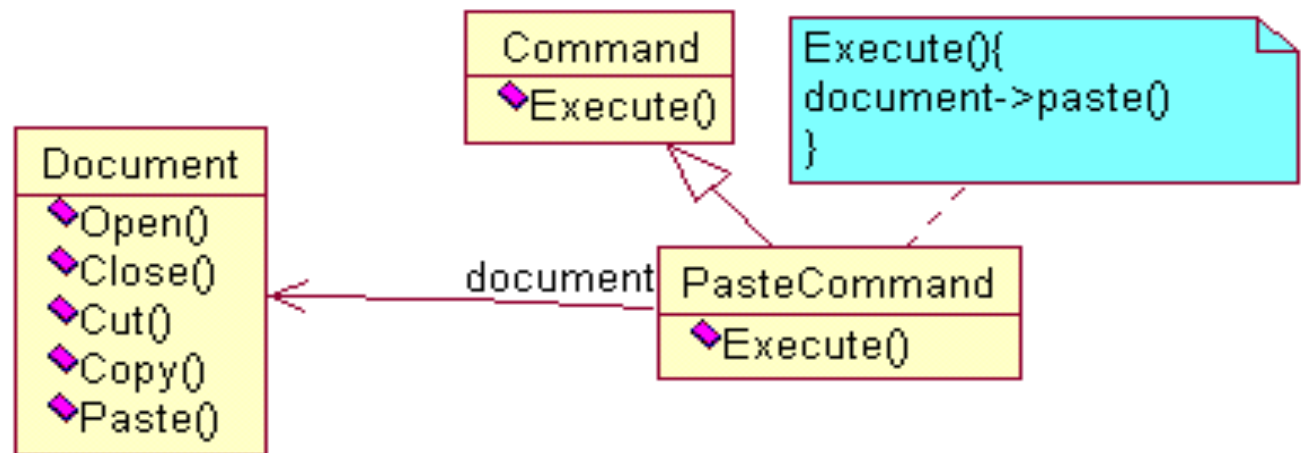
## Illustration: - Menus implemented using command

- ▶ Each choice in a Menu is an instance of a **MenuItem** class. An Application class creates these menus and their menu items along with the rest of the user interface. The Application class also keeps track of Document objects that a user has opened.
- ▶ The application configures each MenuItem with an instance of a concrete Command subclass. When the user selects a MenuItem, the MenuItem calls Execute on its command, and Execute carries out the operation.



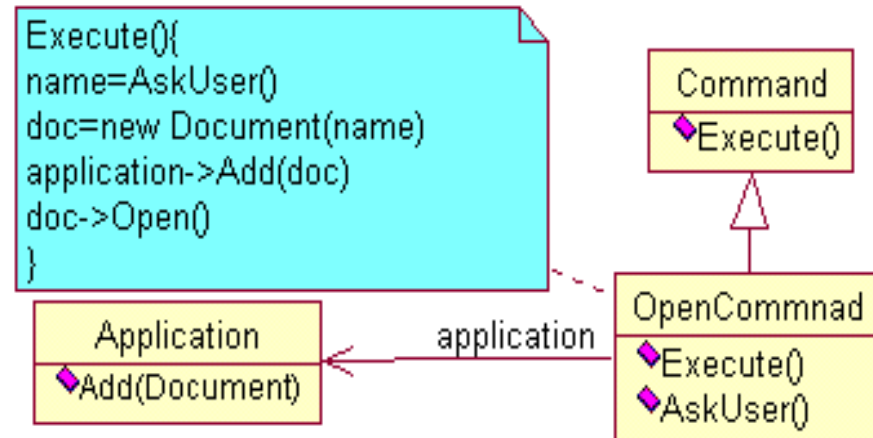
## Illustration: - menus implemented using command

- ▶ MenuItems don't know which subclass of Command they use. Command subclasses store the receiver of the request and invoke one or more operations on the receiver. <<Example next slide>>
- ▶ **For example, PasteCommand** supports pasting text from the clipboard into a Document. PasteCommand's receiver is the Document object it is supplied upon instantiation. The Execute operation invokes Paste on the receiving Document.



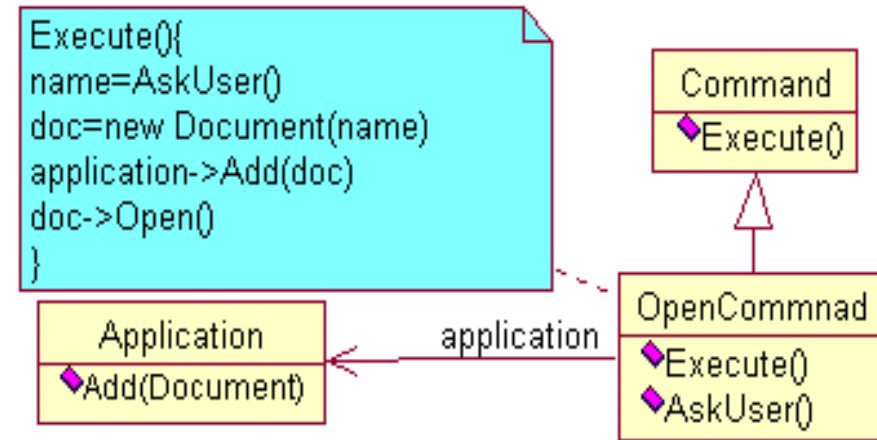
## Illustration: - menus implemented using

**OpenCommand's** Execute operation is different: it prompts the user for a document name, creates a corresponding Document object, adds the document to the receiving application, and opens the document.



In each of these examples, notice how the Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it. This gives us a lot of flexibility in designing our user interface. An application can provide both a menu and a push button interface to a feature just by making the menu and the push button share an instance of the same concrete Command subclass.

# Illustration...

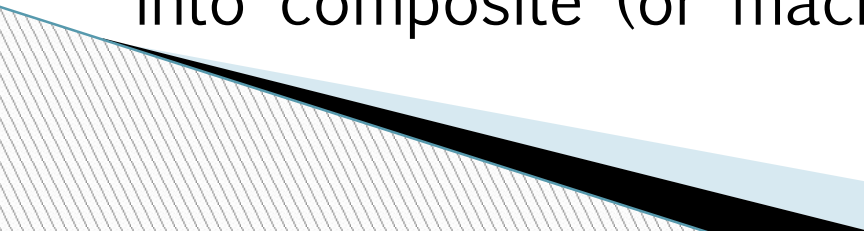


We can replace commands dynamically, which would be useful for implementing context-sensitive menus.

We can also support command scripting by composing commands into larger ones.

All of this is possible because the object that issues a request only needs to know how to issue it; it doesn't need to know how the request will be carried out.

# Solution

- ▶ Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an `execute()` method that simply calls the action on the receiver.
  - ▶ All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual `execute()` method whenever the client requires the object's "service".
  - ▶ Sequences of Command objects can be assembled into composite (or macro) commands.
- 

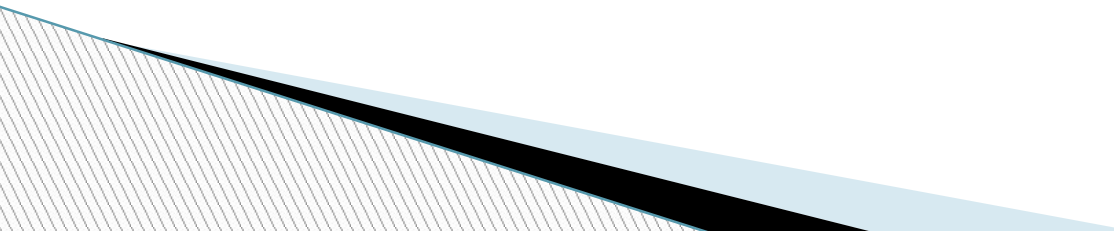
## Applicability

### **Use the Command pattern when you want to**

- Parameterize objects by an action to perform, as MenuItem.

You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point.

Commands are an object Oriented replacement for callbacks.

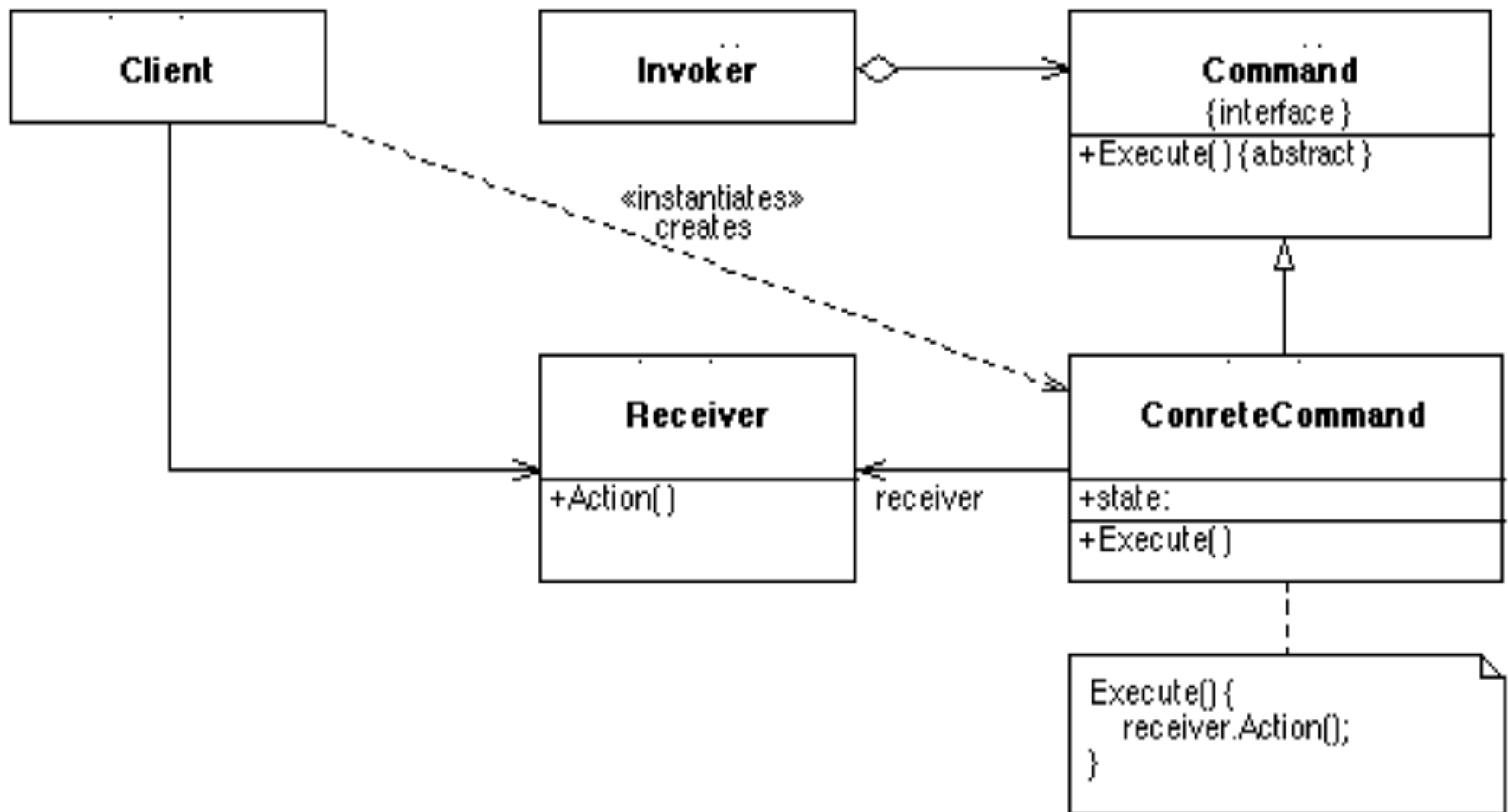
- Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
- 

Applicability.....

## **Use the Command pattern when you want to**

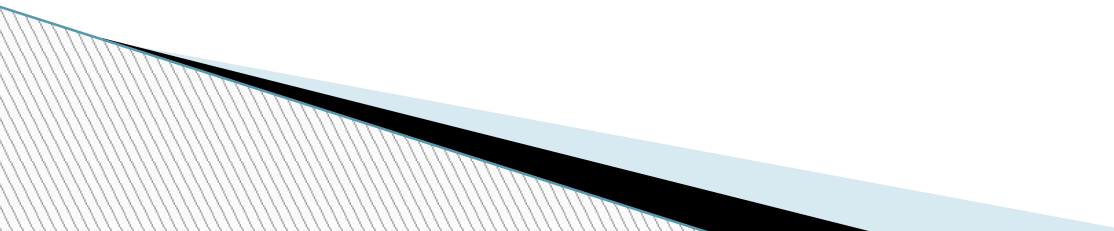
- Support Undo
- Support logging changes so that they can be reapplied in case of a system crash. Recovering from a crash involves reloading logged commands from disk and re-executing them with the execute operation.
- Structure a system around high-level operations built on primitives operations. Such structure is common in information system that support transactions.

# Structure

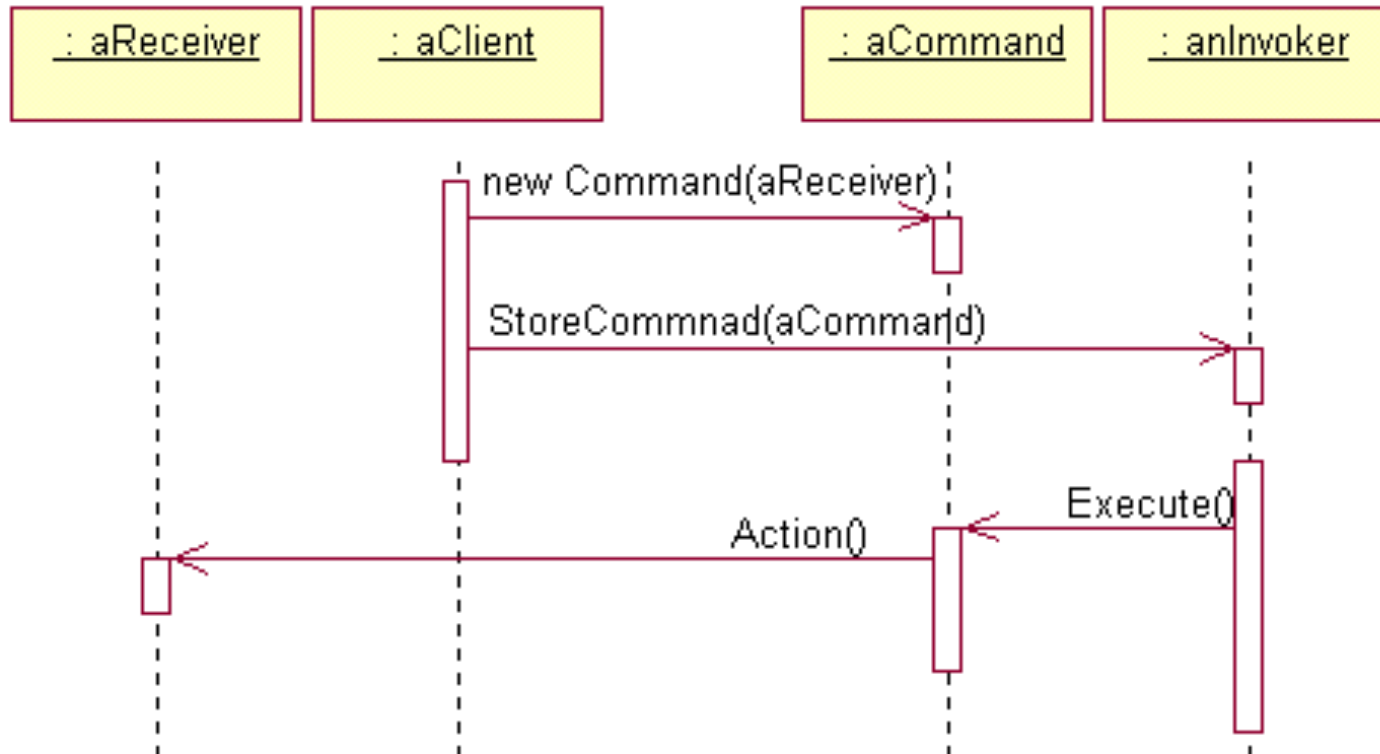




# Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
  - An Invoker object stores the ConcreteCommand object.
  - The invoker issues a request by calling Execute on the command. When commands are undo-able, ConcreteCommand stores state for undoing the command prior to invoking Execute.
  - The ConcreteCommand object invokes operations on its receiver to carry out the request.
- 

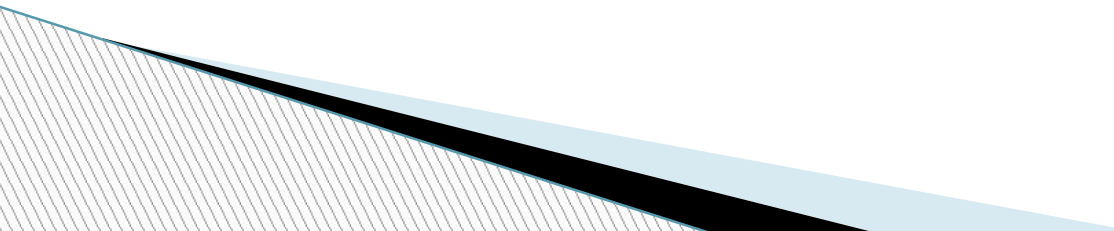
# Collaborations



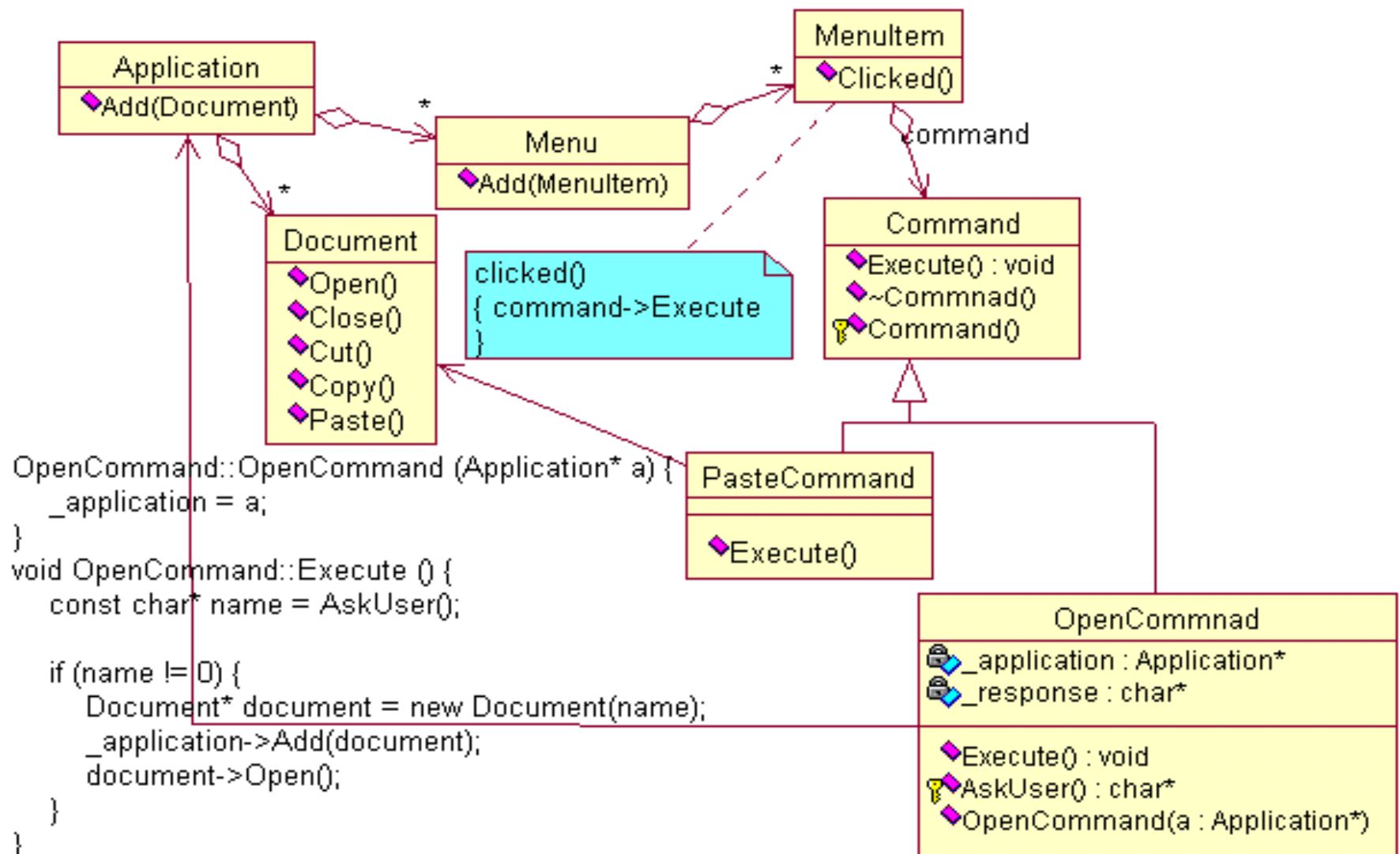
The following diagram shows the interactions between these objects. It illustrates how Command decouples the invoker from the receiver ( and the request it carries out ).

# Consequences

**The Command pattern has the following consequences:**

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

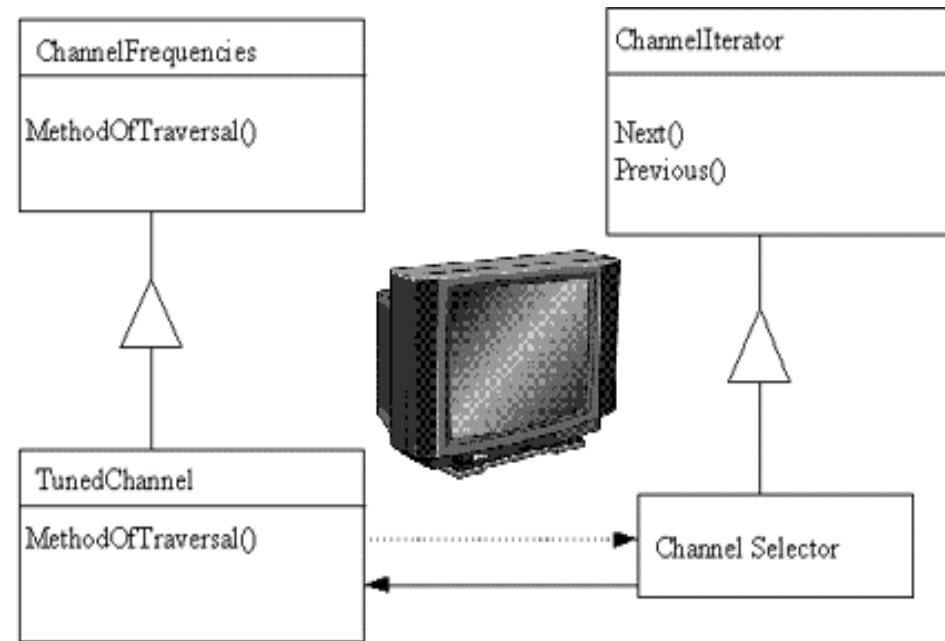
# Model



# Iterator

# Iterator

The *Iterator* provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object.



On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.

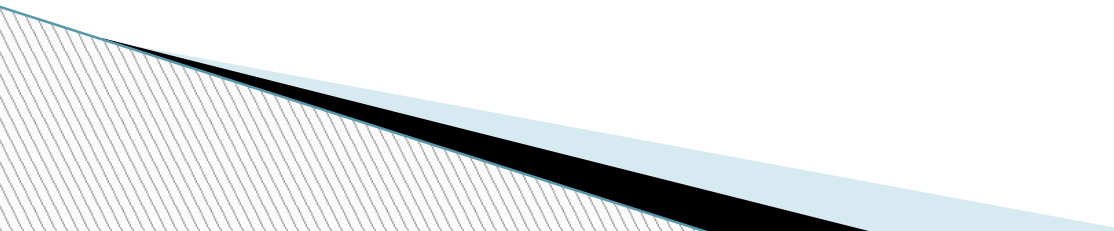
Intent:

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

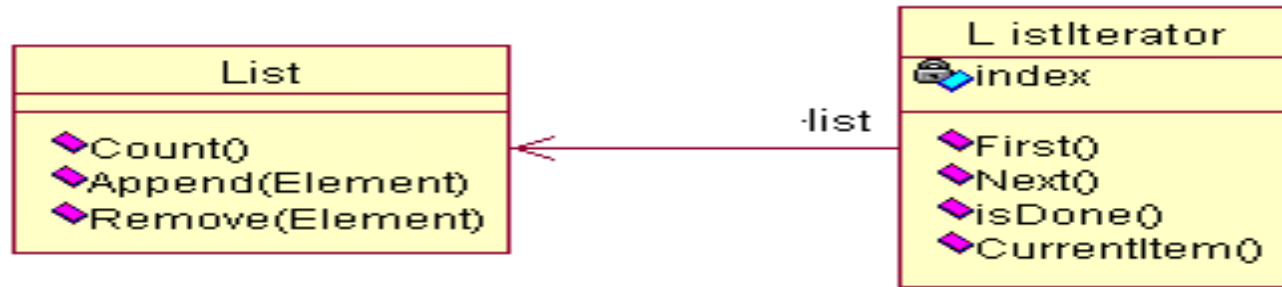
Also Known As:  
cursor

**PROBLEM:**

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.



# Illustration

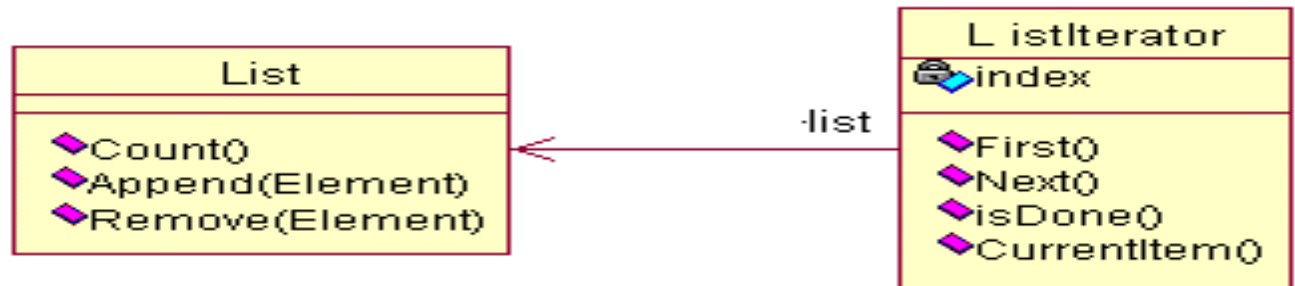


- ▶ For example: a **List** class would call for a **ListIterator** with the following relation-ship between them:
- ▶ "An aggregate object such as a list should give you a way to access its elements without exposing its internal structure".
- ▶ Moreover, you might want to traverse the list in **different ways**, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will require.

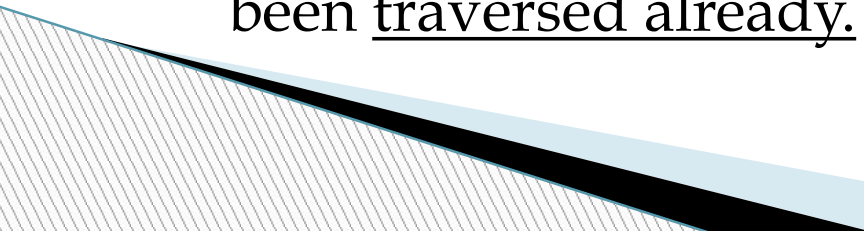


## Illustration.....

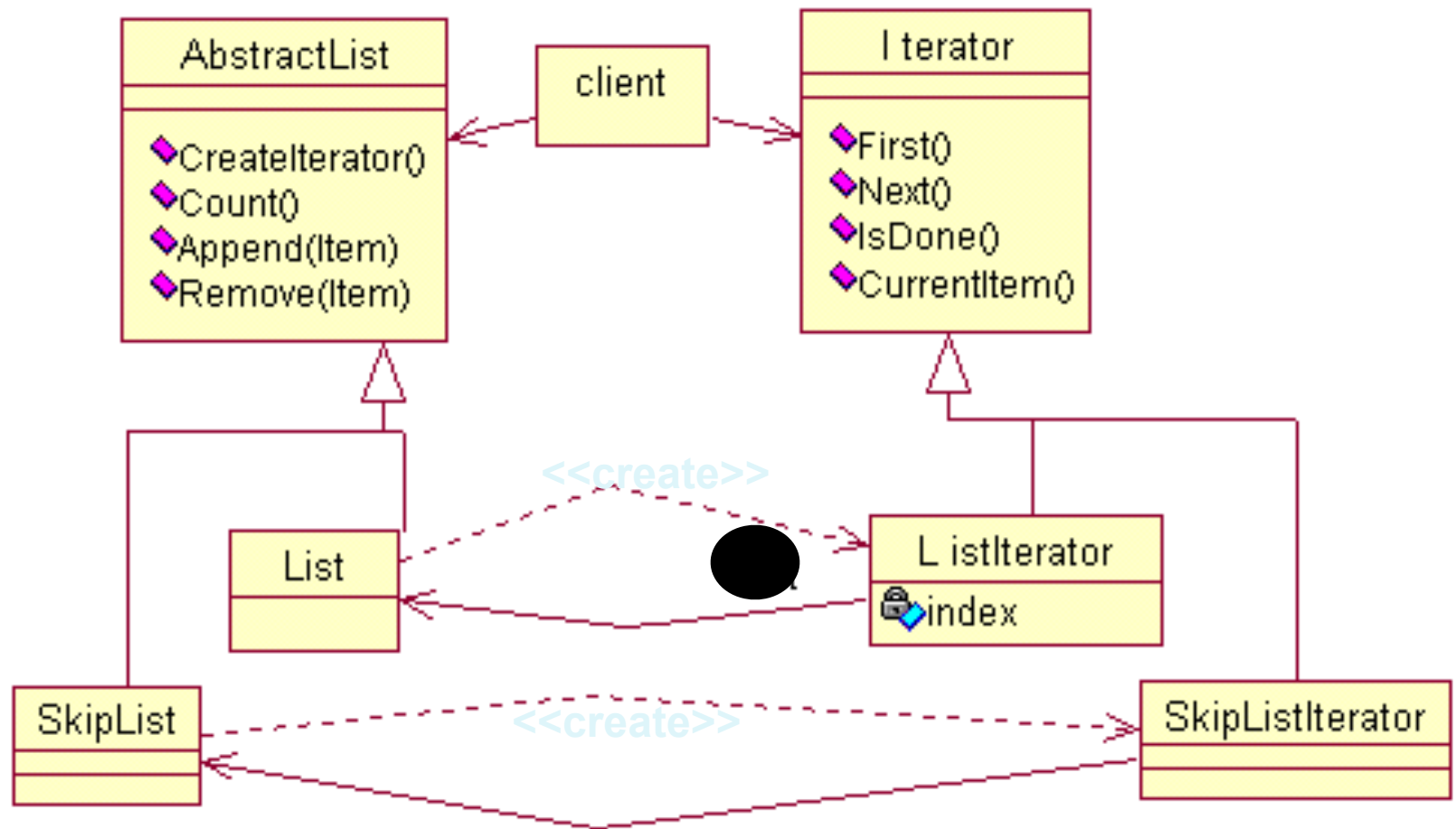
- ▶ You might also need to have more than one traversal depending on the same list." And,
- ▶ providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.



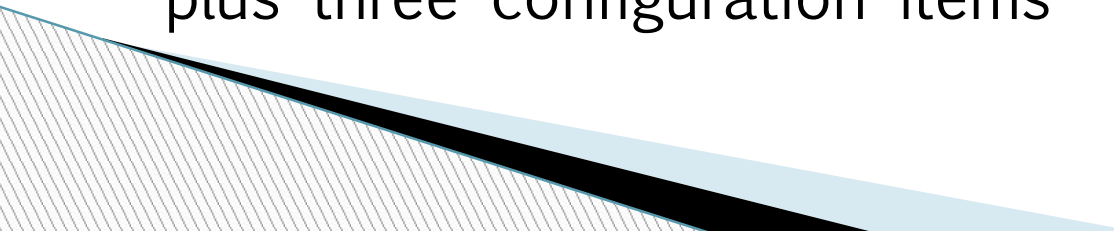
## Solution

- ▶ The Iterator pattern lets you do all this. The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an **Iterator** object that defines a standard traversal protocol.
  - ▶ 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.
- 

## Illustration : Solution

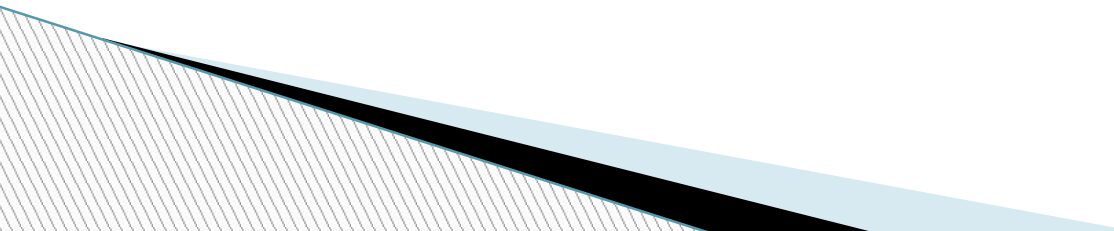


## Discussion

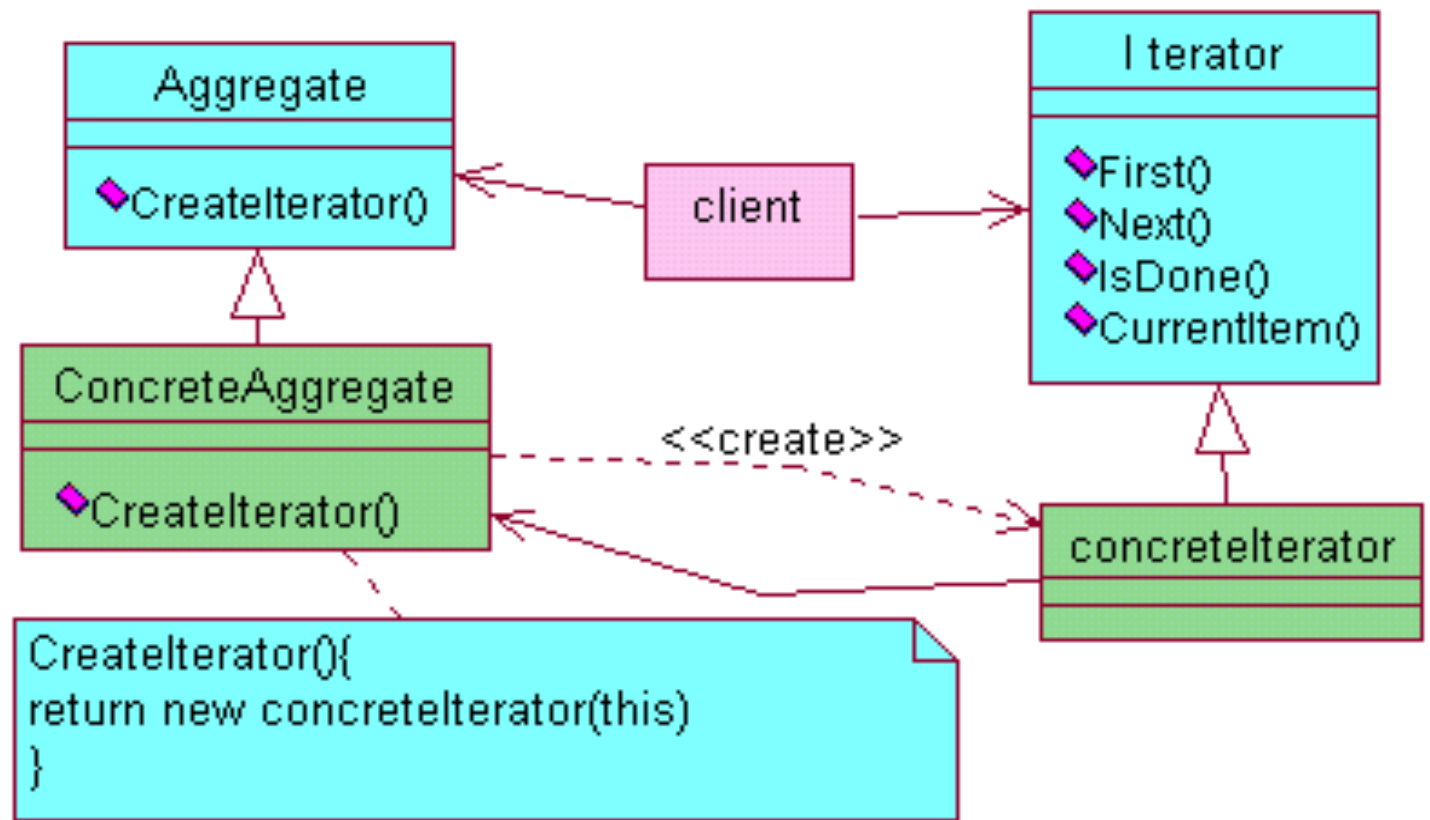
- ▶ The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.
  - ▶ As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items
- 

## 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



## Collaborations:

- A **ConcreteIterator** keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

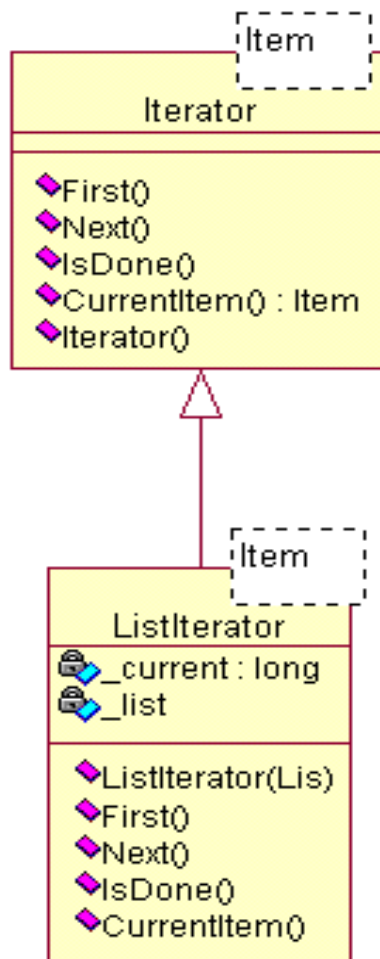
# Consequences

**The Iterator pattern has three important consequences:**

- ▶ **It supports variations in the traversal of an aggregate.** Complex aggregates may be traversed in many ways. For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder. Iterators make it easy to change the traversal
- ▶ **Iterafors simplify the Aggregate interface.** Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
- ▶ **More than one traversal can be pending on an aggregate:** *An* Iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

# Model

## Iterator



## Using the Iterators:

```
void Print Employees ( Iterator<Employee*>&i){
    for(i.First(); !i.IsDone (); i.Next ()) {
        i.CurrentItem() -> Print(); }
}
```

---

```
template < class Item >
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

First position the iterator to the first element:

```
template<class Item>
void ListIterator<Item>::First() {
    _current = 0;
}
```

Next advances the current element:

```
template <class Item>
void ListIterator<Item>::Next (){
    _current++;
}
```

IsDone checks whether the index refers to an element with in the list:

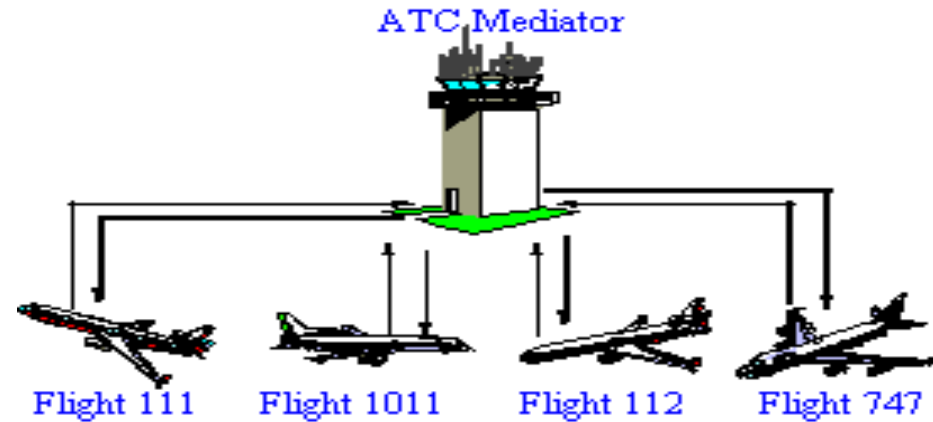
```
template <class Item>
bool ListIterator<Item>::IsDone() const{
    return _current >= _list->Count();
}
```



# Mediator

# MEDIATOR

ATC Tower Example  
of *Mediator*



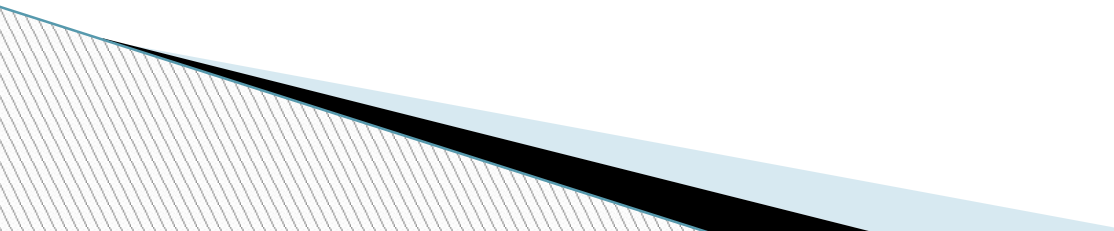
The *Mediator* defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the *Mediator*, rather than with each other.

The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

## Intent

- ▶ Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

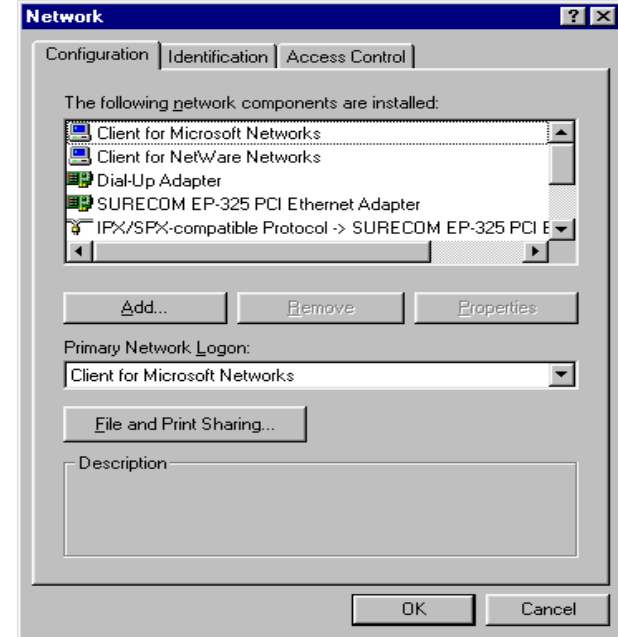
## Problem

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

# Illustration: Problem

Consider the implementation of dialog boxes in a GUI.

A dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields, as shown here.



Often there are dependencies between the widgets in the dialog. For example a button gets disabled or enabled when a certain entry in a list of choices of list box is selected .

Different dialog boxes will have different dependencies between widgets. So even though dialogs display the same kinds of widgets, they can't simply reuse stock widget classes; they have to be customized to reflect dialog-specific dependencies. Customizing them individually by subclassing will be tedious, since many classes are involved.

**Illustration:** We can avoid previous said problem by encapsulating collective behavior in a separate mediator object. .

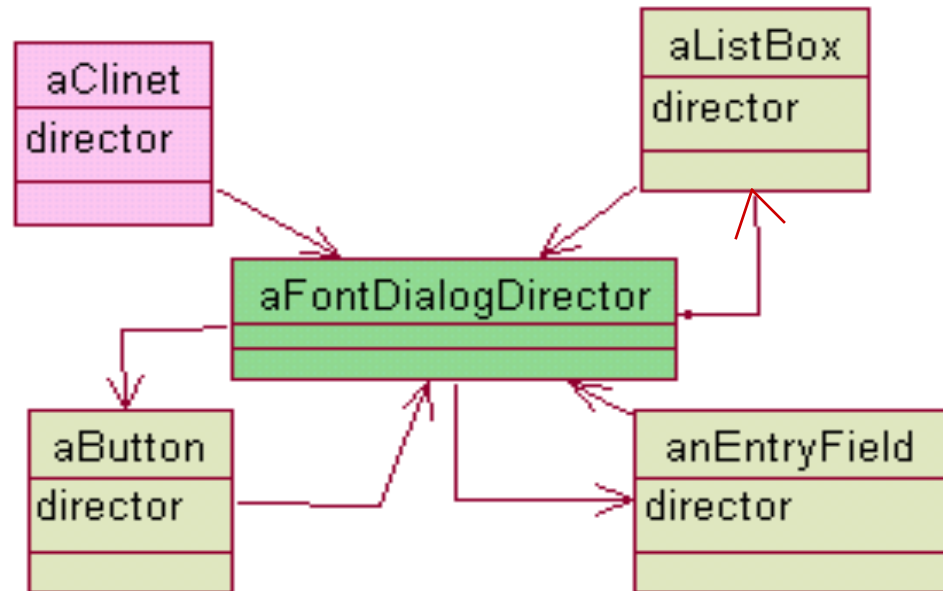
## Solution

A mediator is responsible for controlling and coordinating the iterations 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 interconnection.

FontDialogDirector can be mediator between the widgets in a dialog box.

A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction.

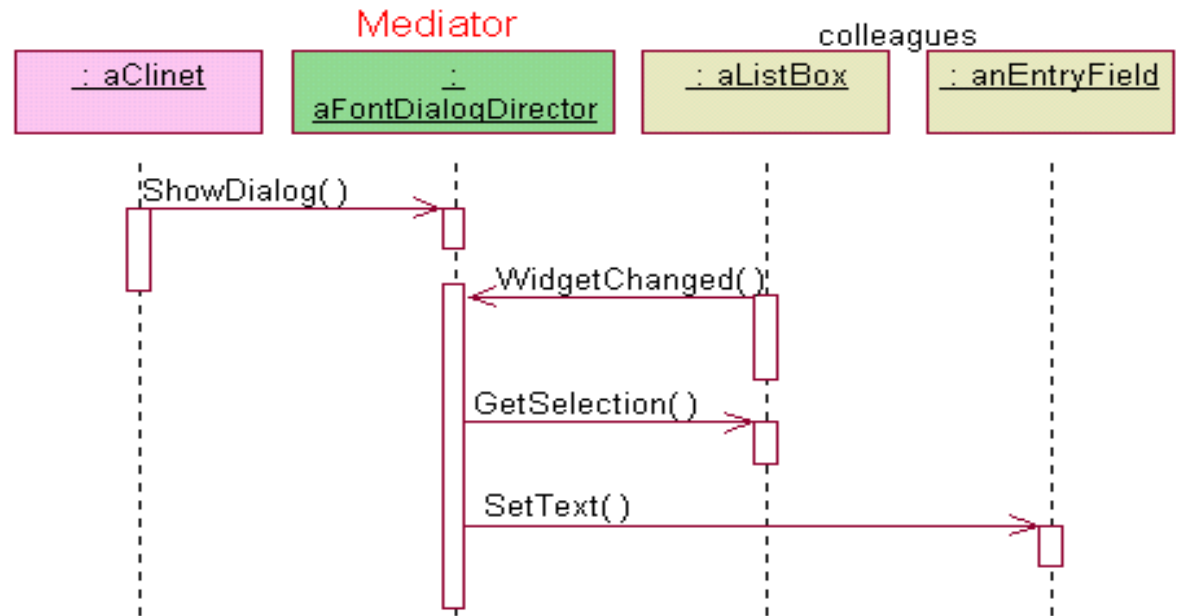
It acts as a hub of communication for widgets.



# Illustration:

## Solution

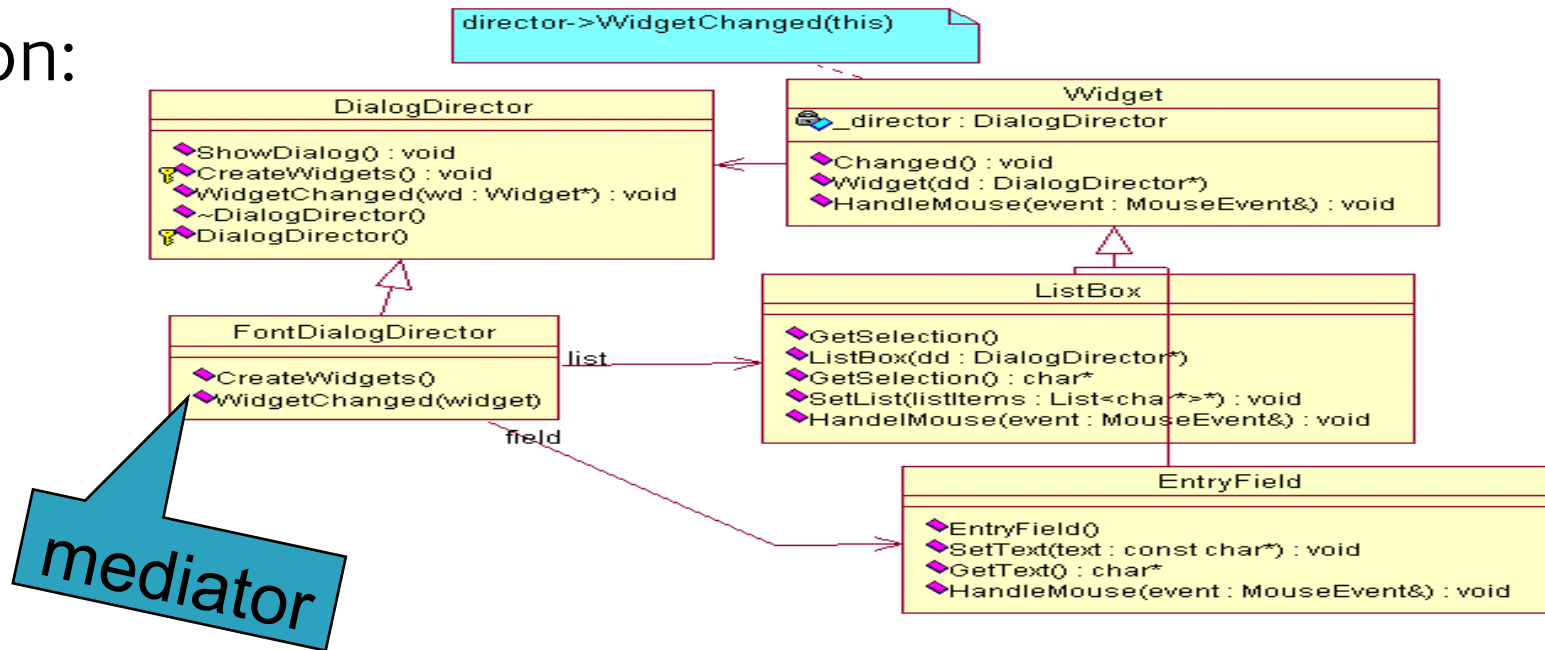
Here's the succession of events by which a list box's selection passes to an entry field:



The following interaction diagram illustrates how the objects cooperate to handle a change in a list box's selection.

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 initializing an action ( e.g. 'demibold','oblique')

# Illustration: Solution

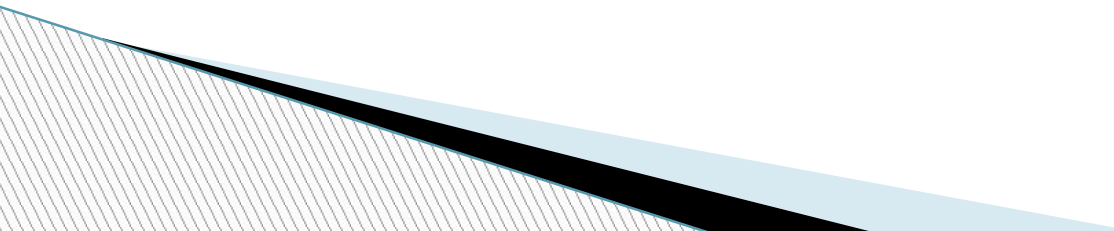


In the above model notice who the director communicates with list box and entry field. Widgets communicate with each other only indirectly, through the director. All they know is the director and they in turn don't know each other.

Furthermore, because the behavior is localized in one class, it can be changed or replaced by extending or replacing that class.

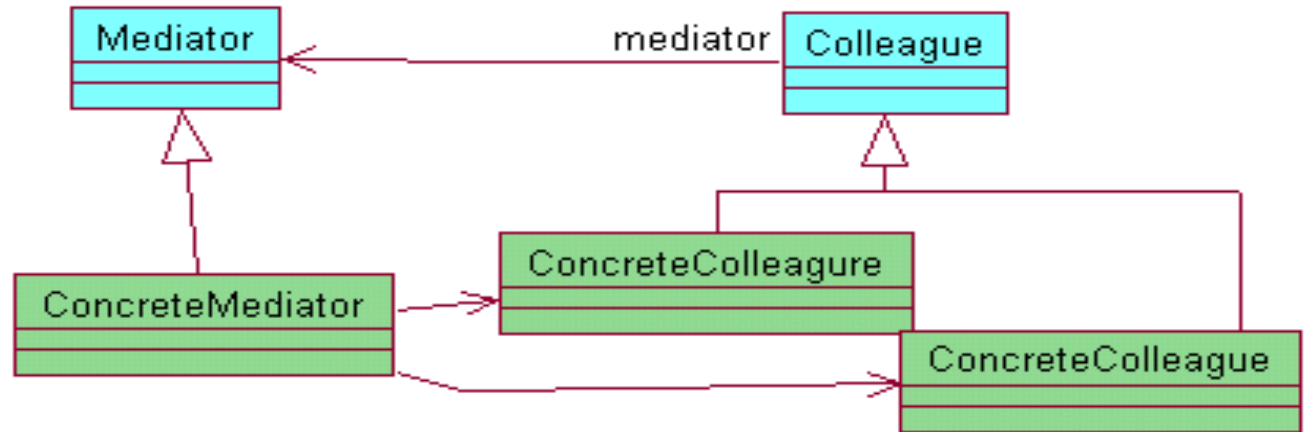
# Applicability

## **Use the Mediator pattern when:**

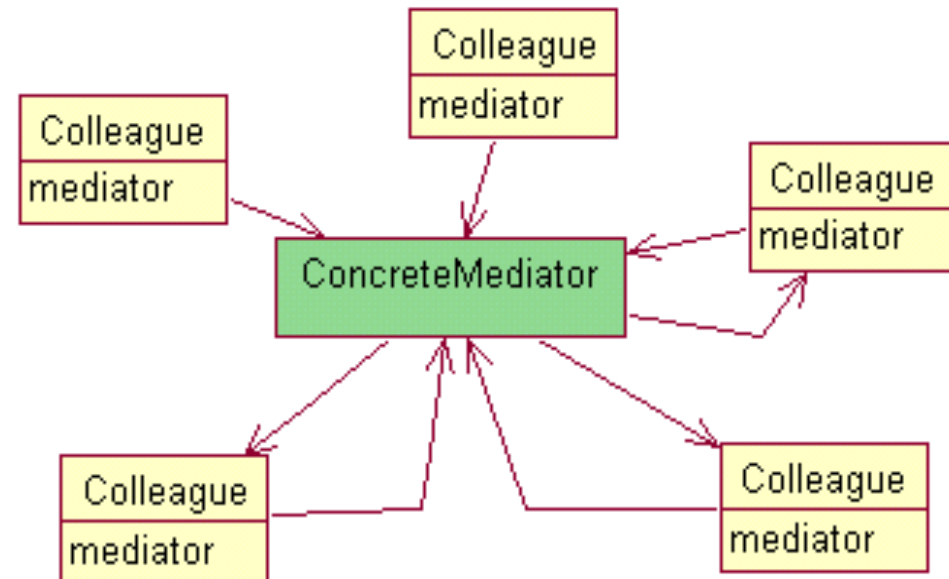
- 1> a set of objects communicate in well-defined but complex ways . The resulting interdependencies are unstructured and difficult to understand.
  - 2> reusing an object is difficult because it refers to and communicates with many other objects.
  - 3> a behavior that's distributed between several classes should be customizable without a lot of subclassing.
- 



# Structure

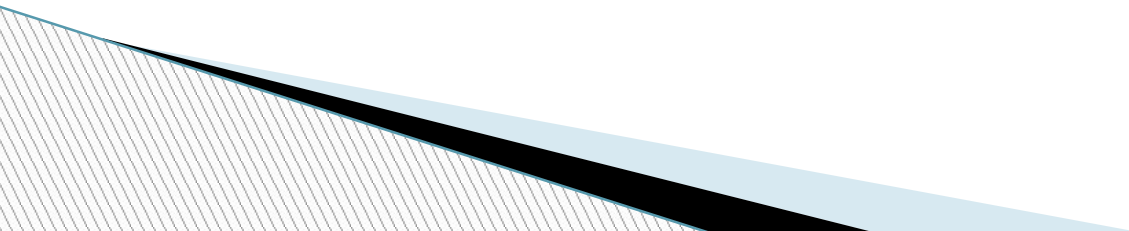


A typical object structure might look like this:

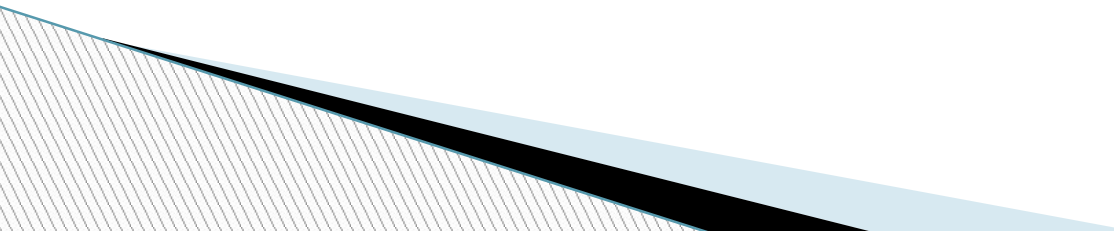


## Collaborations

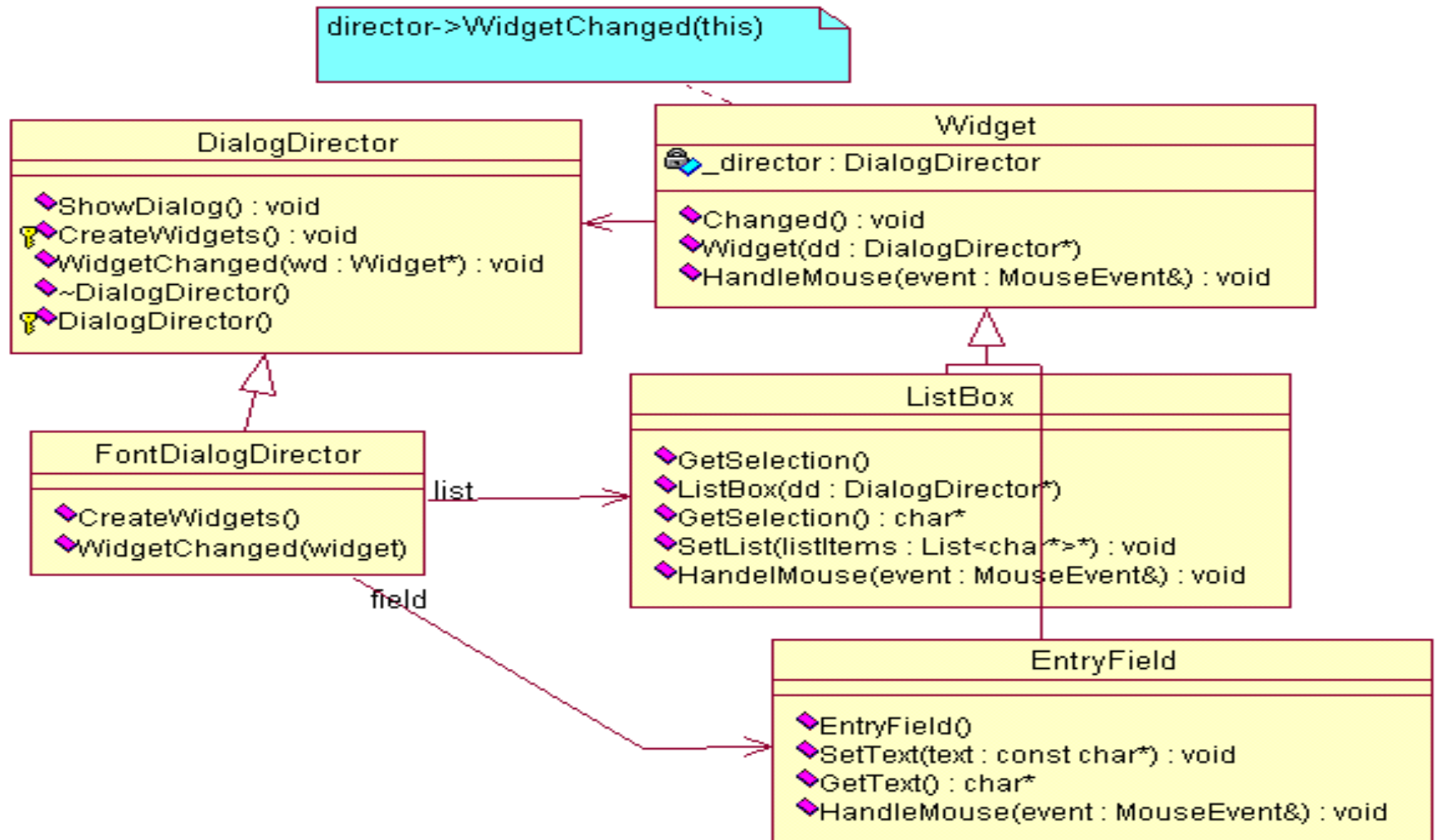
Colleagues send and receive requests form a Mediator object. The mediator implements the cooperative behavior by routing between the appropriate colleague(s).



## Consequences

- 1> **It limits subclassing:** mediator localizes behavior that otherwise would be distributed among various objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as it is.
  - 2> **It decouples colleagues:**
  - 3> **It simplifies object protocols:** A one to many relationships are easier to understand, maintain, and extend. Thus mediator replaces many to many interactions with one-to-many interactions between the mediator and its colleagues.
  - 4> **It abstracts how objects cooperate:**
  - 5> **It centralizes control:**
- 

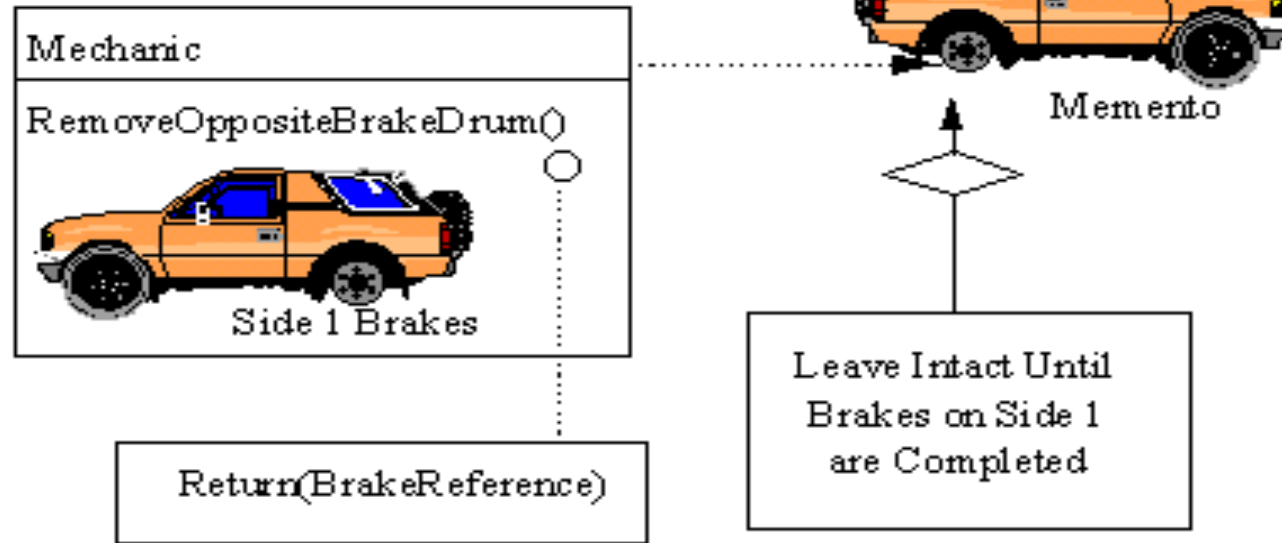
# Model:



# Memento

# MEMENTO

Object Diagram for  
*Memento* using Brake  
Example



The *Memento* captures and externalizes an object's internal state, so the object can be restored to that state later.

This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled, and the other side serves as a *Memento* of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the *Memento*.

# Intent

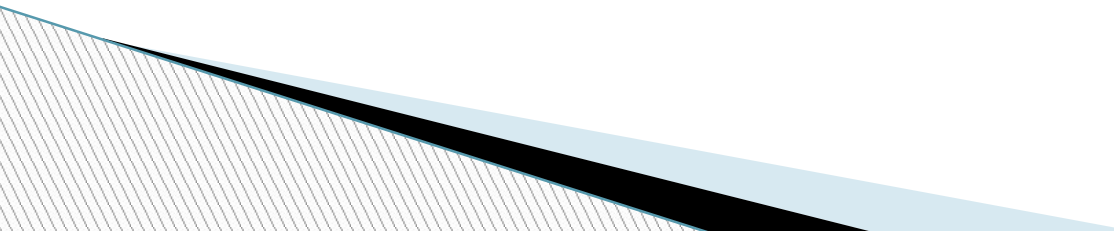
Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.

**Also Known As**

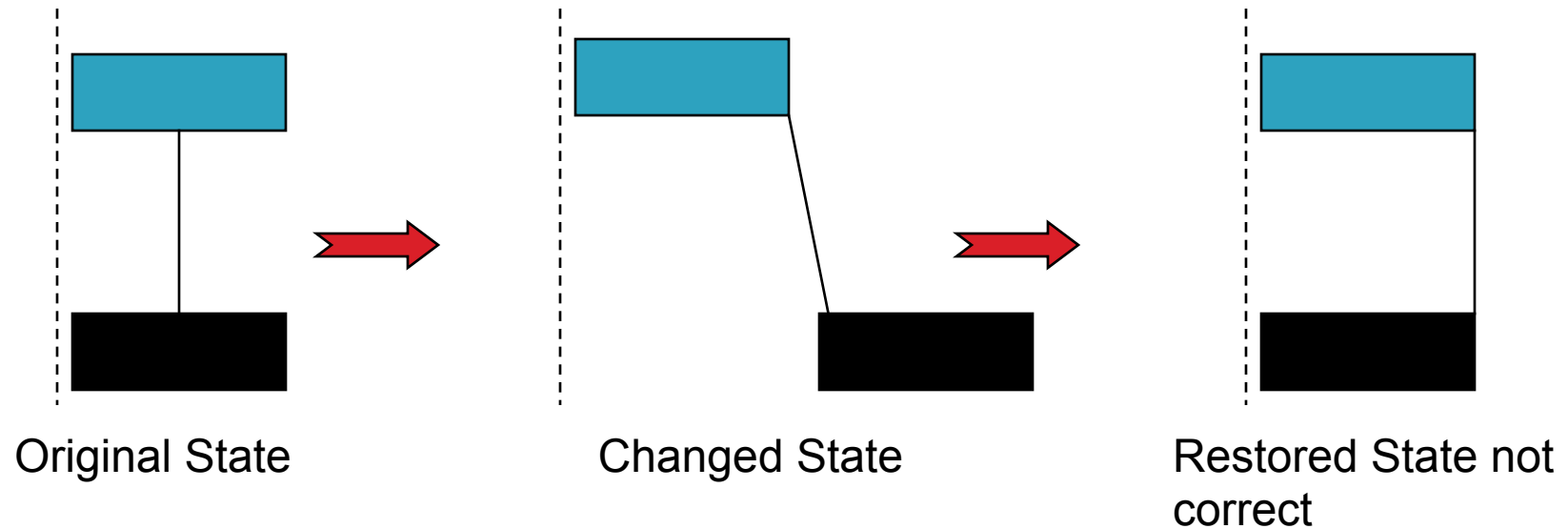
Token

## Problem:

Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations). But object normally encapsulate some or all of their states, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.



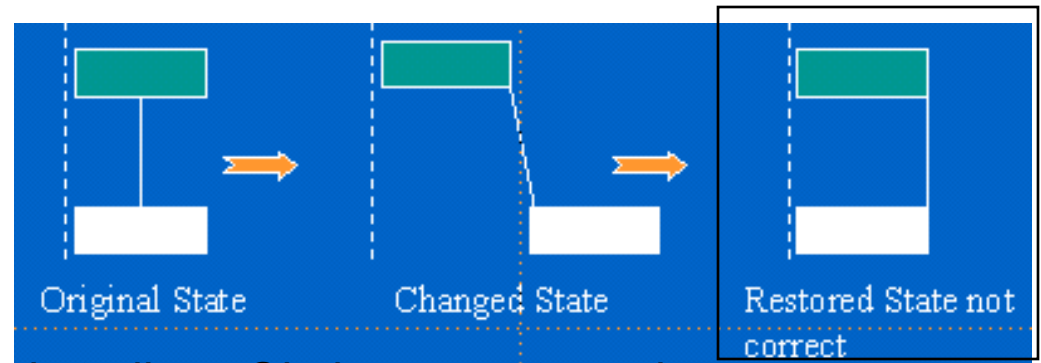
# Illustration: Graphical Editor



Here **Constraint-Solver (C-S)** object will encapsulate the functionality to maintain connectivity relationships between the objects. C-S will record connections as they are made and generates mathematical equations that describe them, and in future uses these equation to calculate results to rearrange the graphics accordingly when ever the user makes a modification to diagram.



# Illustration: Problem



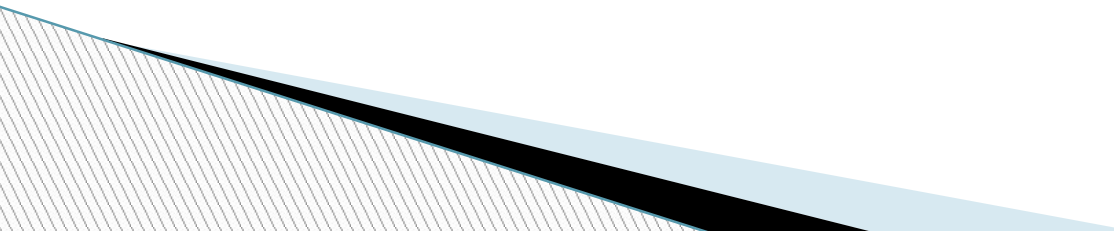
Supporting undo in this application is tedious. Obvious way to undo a move operation is to store original distance moved and move the object back an equivalent distance. But this does not guarantee all objects will appear where they did before

In general, the ConstraintSolver's public interface might be insufficient to allow precise reversal of its effects on other objects.

The undo mechanism must work more closely with ConstraintSolver to reestablish previous state, but we should also avoid exposing the ConstraintSolver's internals to the undo mechanism.

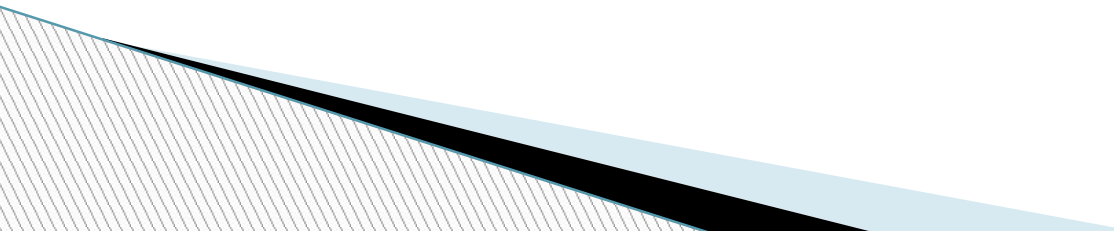
## Illustration: Solution

Solve this problem using **Memento** ( “ an object that stores a snapshot of the internal state of another object - the memento’s **Originator**).

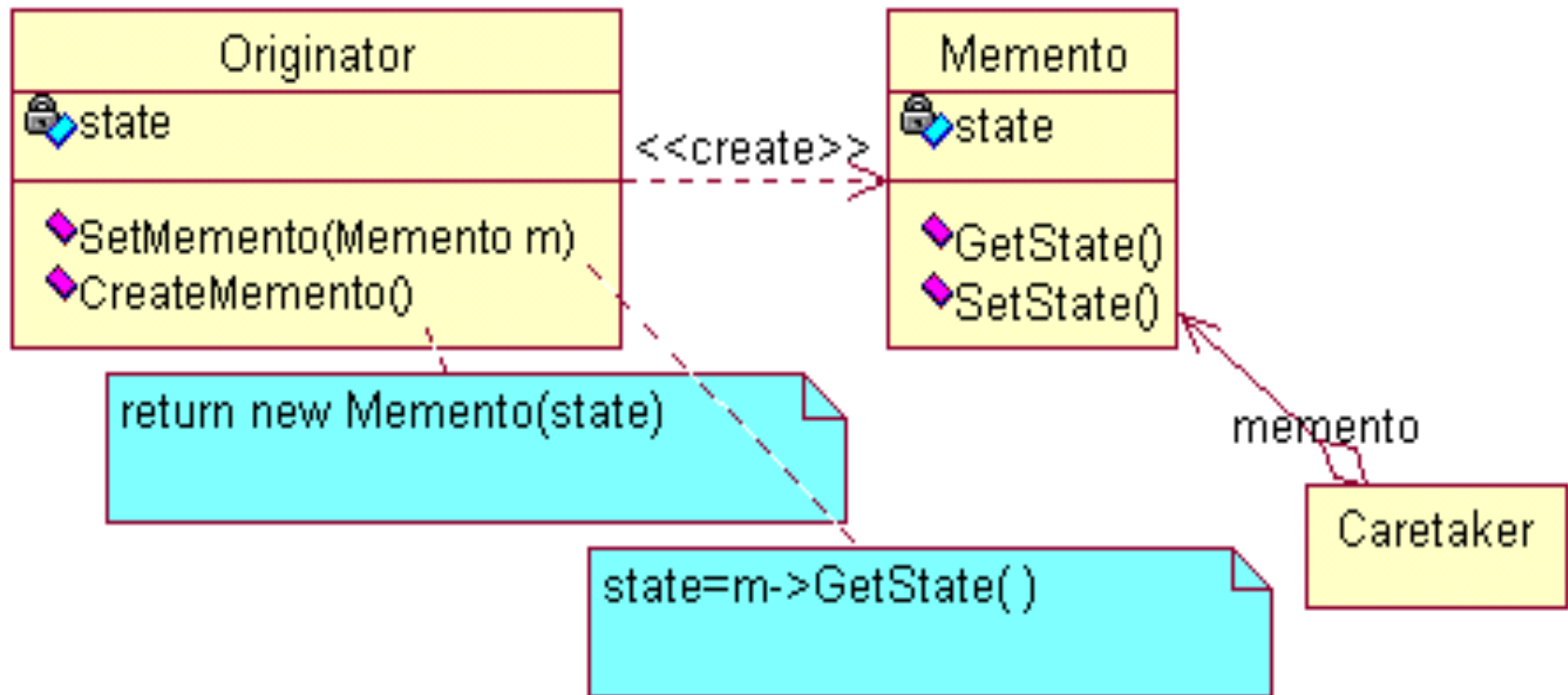
1. The editor request a memento from the ConstraintSolver (**ORIGINATOR**) as a side-effect of the move operation.
  2. The C-S creates and returns a memento, an instance of a class SolverState in this case. A SolverState memento contains data structures that describe the current state of the ConstraintSolver’s internal equations and variables.
  3. Later when the user undoes the move operation, the editor gives the SolverState back to the C-S.
  4. Based on the information in the SolverState, the C-S changes its internal structures to return its equations and variables to their exact previous state.
- 

## Applicability

### **Use the Memento pattern when:**

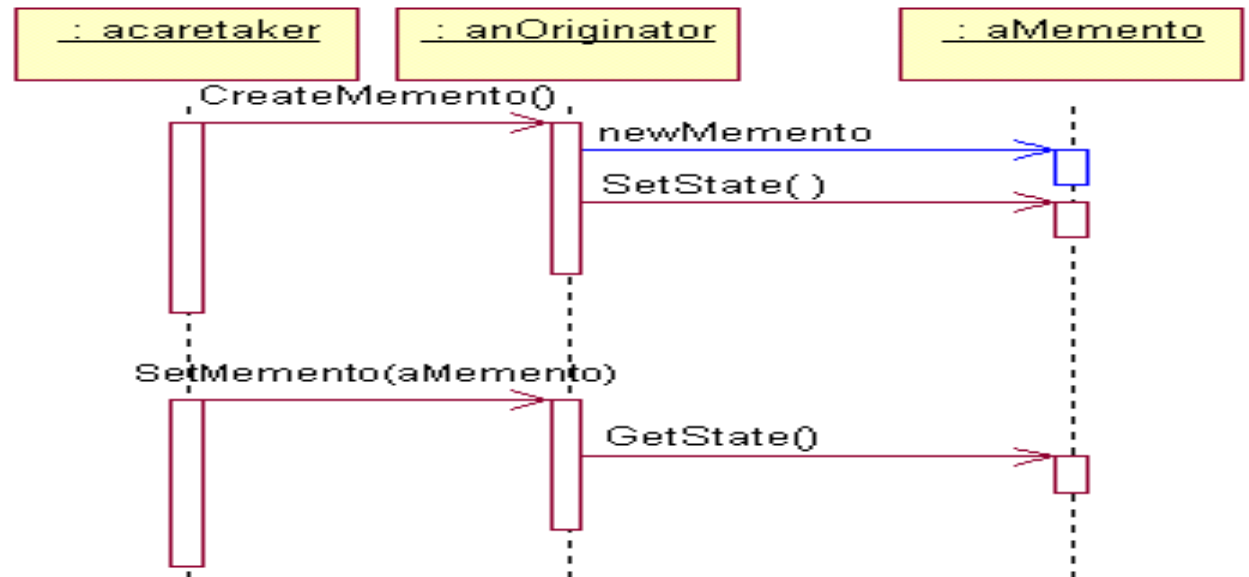
- 1> a snapshot of (some portion of ) an object's state must be saved so that it can be restored to that state later
  - 2> a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.
- 

# Structure



**Caretaker** responsible for memento safekeeping  
never operates on or examines the contents of a memento

# Collaborations

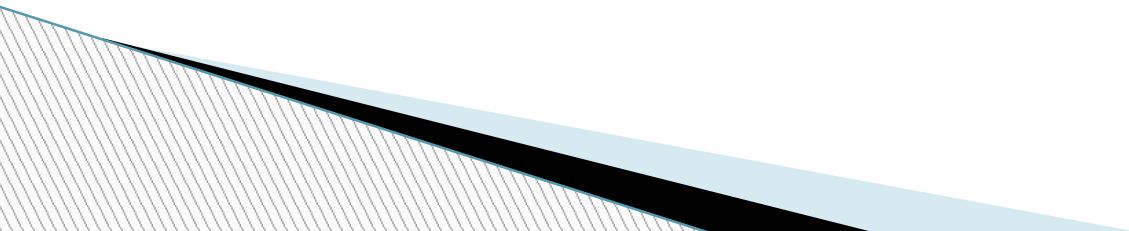


A caretaker request a memento form a originator, holds it for a time, and passes it back to the originator, as the following interaction diagram illustrates.

Memento are passive. Only the originator that created a memento will assign or retrieve its state.

## Consequences:

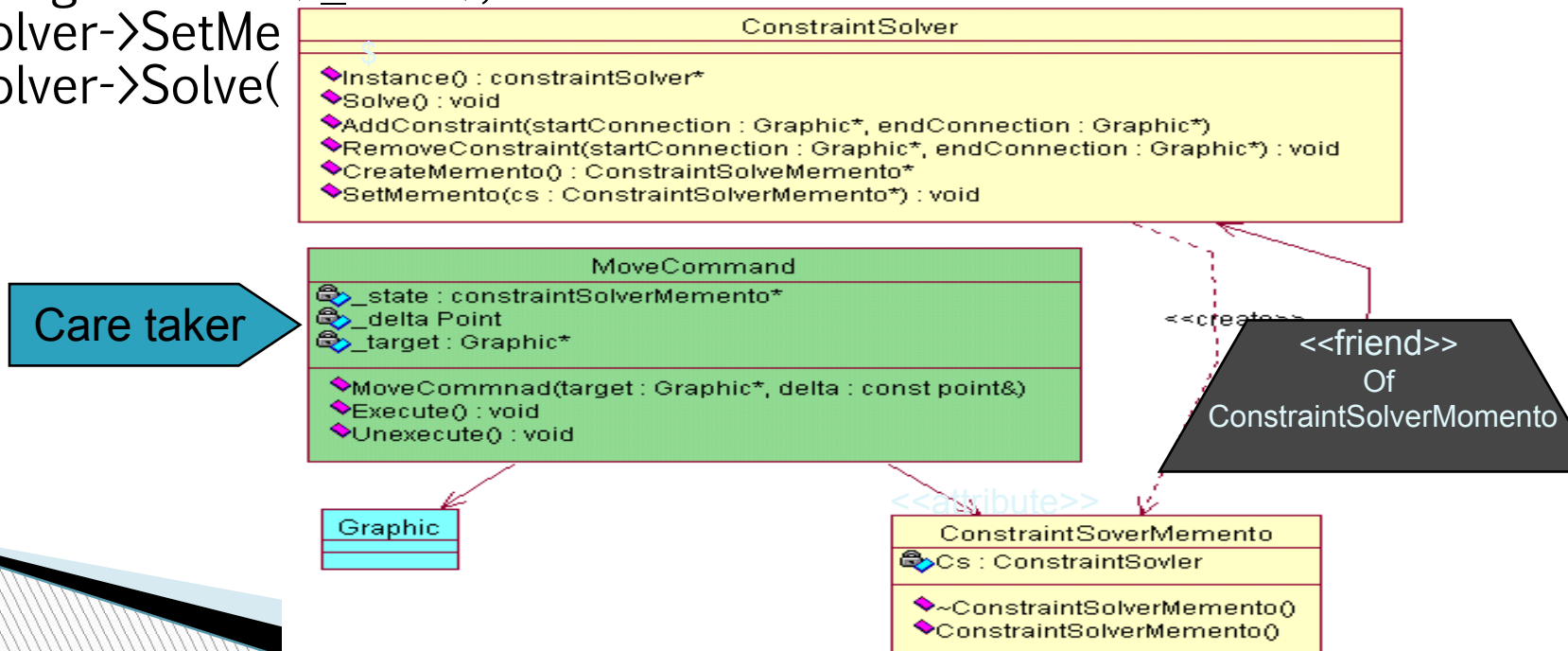
- 1> preserving encapsulating boundary
- 2> It simplifies originator
- 3> using mementos might be expensive
- 4> Defining narrow and wide interfaces
- 5> Hidden costs in caring for mementos



# Model

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // create a memento
    _target->Move(_delta);
    solver->Solve();
}
```

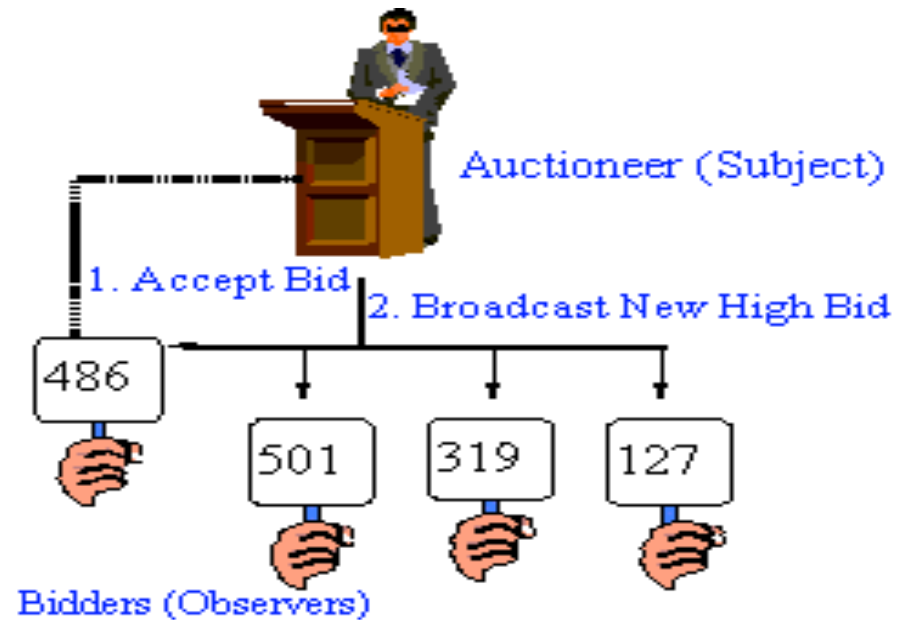
```
void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMe
    solver->Solve(
}
```



# Observer



# OBSERVER



The *Observer* defines a one to many relationship, so that when one object changes state, the others are notified and updated automatically.

Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid

# Intent

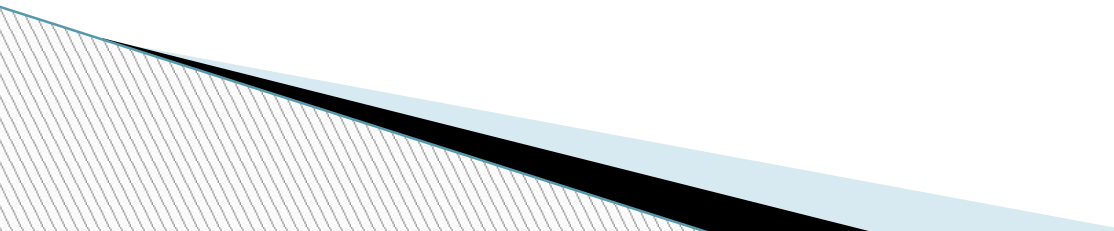
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

## **Alternative Name:**

**Dependents, Publish-Subscribe**

## **Problem:**

How to handle a common side-effect (i.e. need to maintain consistency between related object) of partitioning a system into a collection of cooperating classes without making the classes tightly coupled, - which in turn makes classes less reusable.



# Illustration

Microsoft Excel - Solvsamp

Example 3: Personnel scheduling for an Amusement Park.

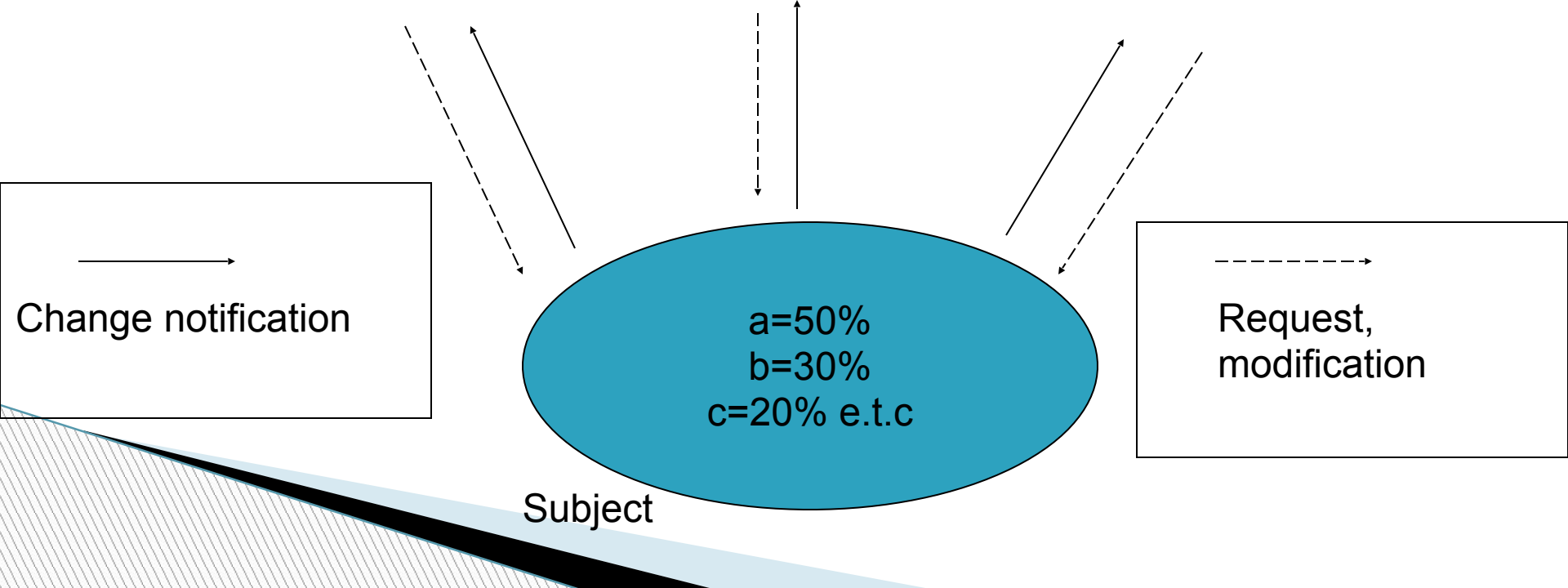
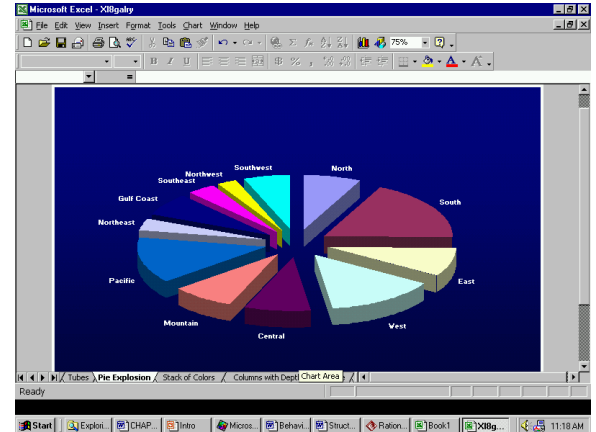
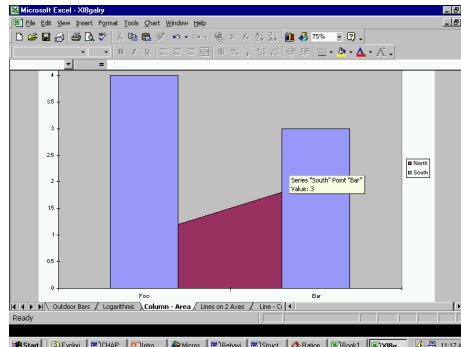
For employees working five consecutive days with two days off, find the schedule that meets demand from attendance levels while minimizing payroll costs.

Sch.	Days off	Employees	Sun	Mon	Tue	Wed	Thu	Fri	Sa
A	Sunday, Monday	4	0	0	1	1	1	1	1
B	Monday, Tuesday	4	1	0	0	1	1	1	1
C	Tuesday, Wed	4	1	1	0	0	1	1	1
D	Wed., Thursday	6	1	1	1	0	0	1	1
E	Thursday, Friday	6	1	1	1	1	0	0	1
F	Friday, Saturday	4	1	1	1	1	1	0	1
G	Saturday, Sunday	4	0	1	1	1	1	1	0

Schedule Totals: 32      24   24   24   22   20   22   28

Total Demand:      22   17   13   14   15   18   24

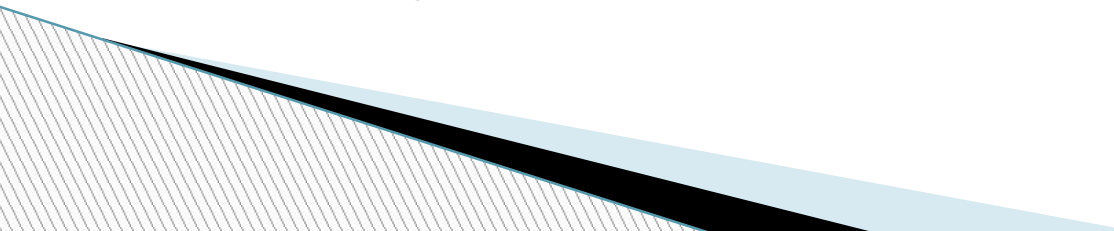
Payroll: \$40



## Solution:

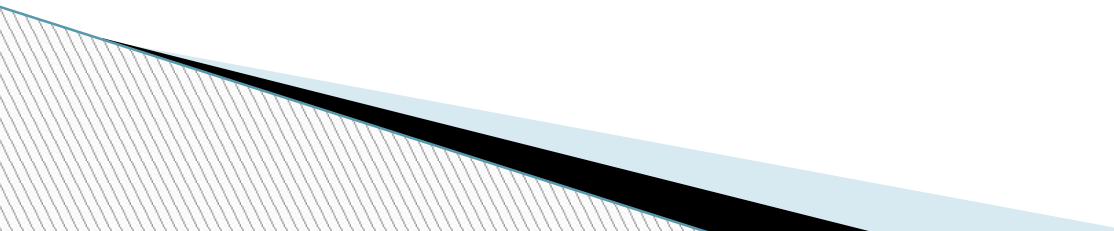
The observer pattern describes how to establish these relationships.

The key objects in this pattern are subject and observer.

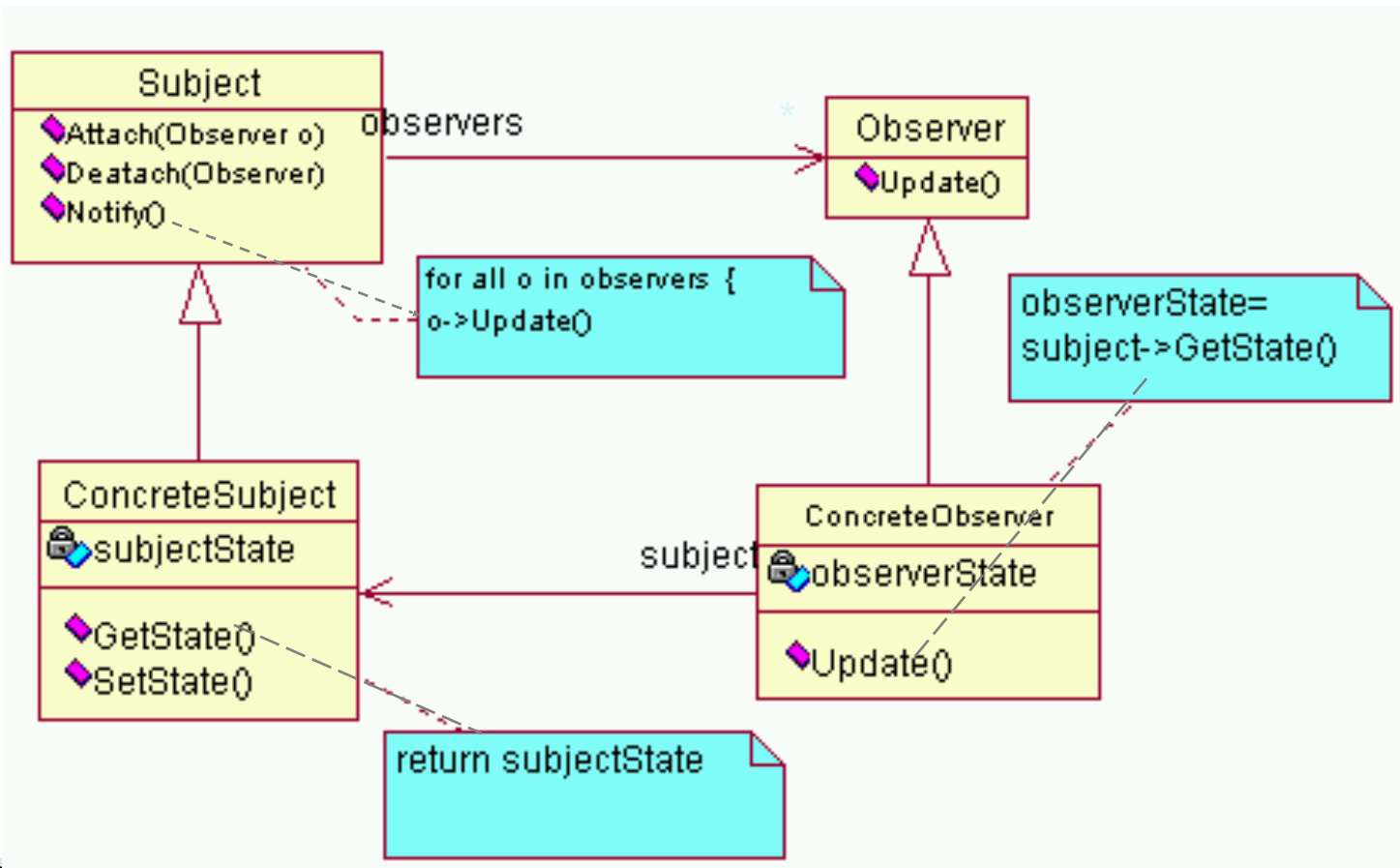
- A subject may have any number of dependent observer.
  - All observers are notified whenever the subject undergoes a change in state.
  - In response, each observer will query the subject to synchronize its state with the subject's state.
- 

# 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 tightly coupled.
- 


# Structure



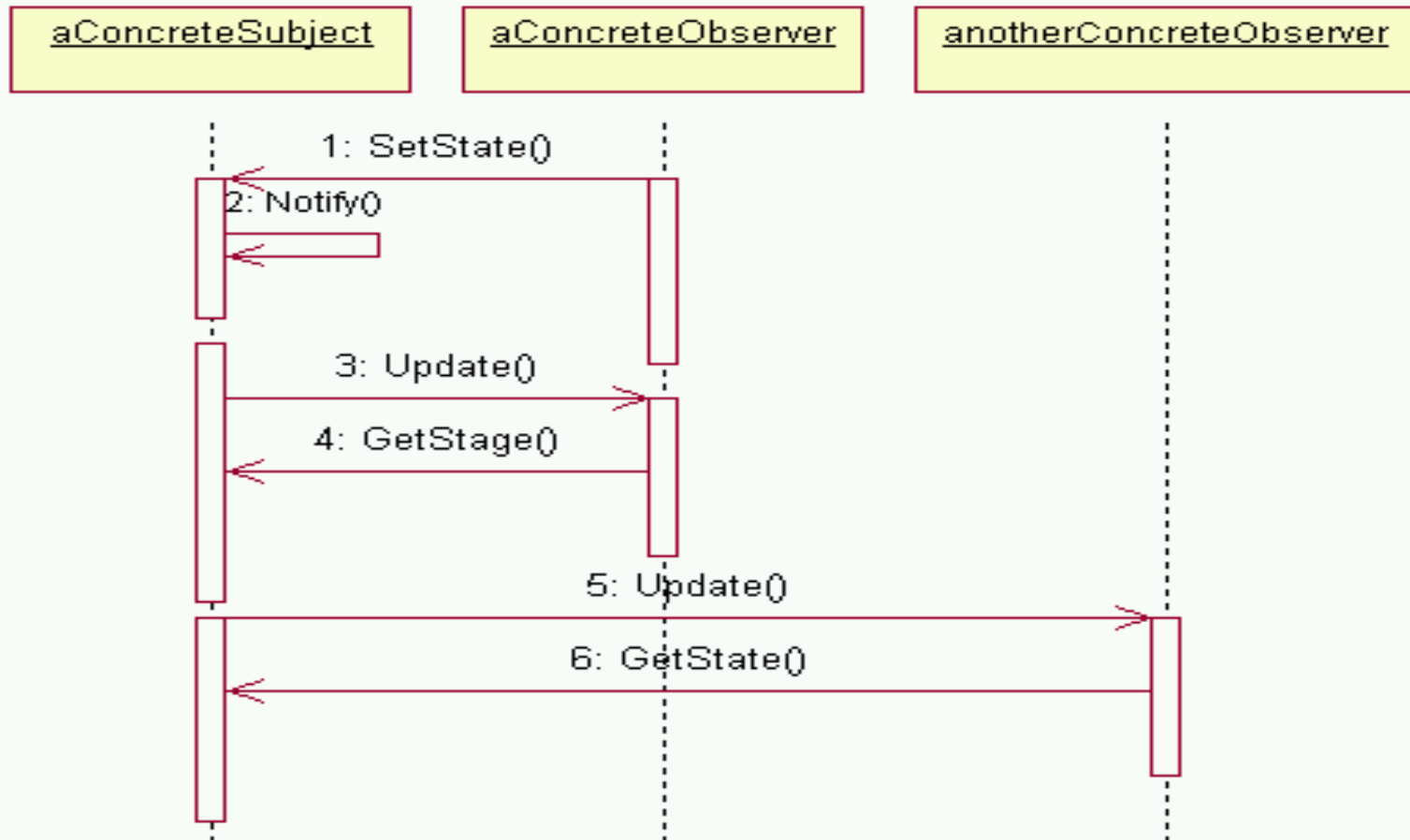
# Collaborations

- 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.



# Collaborations



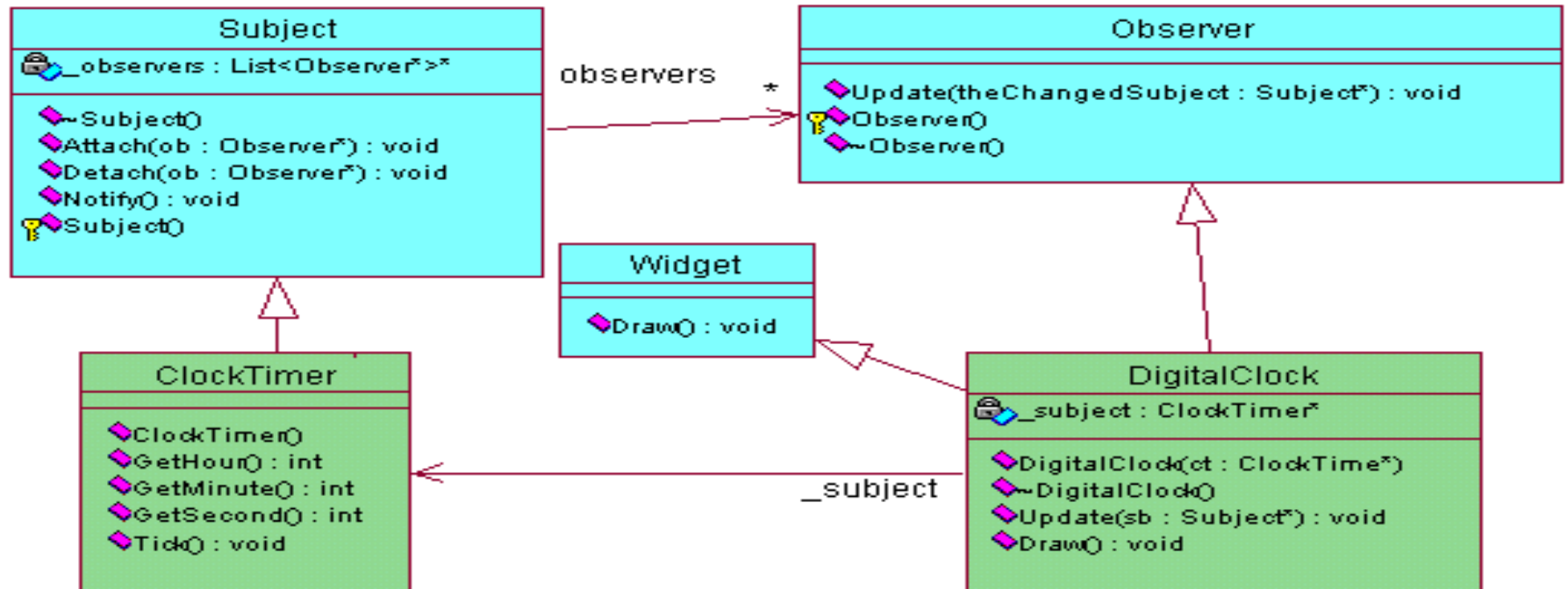
NOTE: How observer object that initiates the change request postpones its update until it gets a notification from the subject.



# Consequences

- ▶ Abstract coupling between Subject and Observer
- ▶ Support for broadcast communication.
- ▶ Unexpected updates.

# Model



```

//Client code
ClockTimer* timer = new ClockTimer;
DigitalClock* digitalClock = new DigitalClock(timer);

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}
    
```

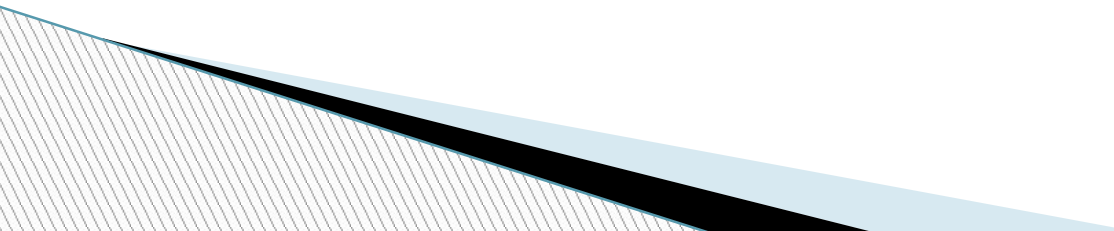
```

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.
}

// draw the digital clock
    
```

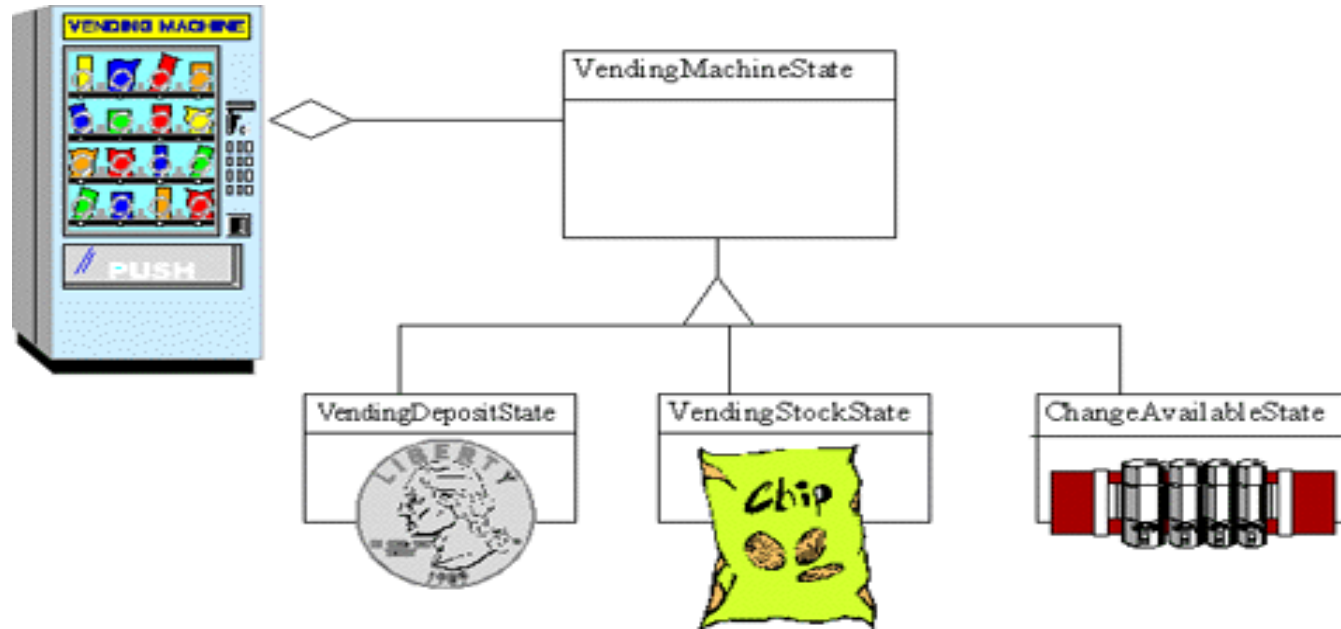
# Related patterns

- ▶ Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.
  - ▶ Singleton: The ChangeManager may use the Singleton pattern to make it unique and globally accessible.
- 

# State

# STATE

Object Diagram for  
*State* using Vending  
Machine Example



The *State* pattern allows an object to change its behavior when its internal state changes.

This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion

## Intent

- ▶ Allow an object to alter its behavior when its internal state changes. The object will appear to change its class

## **Also Known As**

- Objects for States
- 

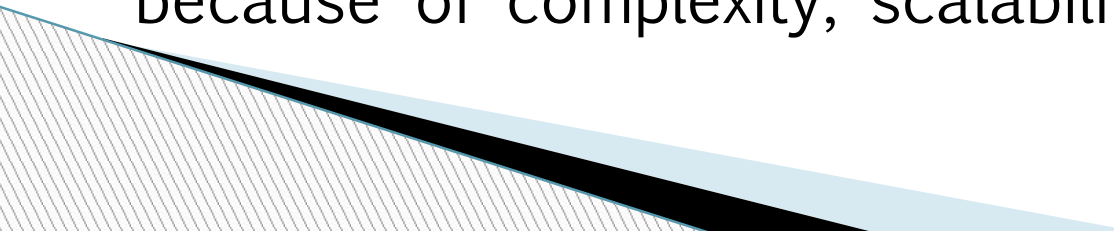
Problem:

A object can be in one of several different states, i.e.  
its behavior is a function of its state

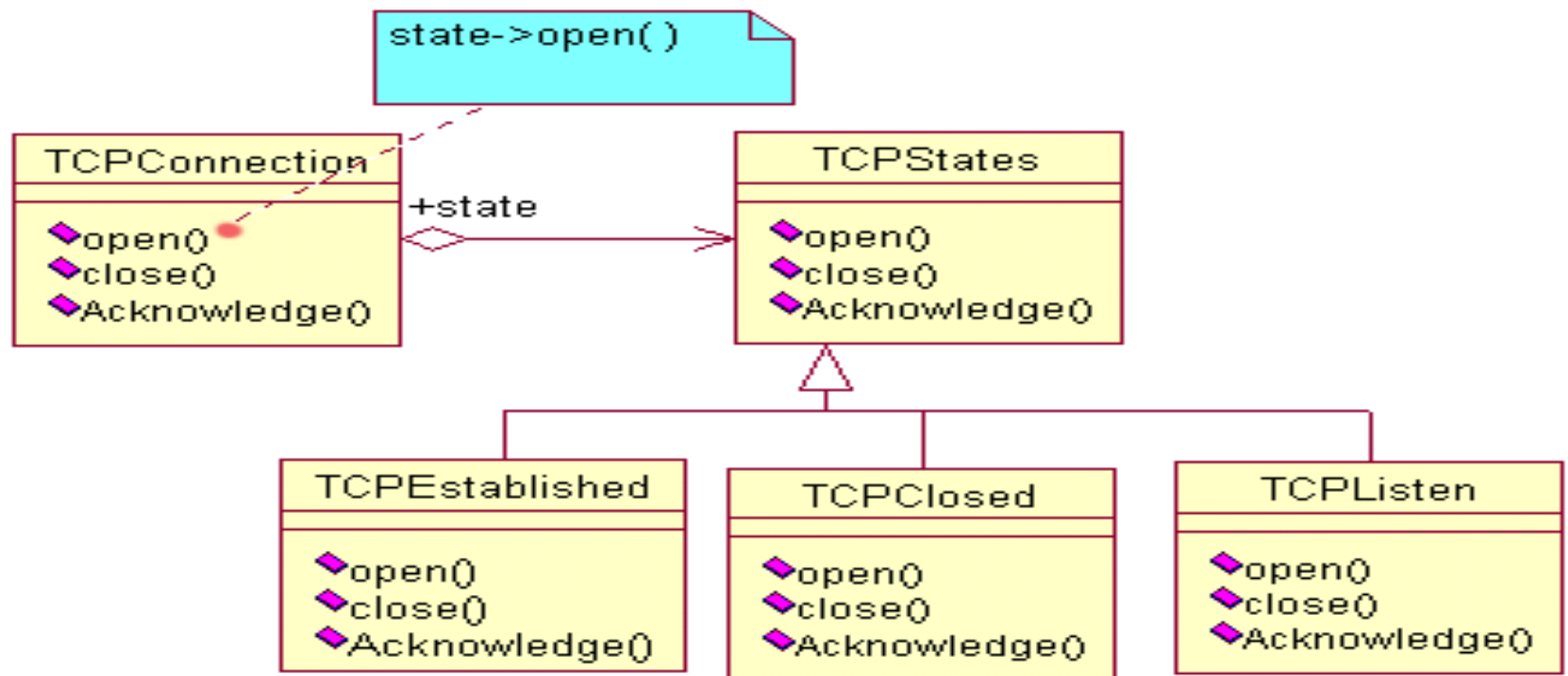
When an object receives request form other objects, it  
responds differently ( i.e. changes its behavior )  
depending on its current state

Such an application is likely to be characterized by  
large and numerous case statements

But we know that conditional logic is undesirable  
because of complexity, scalability or duplication.



## Illustration : Example

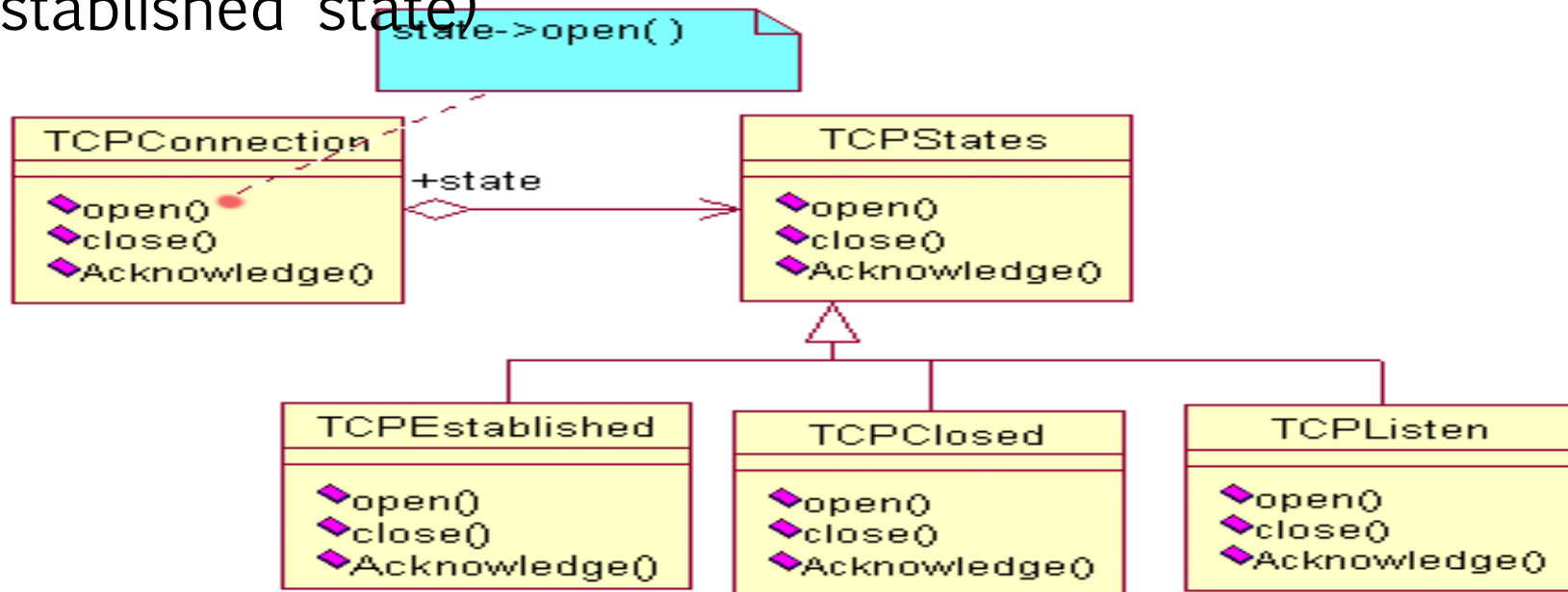


- Consider a class `TCPConnection` that represents a network connection.
- A `TCPConnection` object can be in one of several different states: Established, Listening, closed.



# Illustration : Problem

- ▶ When TCPconnection object receives requests from other objects, it responds differently depending on its current state.
- ▶ The effect of an open request depends on the state of the connection (i.e. closed state or its established state)



## **Illustration : Solution**

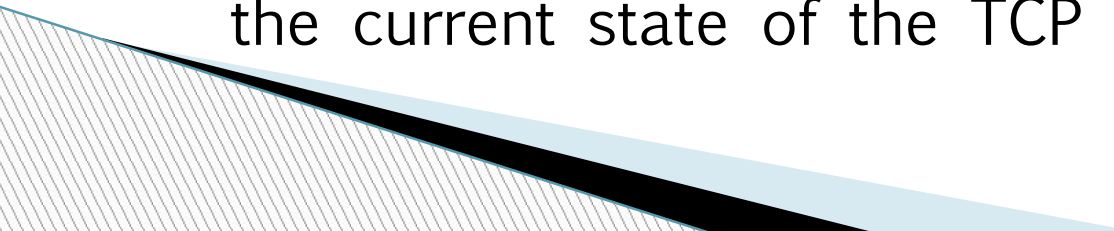
Introduce an abstract class called TCPState to represent the states of the network connection.

The TCPState class declares an interface common to all classes that represent different operational states.

Subclasses of TCPState implement State-specific behavior like TCPClosed implements behavior particular to Closed state

& TCPEstablished behavior particular to Established.

Maintain in TCPConnection a state object ( ie. An instance of a subclass of TCPState) that represent the current state of the TCP connection.



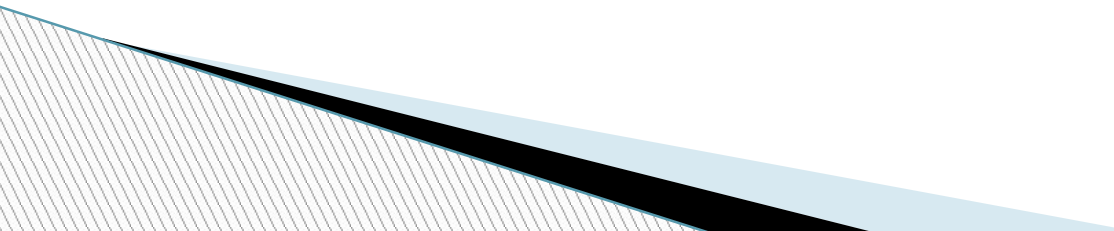
## Applicability

**Use the State pattern in either of the following cases**

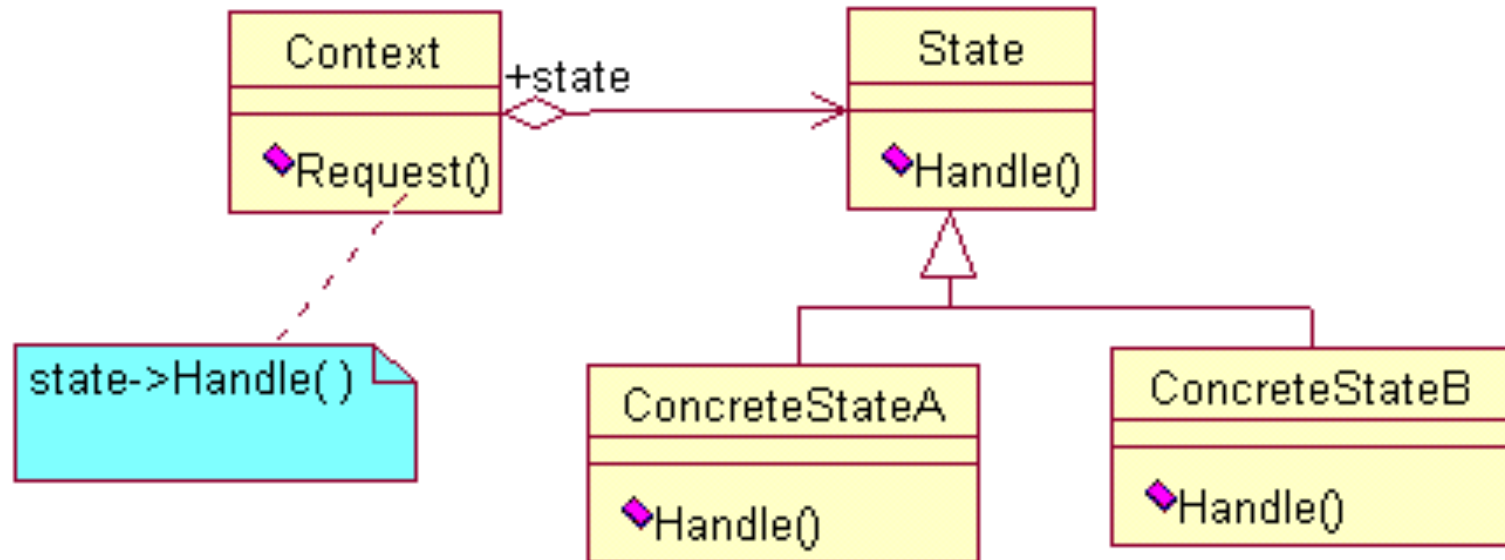
- ▶ An object's behavior depends on its state, and it must change its behavior at runtime depending on that state.
- ▶ Operations have large, multipart conditional statements that depends on the object's state.

Often, several operations will contain this same conditional structure.

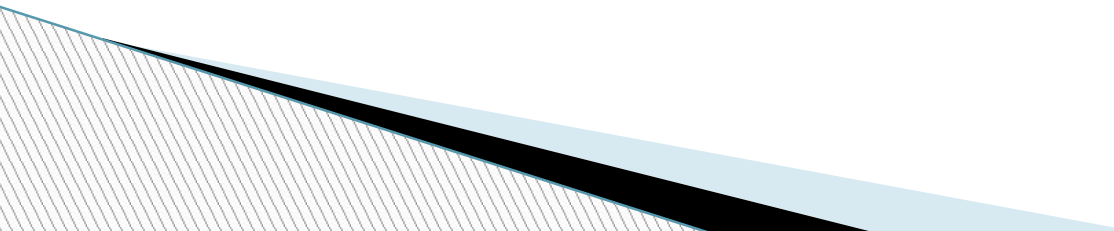
The state pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.



# Structure



# Collaborations

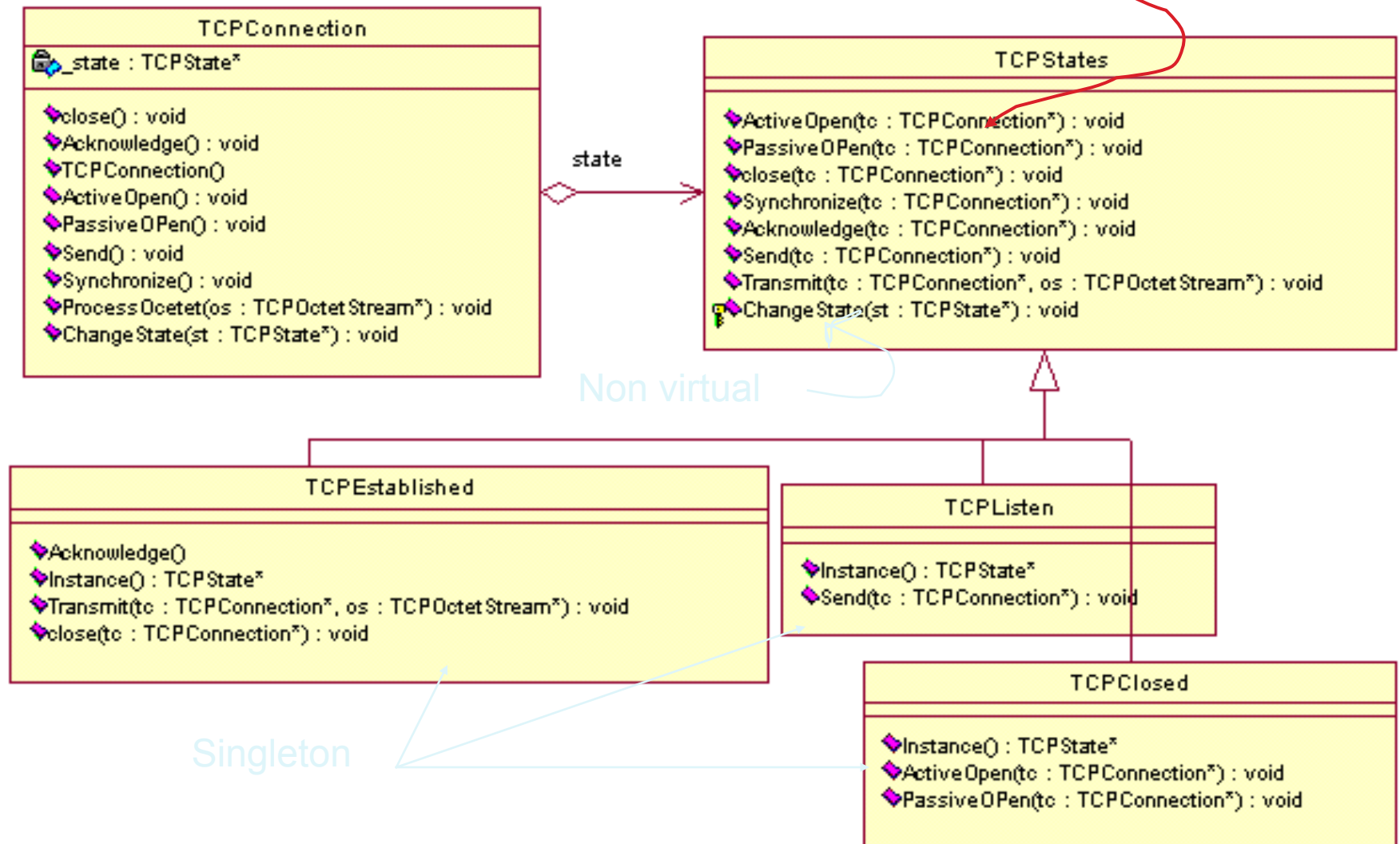
- ▶ Context delegates state-specific request to the current concreteState object.
  - ▶ A context may pass itself as an argument to the state object handling the request. This lets the state object access the context if necessary
  - ▶ Context is the primary interface for clients. Clients can configure a context with state objects. Once a context is configured, its clients don't have to deal with the state object directly
  - ▶ Either context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.
- 

# Consequences

- ▶ It localizes state-specific behavior and partitions behavior for different states.
- ▶ It makes state transitions explicit.
- ▶ State objects can be shared.

# Model

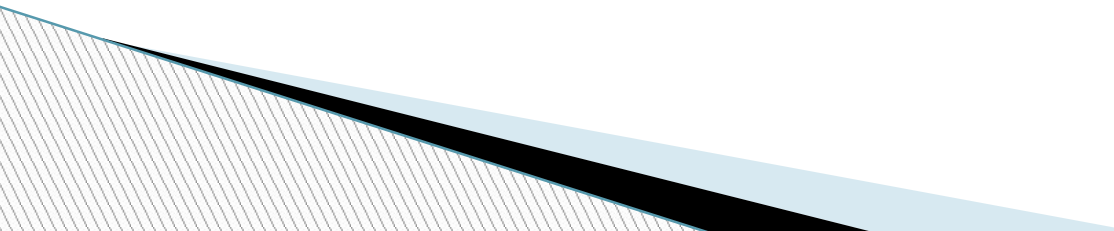
Note Ref of TCPConnection is taken



Code next slide>>>

## Code:

```
▶ //Default behavior:  
▶ /* virtual */  
▶ void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }  
▶ void TCPState::ActiveOpen (TCPConnection*) { }  
▶ void TCPState::PassiveOpen (TCPConnection*) { }  
▶ void TCPState::Close (TCPConnection*) { }  
▶ void TCPState::Synchronize (TCPConnection*) { }  
  
▶ /*  
▶ non - virtual */  
  
▶ void TCPState::ChangeState (TCPConnection* t, TCPState* s) {  
▶ t->ChangeState(s);  
▶ }
```



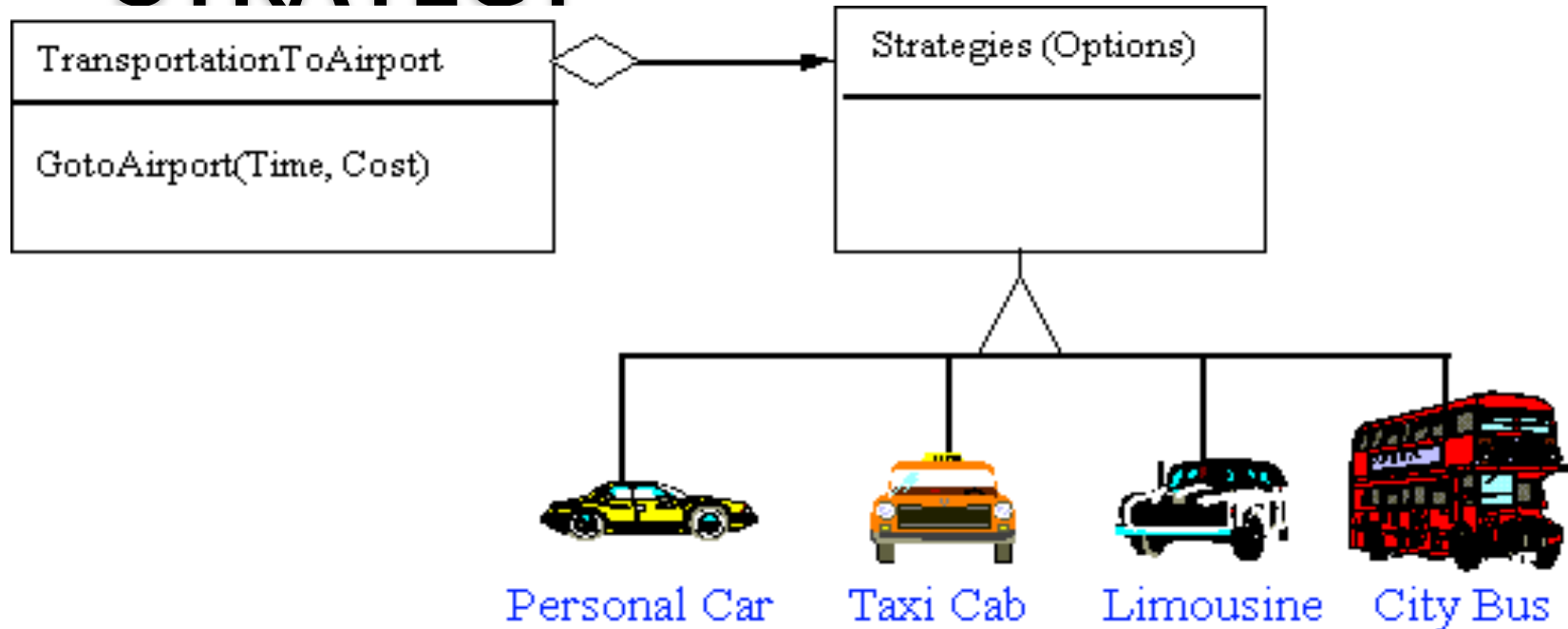


## Code:

- ▶ **/\* Individual state classes will have there own specafic methods \*/**
- ▶ `void TCPEstablished::Close (TCPConnection* t) {`
- ▶  `// send FIN, receive ACK of FIN`
- ▶  `ChangeState(t, TCPListen::Instance());`
- ▶ `}`
- ▶ `void TCPEstablished::Transmit (`
- ▶  `TCPConnection* t, TCPOctetStream* o`
- ▶ `) {`
- ▶  `t->ProcessOctet(o);`

# Strategy

# STRATEGY



A *Strategy* defines a set of algorithms that can be used interchangeably.

Modes of transportation to an airport is an example of a *Strategy*. Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the *Strategy* based on tradeoffs between cost, convenience, and time.

# STRATEGY

Intent

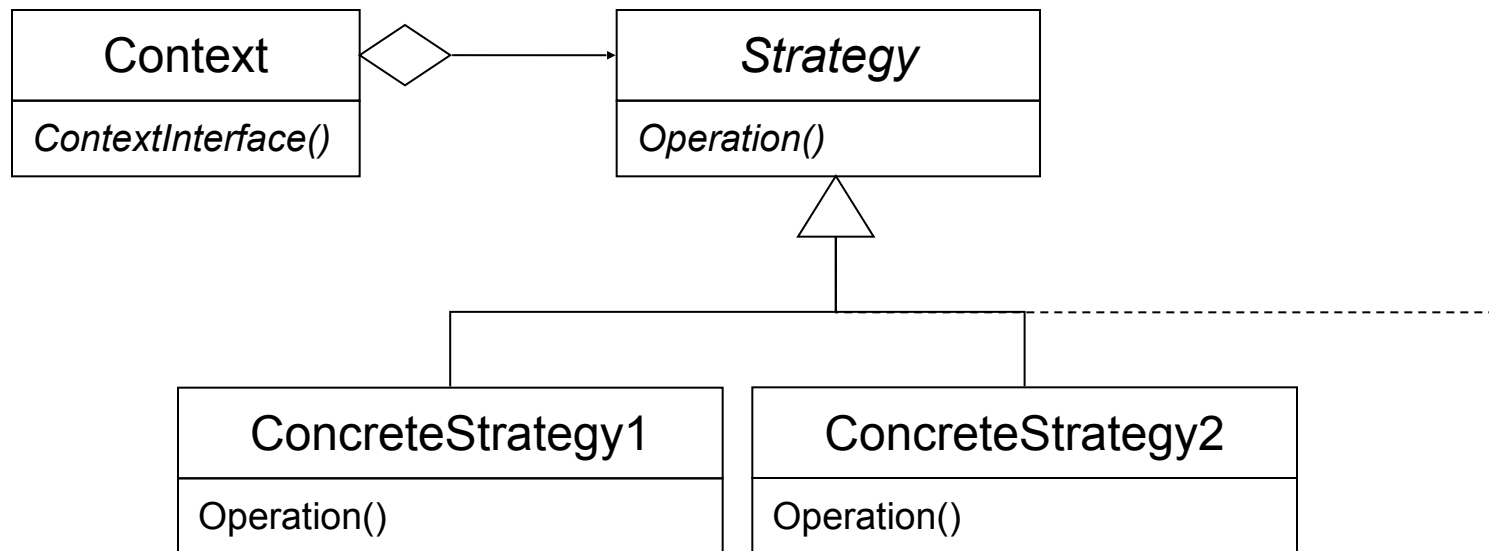
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it

Also Known As  
policy



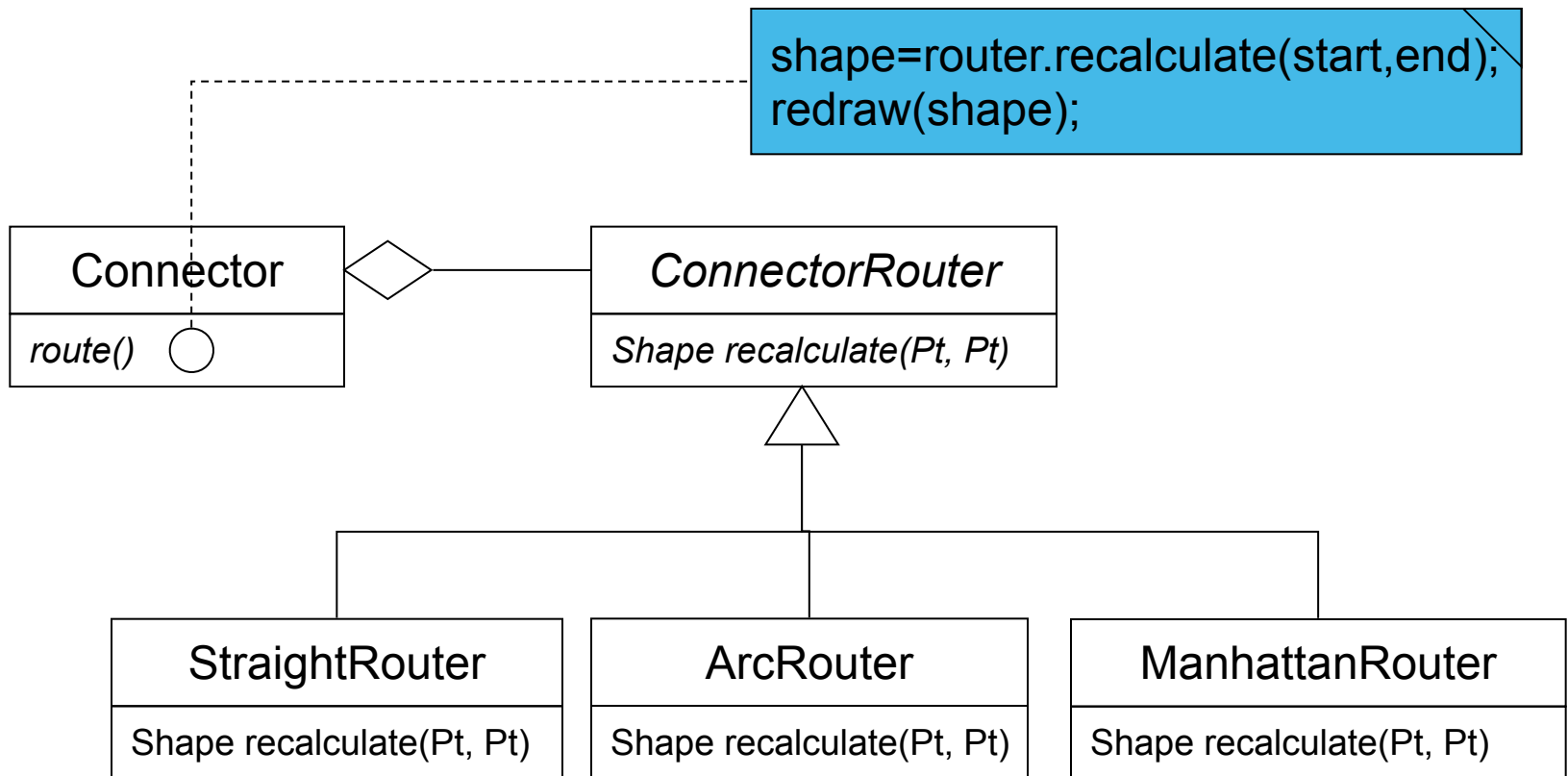
# Strategy

- ▶ Make algorithms interchangeable---"changing the guts"
- ▶ Alternative to subclassing
- ▶ Choice of implementation at run-time
- ▶ Increases run-time complexity

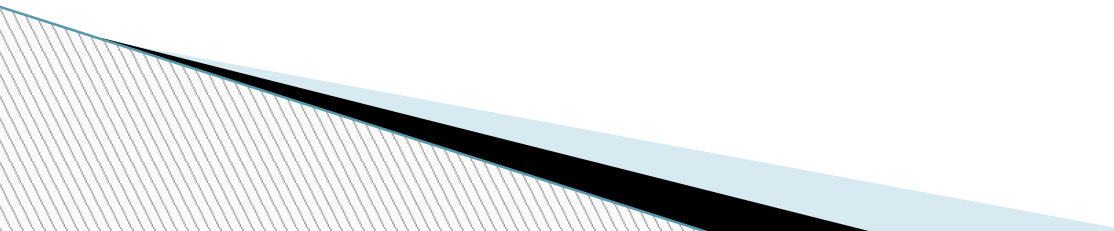


# Strategy - Example

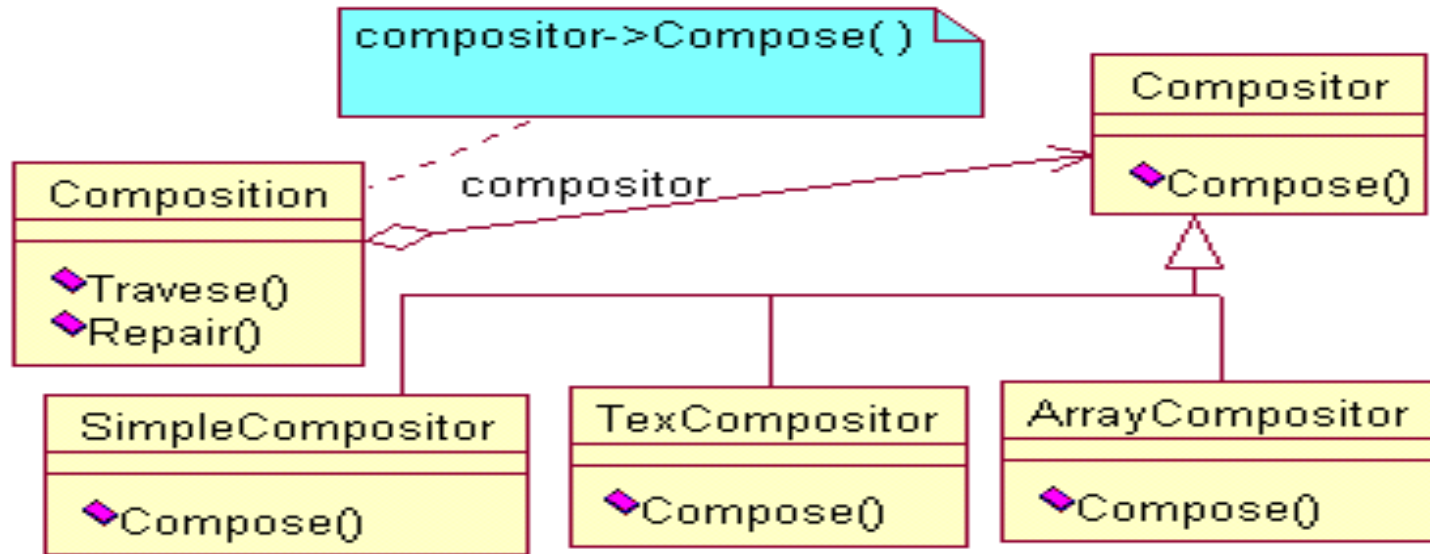
- ▶ Example: drawing different connector styles



# Problem

- ▶ If clients have potentially generic algorithms embedded in them, it is difficult to: reuse these algorithms, exchange algorithms, decouple different layers of functionality, and vary your choice of policy at run-time. These embedded policy mechanisms routinely manifest themselves as multiple, monolithic, conditional expressions
- 

# Illustration : Example Text Editors



Many algorithms exist for performing a particular task [e.g – *breaking a stream of text into lines*]. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons



# Illustration

Clients that need these algorithm ( *line breaking* ) get more complex if they include algorithm's code ( *line-breaking code*). That makes client bigger and harder to maintain, especially if they support multiple algorithms( *line breaking algorithms*).

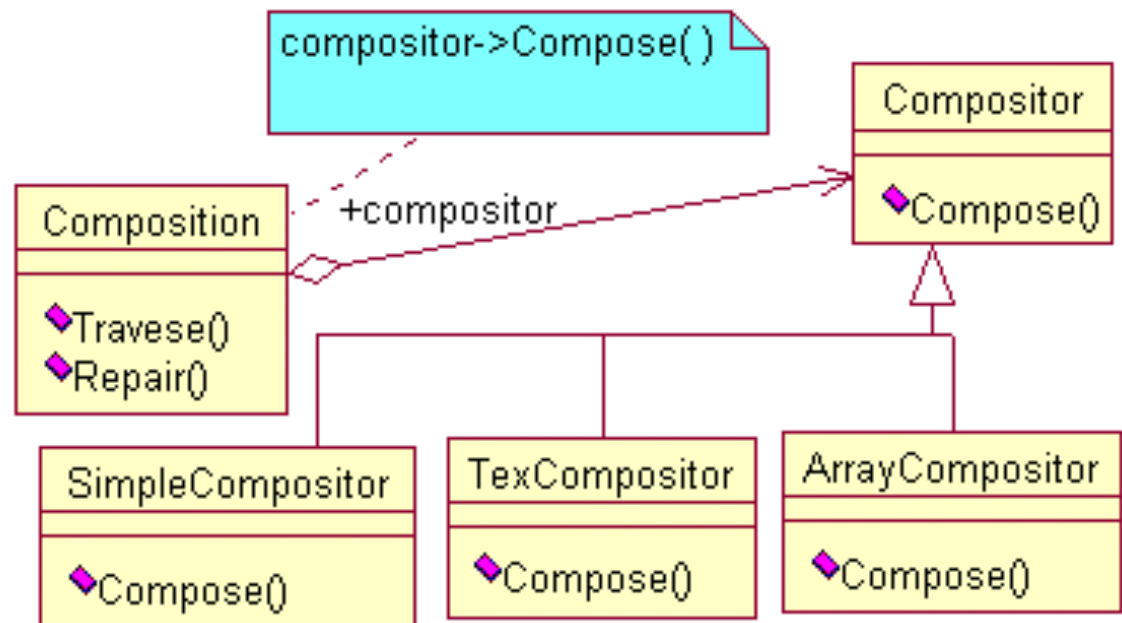
- Different algorithms will be appropriate at different times. We don't want to support multiple algorithms ( *line breaking algorithms* ) if we don't use them all
- Its difficult to add new algorithms and vary existing ones when a particular algorithm ( *line breaking* ) is an integral part of client.

## Illustration: solution

The above explained problem can be avoided by defining classes that encapsulate different algorithm(*line breaking algorithms*).

An algorithm that's encapsulated in this way is called a strategy.

**Note:** separate algorithm is encapsulated in individual objects that conform to common interface



# Solution

Replace the many, monolithic, conditional constructs with a Strategy inheritance hierarchy and dynamic binding.

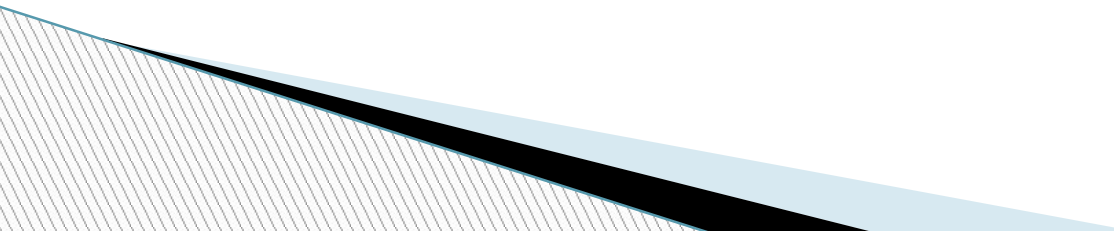
- Identify the protocol that provides the appropriate level of abstraction, control, and interchangeability for the client.
- Specify this protocol in an abstract base class.
- Move all related conditional code into their own concrete derived classes.
- Configure the original application with an instance of the Strategy hierarchy, and delegate to that "contained" object whenever the "algorithm" is required.

**"Strategies can provide different implementations of the same behavior. The client can choose among Strategies with different time and space trade-offs."**

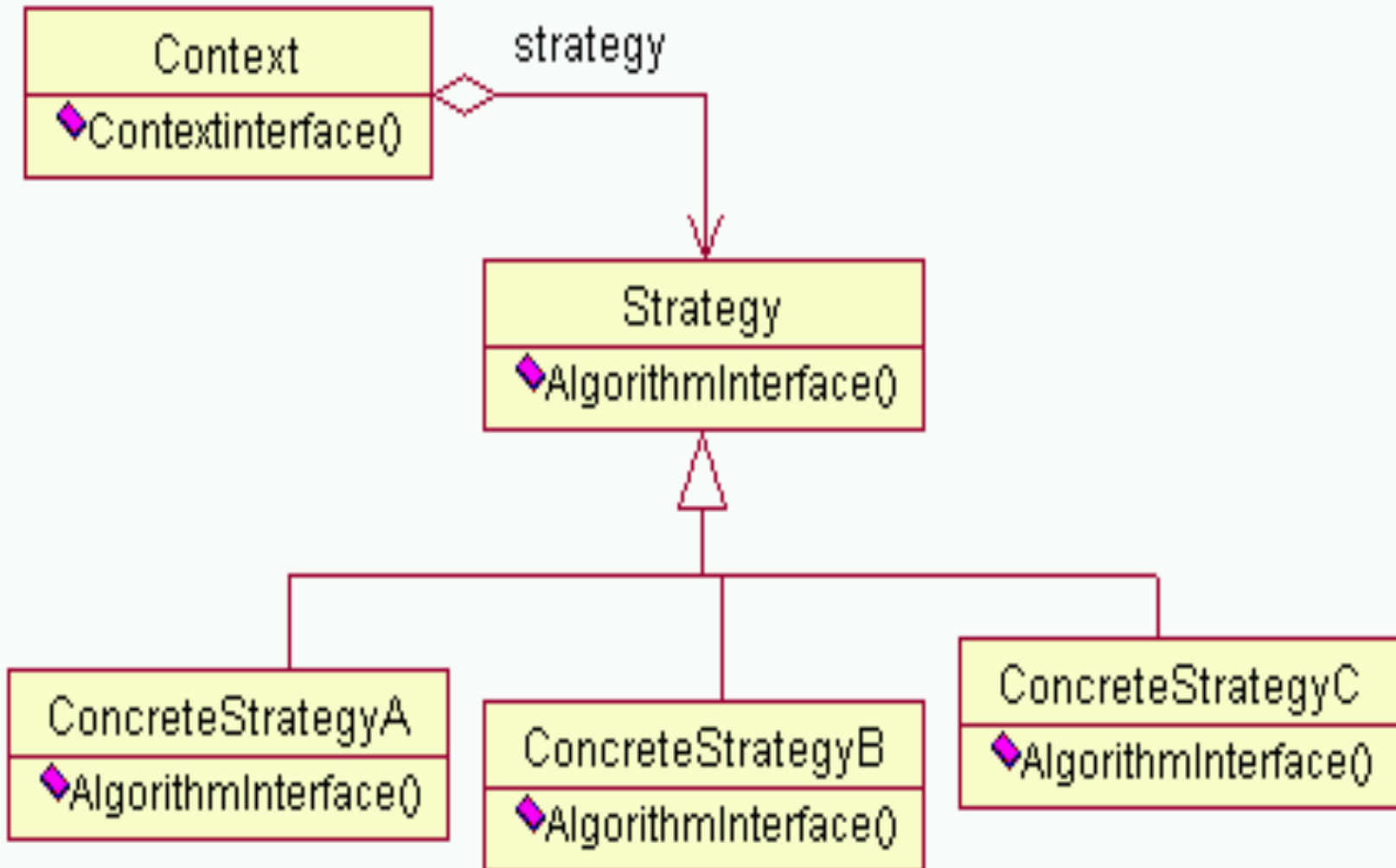


# Applicability

Use the Strategy pattern when

- ▶ Many related classes differ only in their behavior.
  - ▶ You need different variants of an algorithm.
  - ▶ An algorithm uses data that clients shouldn't know about.
  - ▶ A class defines many behaviors, and these appear as multiple conditional statement in its operations.
- 


# Structure



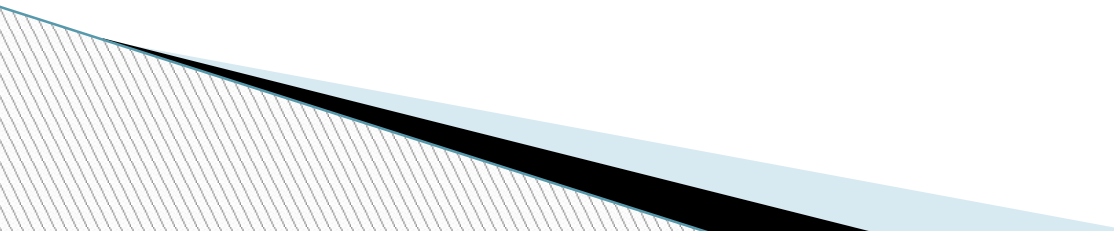
# Collaborations

- ▶ Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.

A context forward requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

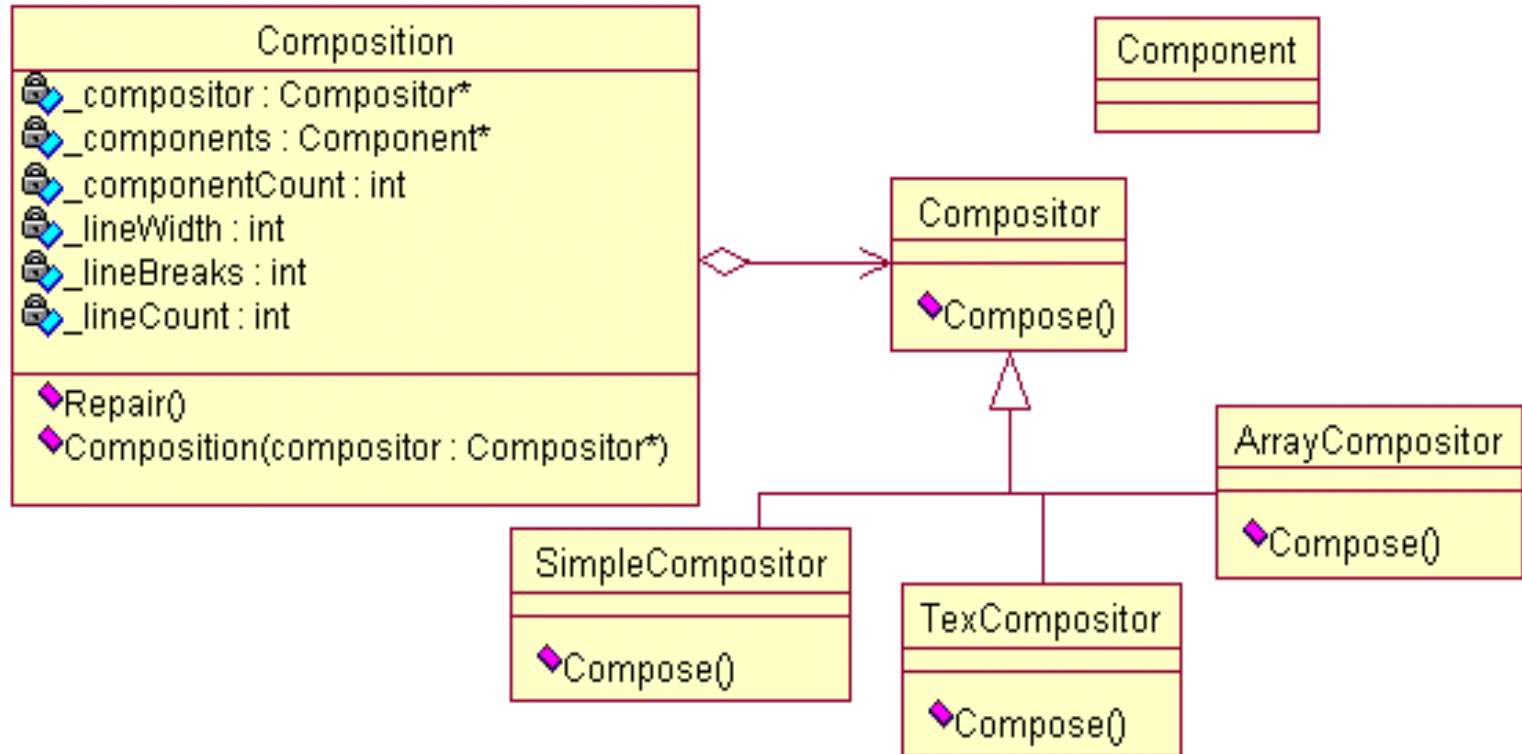


# Consequences

1. **Defines Families of related algorithms or behaviors for contexts to reuse.**
  2. **An alternative to sub-classing** : encapsulating the algorithm in separate strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand and extend
  3. **Strategies eliminate conditional statements:**
  4. **A choice of implementation:** provide different implementations of the same behaviors
  5. **Clients must be aware of different Strategies** : client must be exposed to implementation issues. Because client must understand how strategies differ before it can select the appropriate one.
  6. **Communication overhead between Strategy and context.**
  7. **Increased number of objects:**
- 

# Model

## Strategy



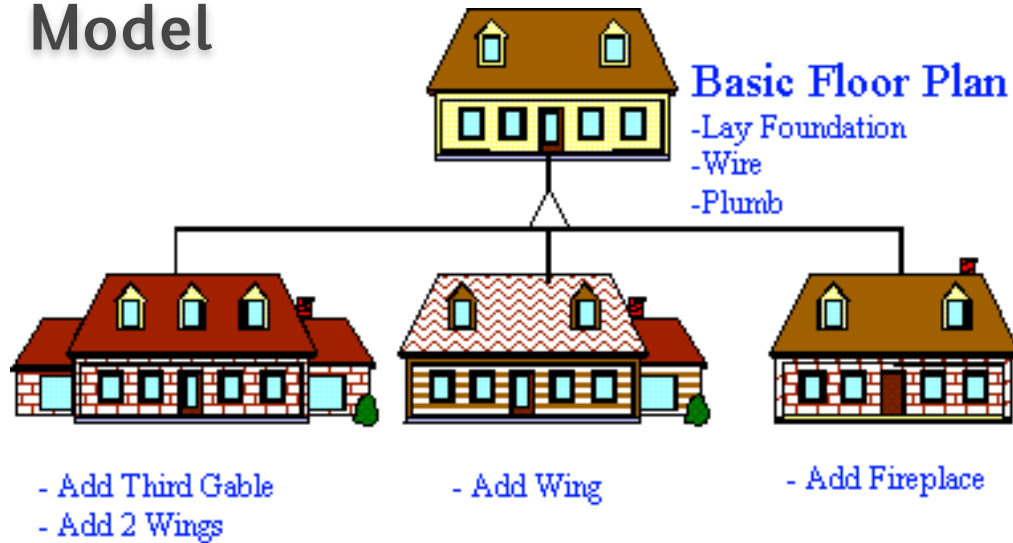
## Related Patterns :

flyweight: Strategy objects often make good flyweights.



# Template

# Model



Basic Floor Plan Example of *Template Method*

The *Template Method* defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.

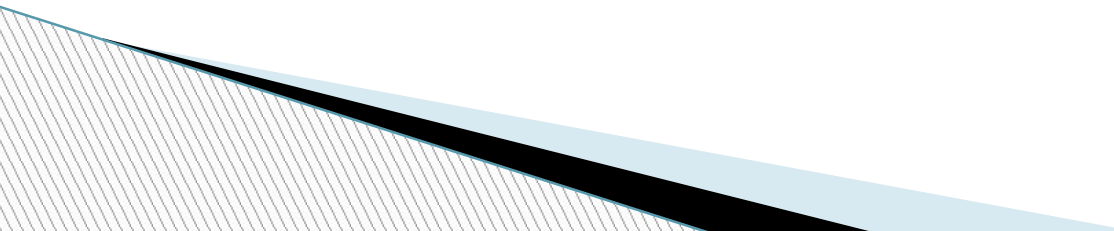
## Variations added to Template Floor Plan

Home builders use the *Template Method* when developing a new subdivision. A typical subdivision consists of a limited number of floor plans, with different variations available for each floor plan. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the latter stages of construction to produce a wider variety of models.

## Template Method

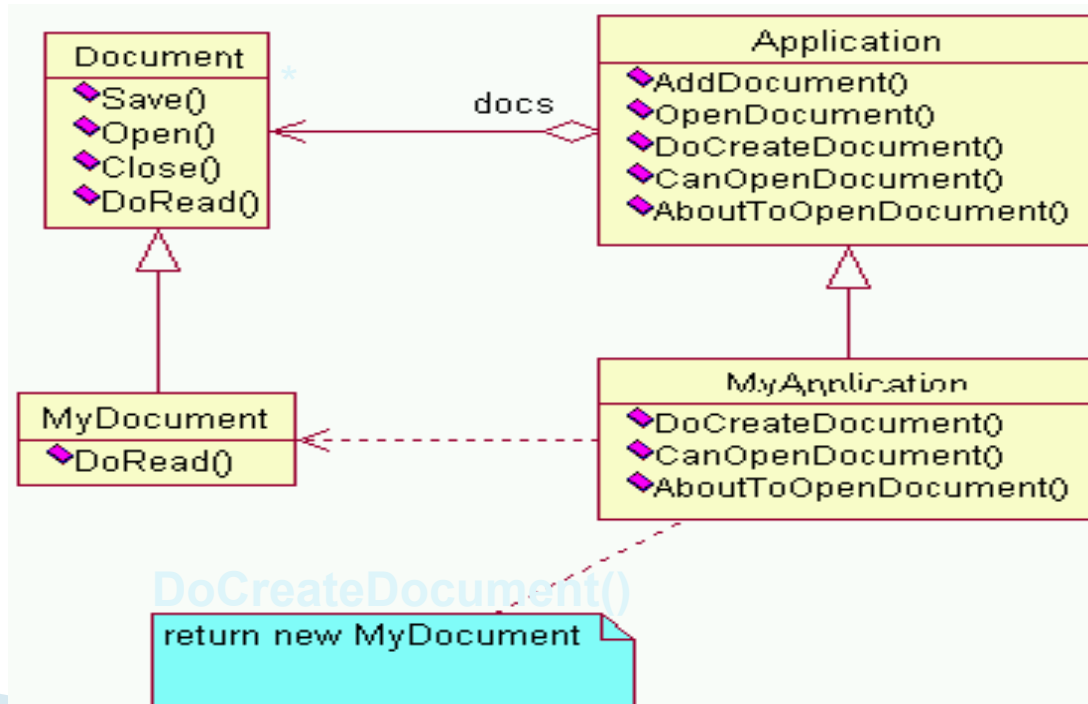
### Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

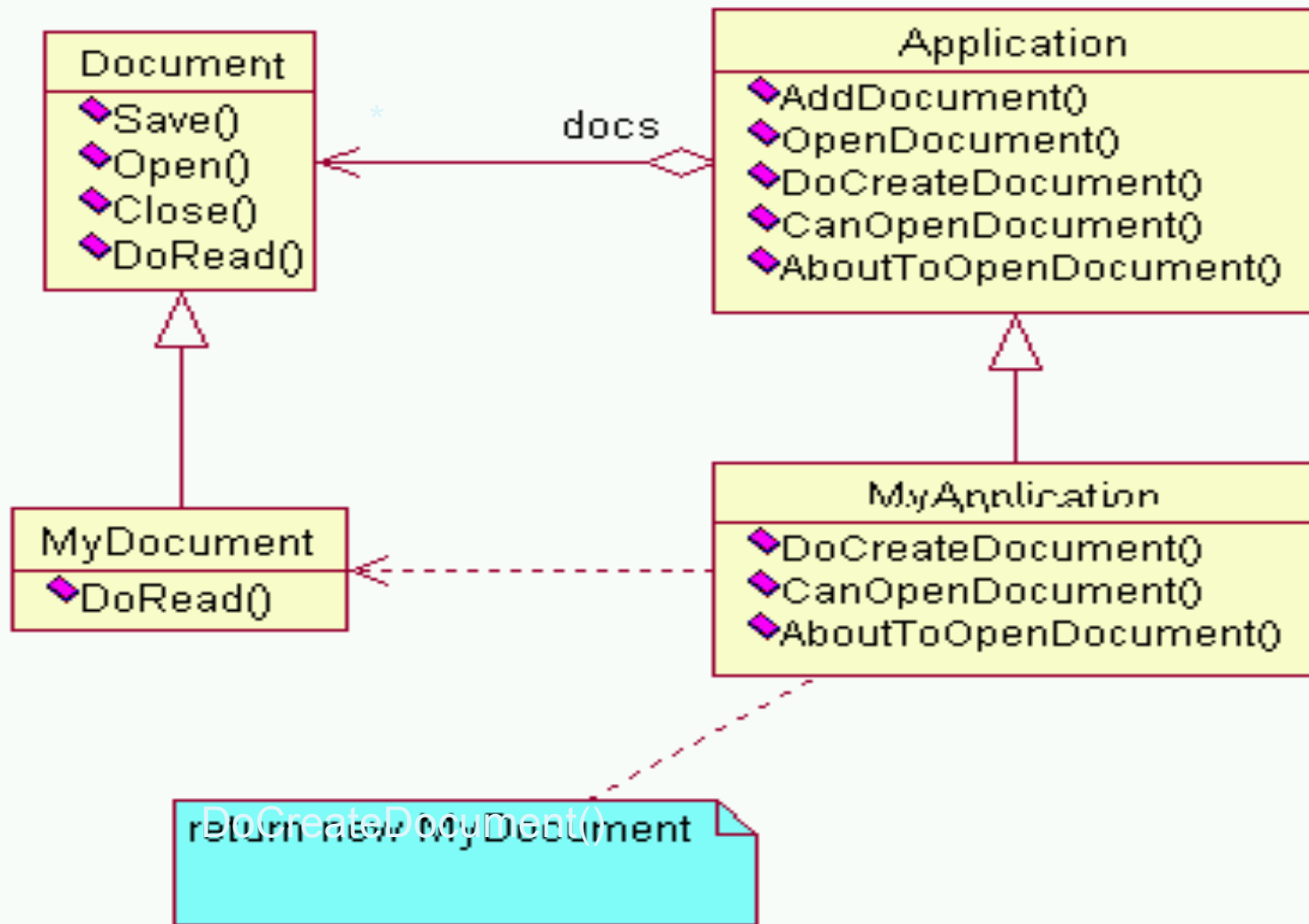


## Problem:

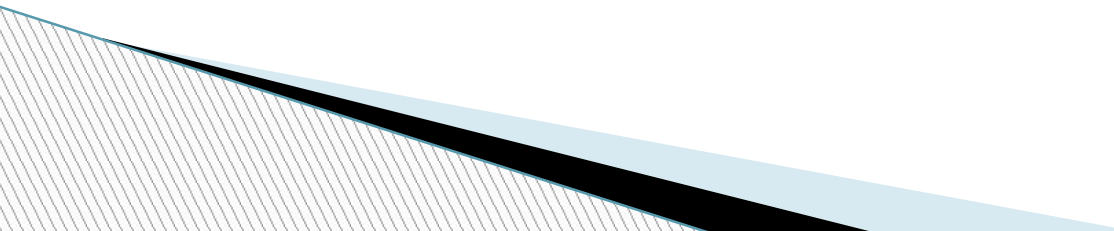
Given a template method if we need to customize or redefine a particular set in the algorithm one may have to override the template function this causes duplication and an inefficient usage of template methods.



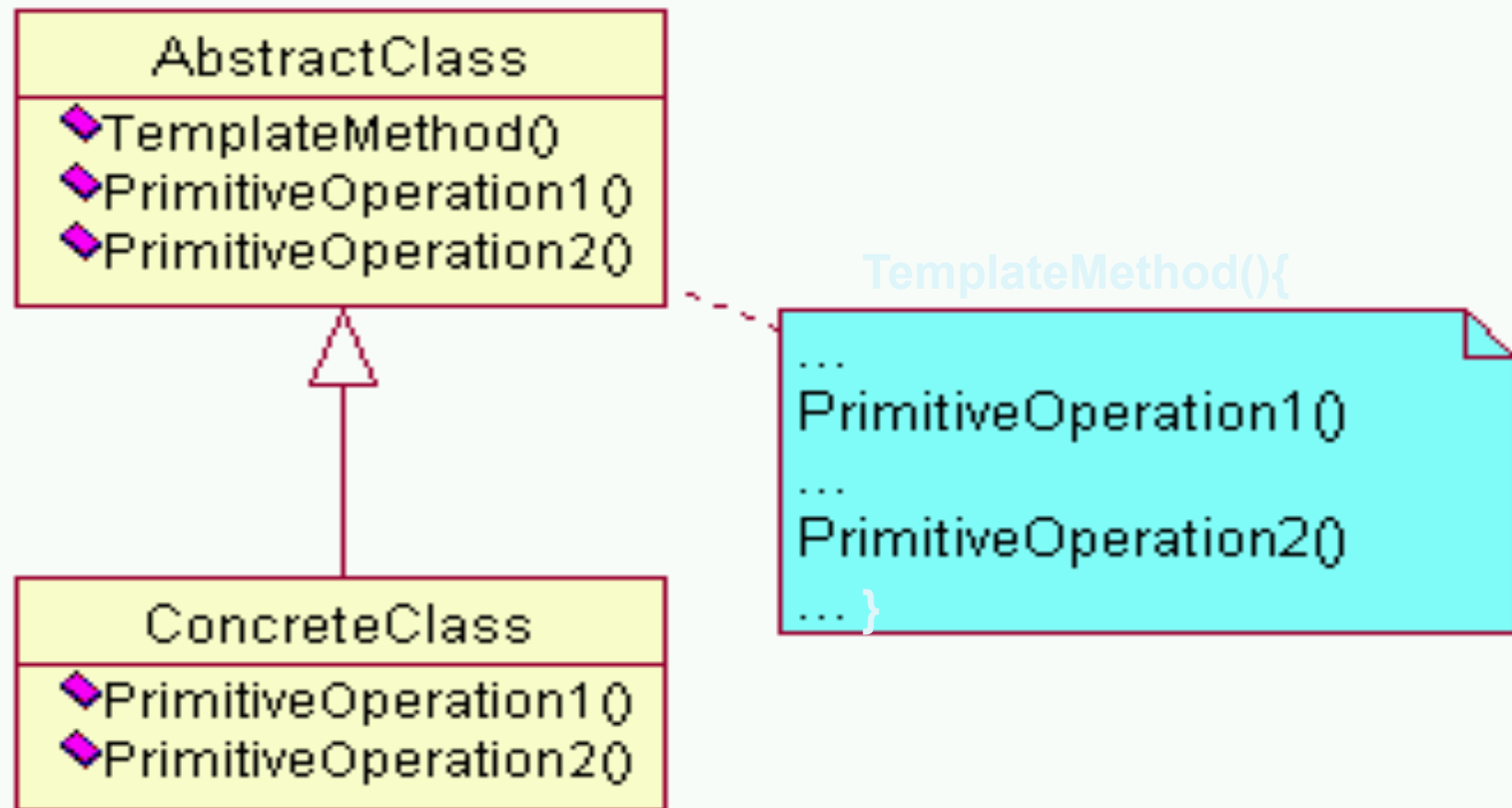
# Illustration



# Solution

- ▶ A template method pattern gives a solution.
  - ▶ A template method defines an algorithm in terms of abstract operation the subclasses override to provide concrete behavior.
- 

# Structure

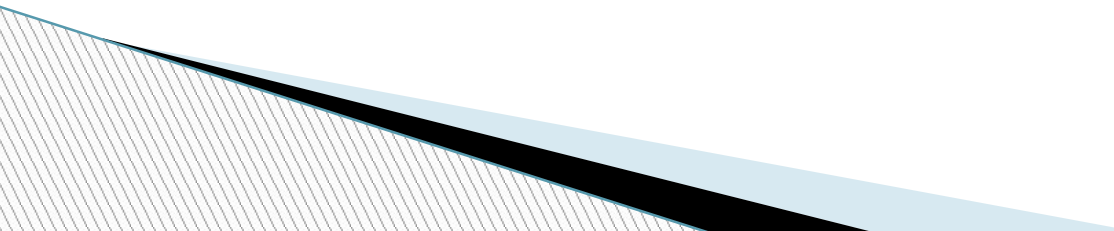


# Applicability

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

When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.

To control subclasses extensions. You can define a template method that calls “hook” operations (see Consequences) at specific points, thereby permitting extensions only at those points.

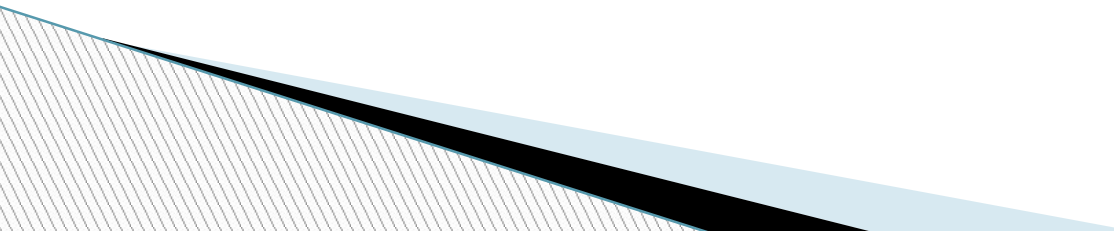




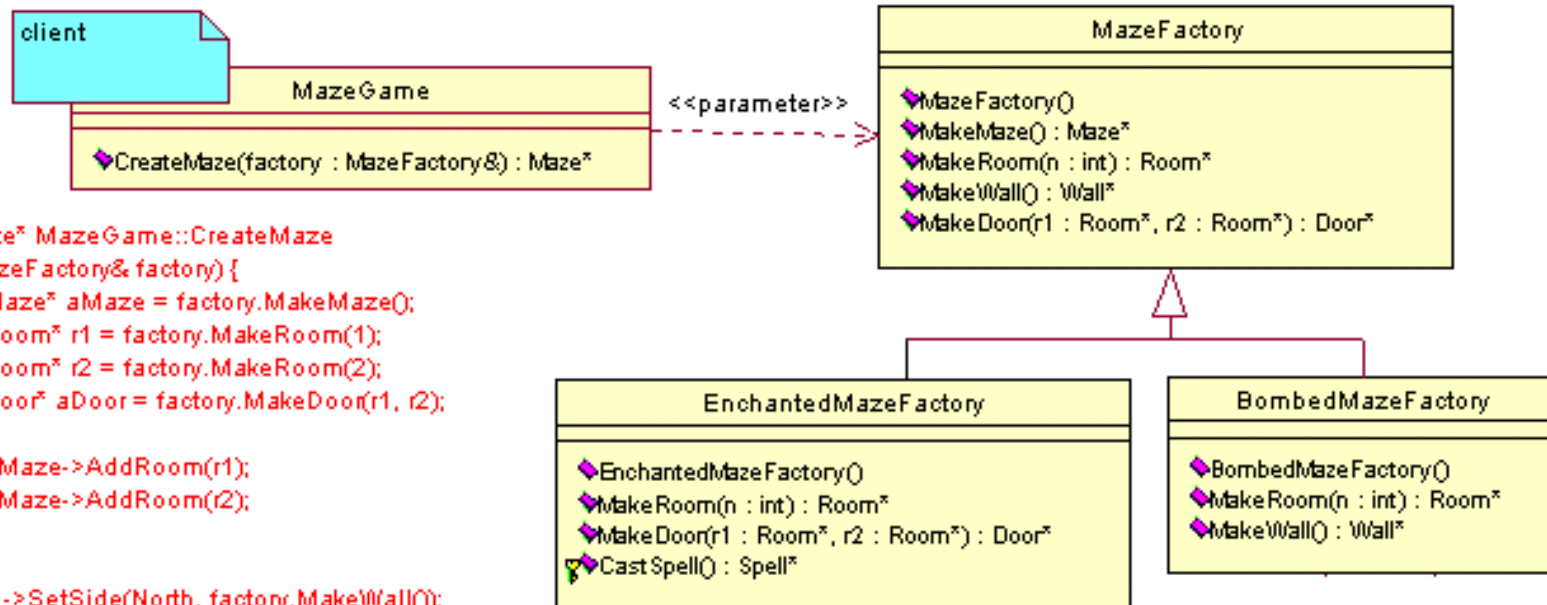
# Collaborations

- ConcreteClass relies on abstractClass to implement the invariant steps of the algorithm.

# Consequences

1. Concrete operations
  2. Concrete `AbstractClass` operations
  3. Primitive operations
  4. Factory methods
  5. Hook operations, which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.
- 

# Model



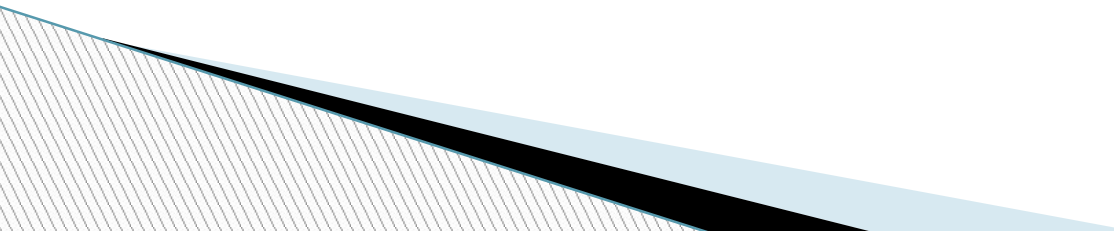
```
Maze* MazeGame::CreateMaze
(MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
}

r1->SetSide(North, factory.MakeWall());
r1->SetSide(East, aDoor);
r1->SetSide(South, factory.MakeWall());
r1->SetSide(West, factory.MakeWall());

r2->SetSide(North, factory.MakeWall());
r2->SetSide(East, factory.MakeWall());
r2->SetSide(South, factory.MakeWall());
r2->SetSide(West, aDoor);
```

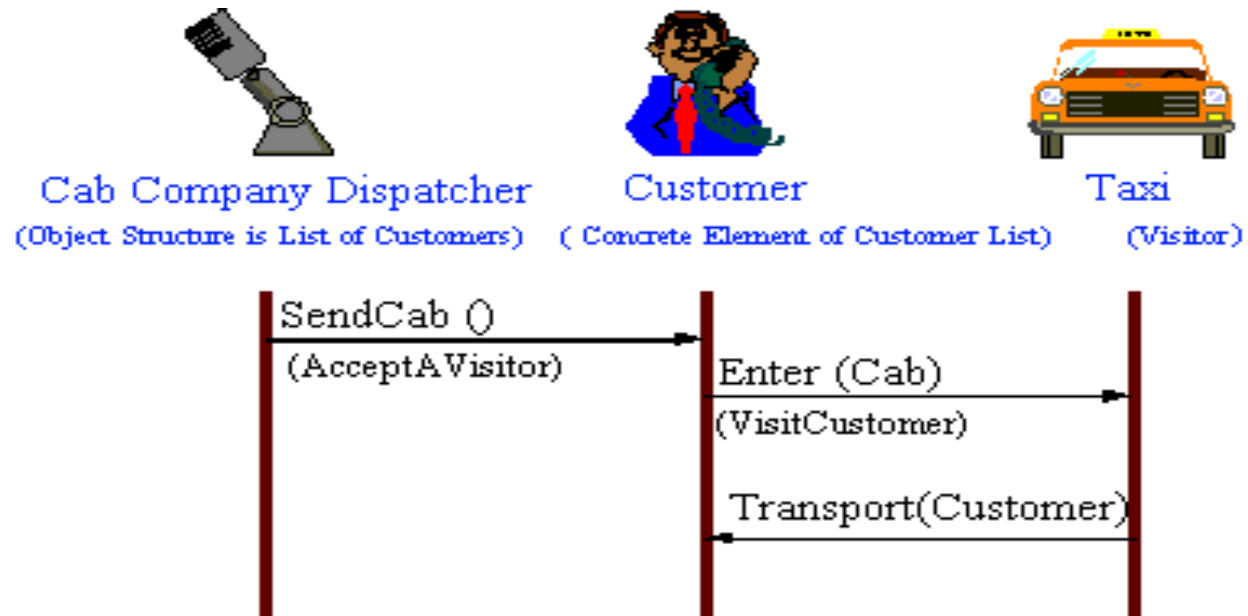
# Related Patterns

- ▶ Factory Methods are often called by template methods. In the Motivation example, the factory method `DoCreateDocument` is called by the template method `OpenDocument`.
  - ▶ Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.
- 

# Visitor

# VISITOR

Object Interaction  
Diagram for *Visitor* using  
Taxi Cab Example



The *Visitor* pattern represents an operation to be performed on the elements of an object structure, without changing the classes on which it operates.

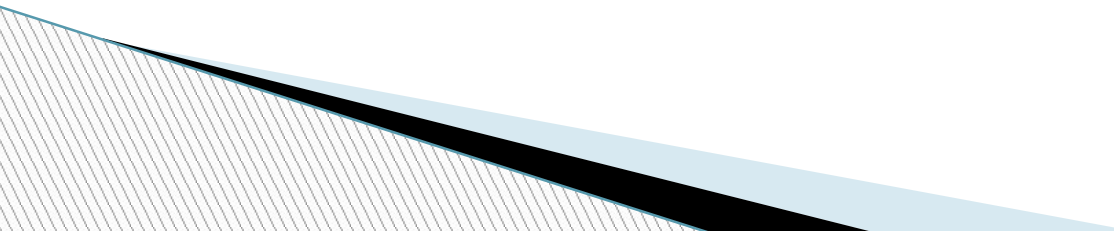
This pattern can be observed in the operation of a taxi company. When a person calls a taxi company he or she becomes part of the company's list of customers. The company then dispatches a cab to the customer (accepting a visitor). Upon entering the taxi, or *Visitor*, the customer is no longer in control of his or her own transportation, the taxi (driver) is.

# Intent

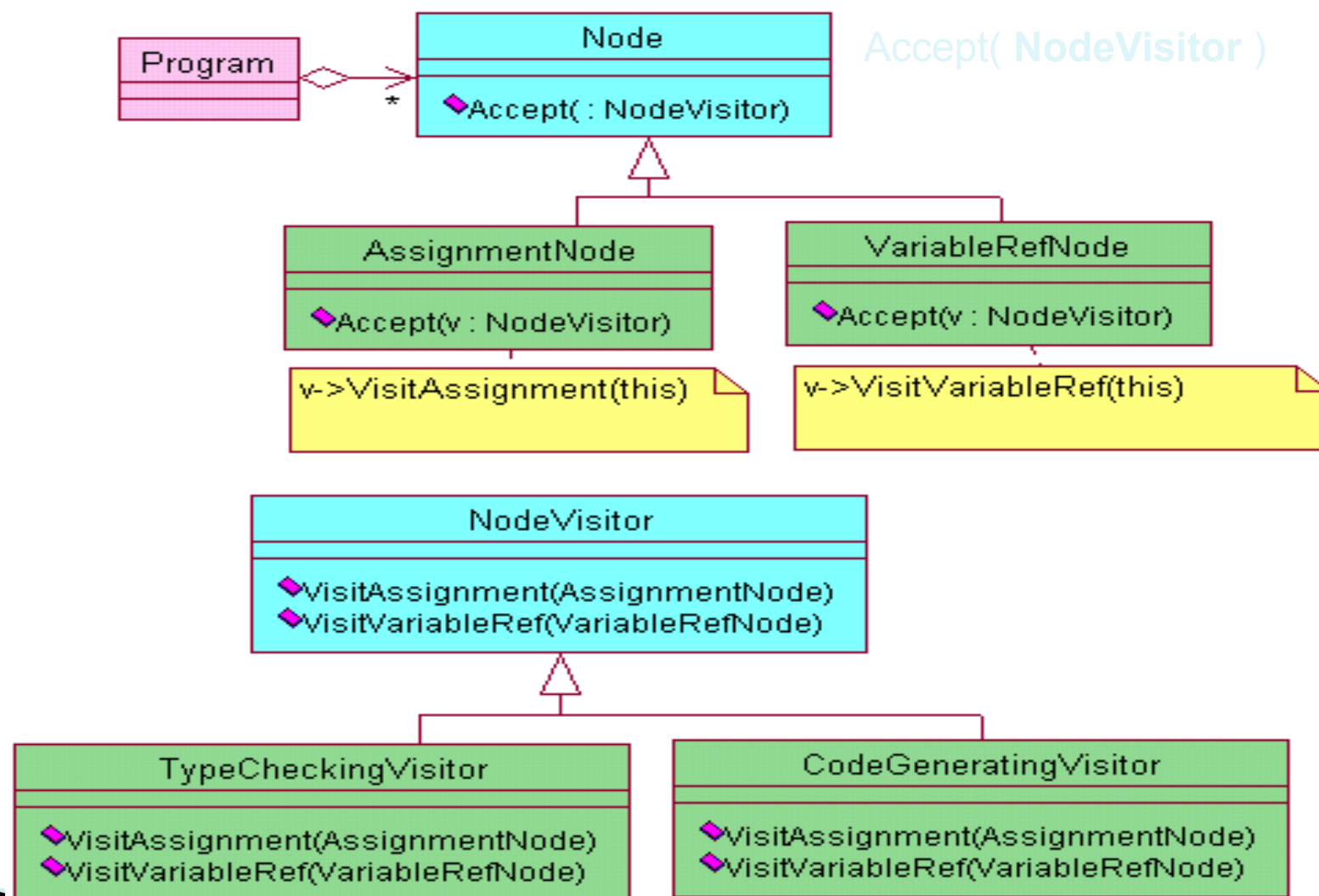
- ▶ Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

## **Problem:**

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.



# Illustration:





## Solution: Step by Step

The implementation proceeds as follows. Create a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.

- ▶ Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy. The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.
- ▶ Add a single pure virtual accept() method to the base class of the Element hierarchy. accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

## Solution: Step by Step ...

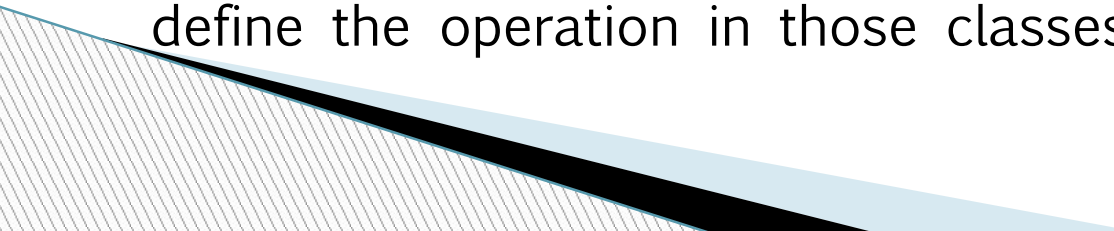
- ▶ Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.
- ▶ When Everything for "elements" and "visitors" is now set-up.
  - When the client needs an operation to be performed,
    - Client creates an instance of the Visitor object,
    - calls the accept() method on each Element object,
    - and passes the Visitor object.
- ▶ The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass. accept() dispatch plus visit() dispatch equals double dispatch.

# Applicability

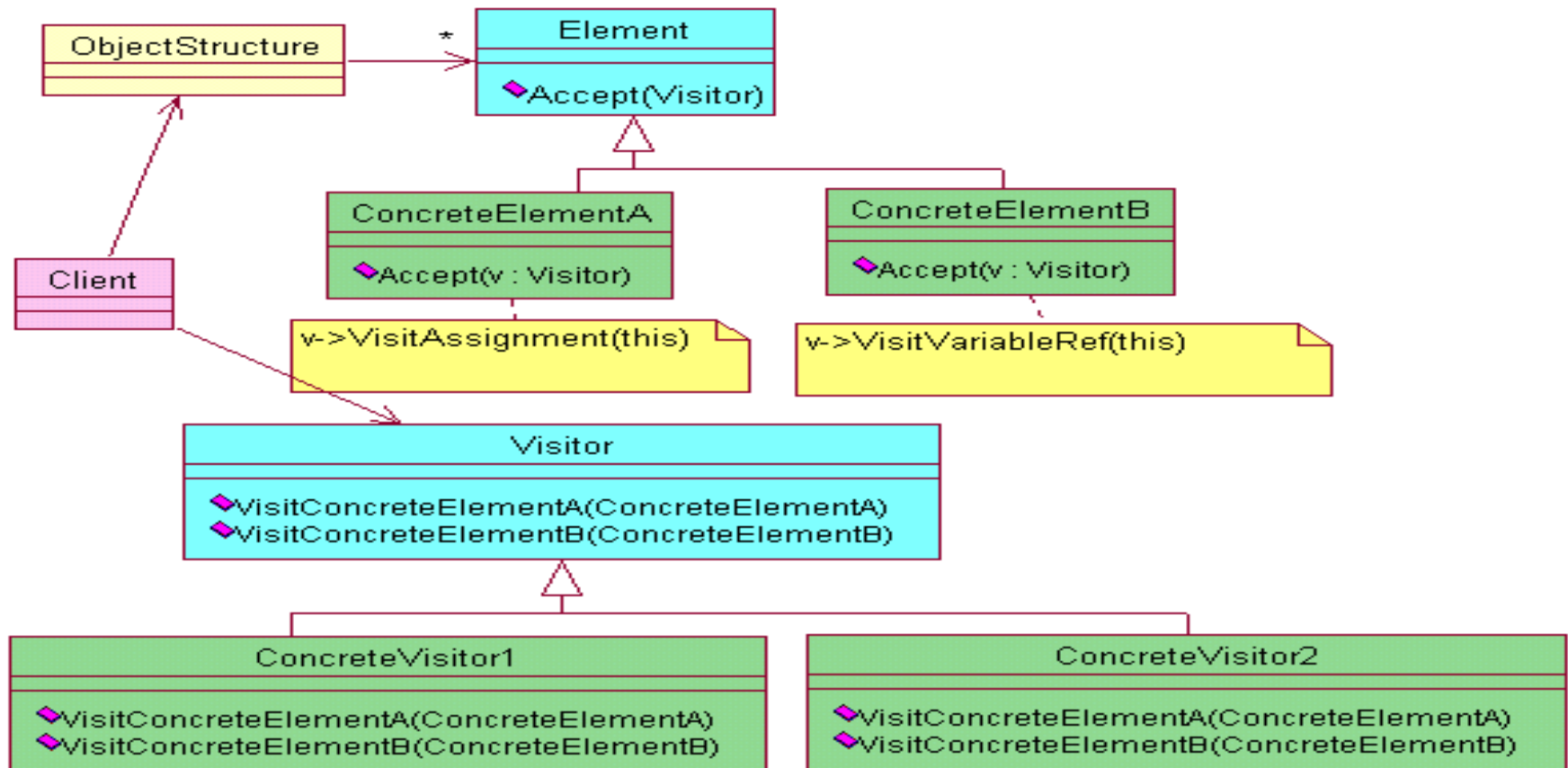
An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.

Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid “polluting” their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.

The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operation in those classes.



# Structure

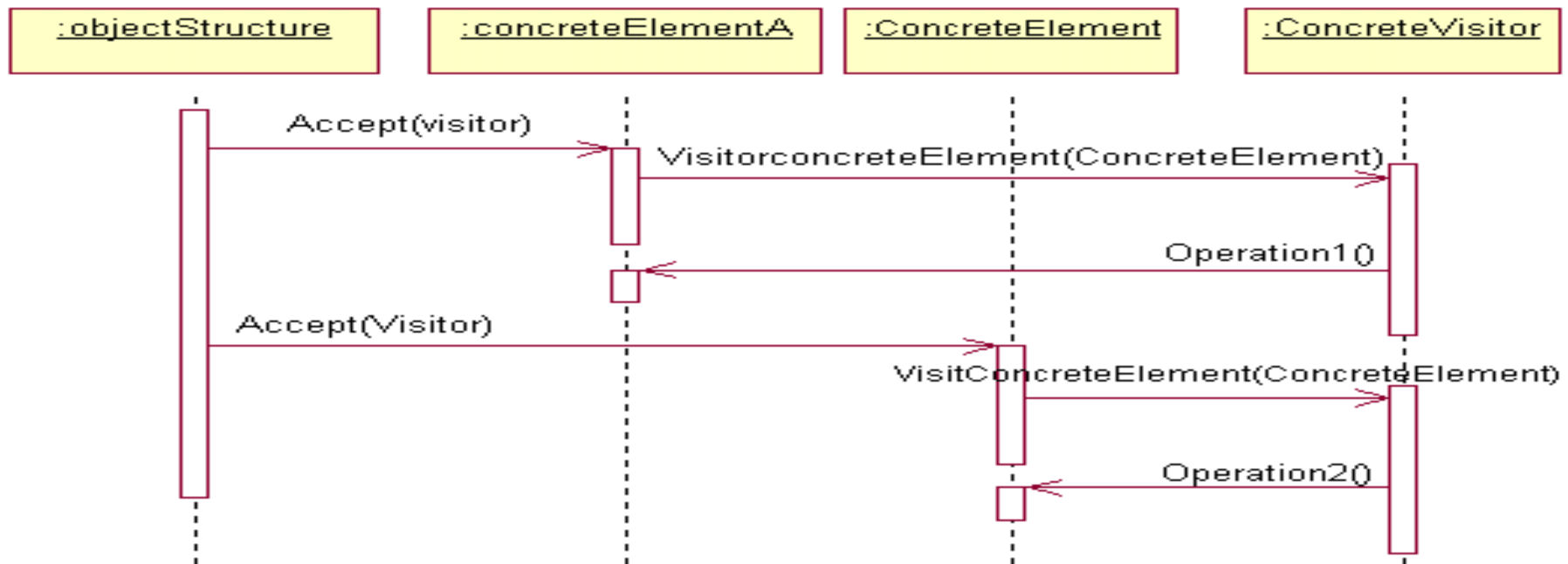


"The Visitor becomes more useful when there are several classes with different interfaces and we want to encapsulate how we get data from these classes."

# Collaborations

A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.

When an element is visited, it calls the visitor operation that corresponds to visitor access its state, if necessary.



## Consequences

### **1 Visitor makes adding new operations easy:**

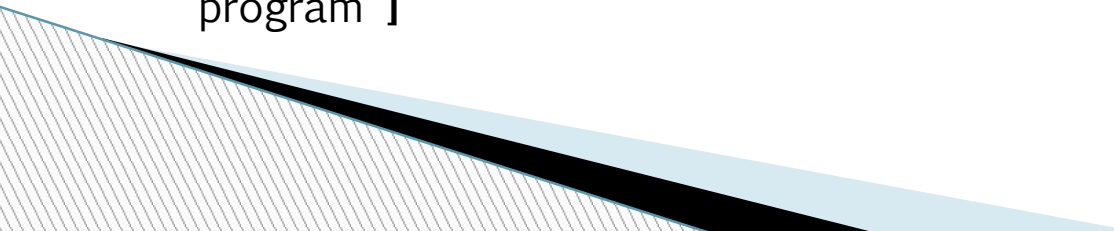
Add functions to class libraries for which you either do not have the source or cannot change the source

### **2. A visitor gathers related operations and separates unrelated ones:** Gather related operations into a single class rather than force you to change or derive classes to add these operations

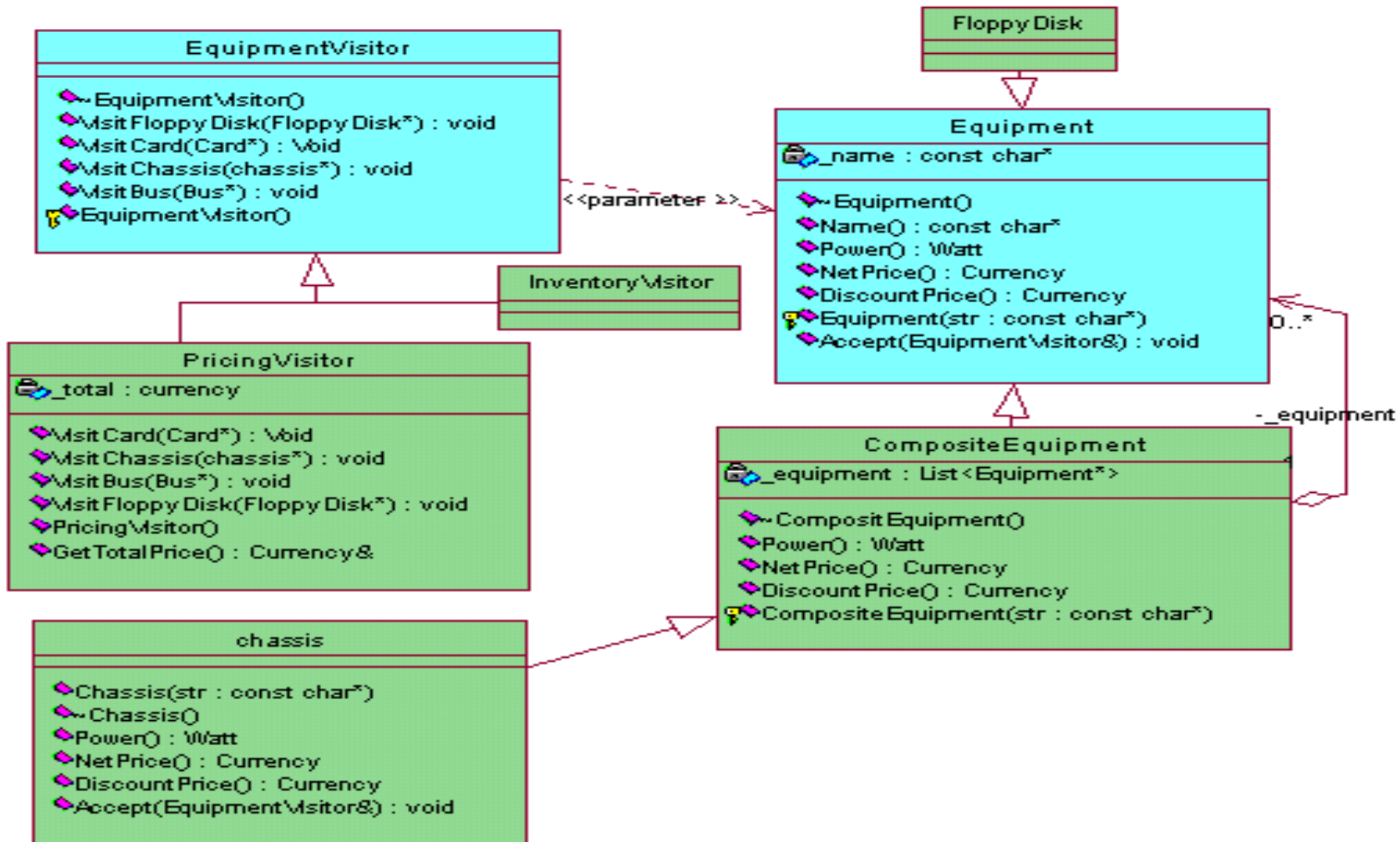
### **3. Adding new ConcreteElement classes is hard.**

### **4. Visiting across class hierarchies:**

**[NOTE:** Obtain data from a disparate collection of unrelated classes and use it to present the results of a global calculation to the user program ]



# Model



```
void Chassis::Accept (EquipmentVisitor& visitor) {  
    for (  
        ListIterator<Equipment*> i(_parts);  
        !i.IsDone();  
        i.Next()  
    ) {  
        i.CurrentItem()->Accept(visitor);  
    }  
    visitor.VisitChassis(this);  
}
```

---

```
void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {  
    _total += e->NetPrice();  
}
```

```
void PricingVisitor::VisitChassis (Chassis* e) {  
    _total += e->DiscountPrice();  
}
```

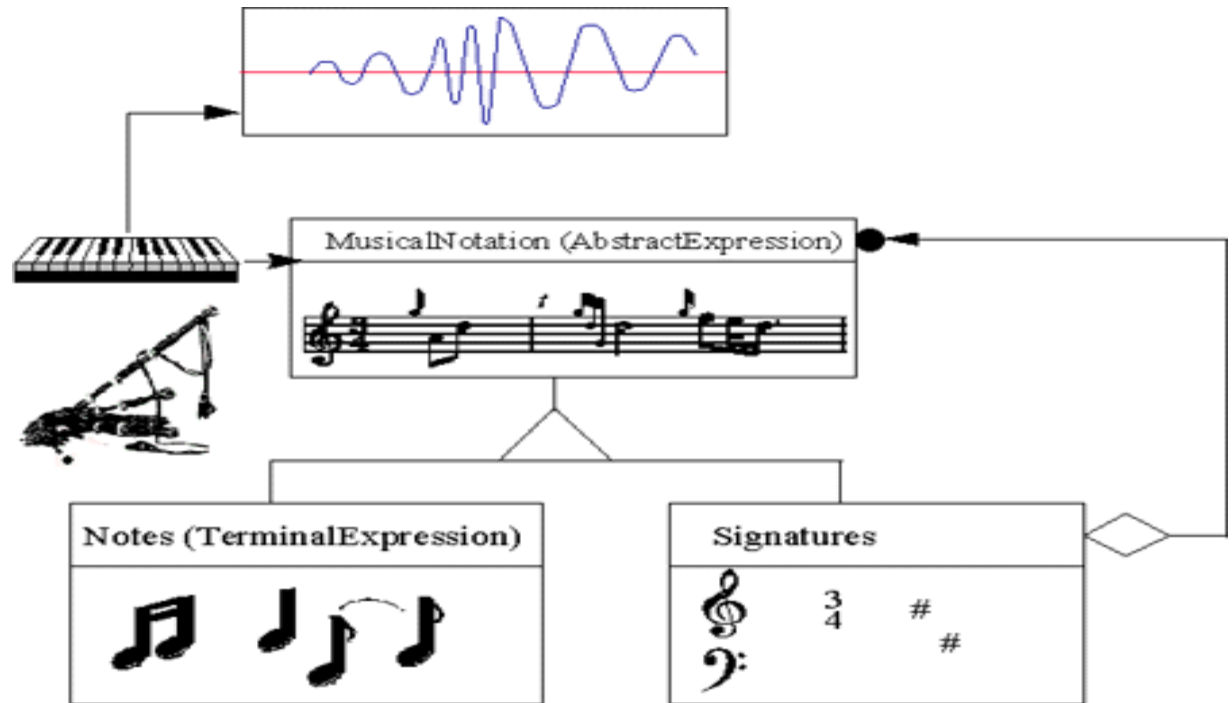




# Interpreter

# INTERPRETER

Object Diagram for  
*Interpreter* using Music  
Example



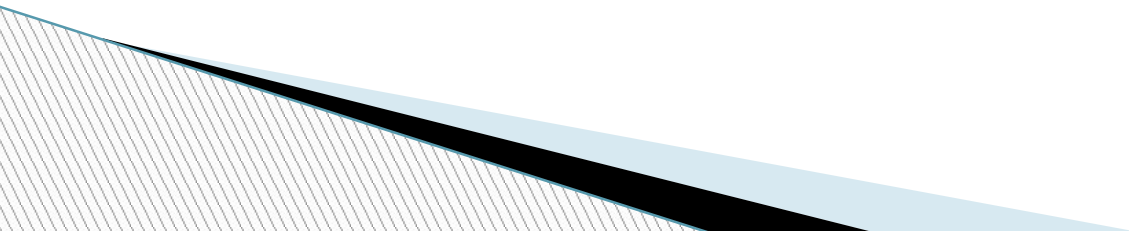
The *Interpreter* pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of *Interpreters*. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.

## **Intent**

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

## **Problem**

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



# Applicability

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees.

## **The Interpreter pattern works best when**

- **the grammar is simple.** For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.

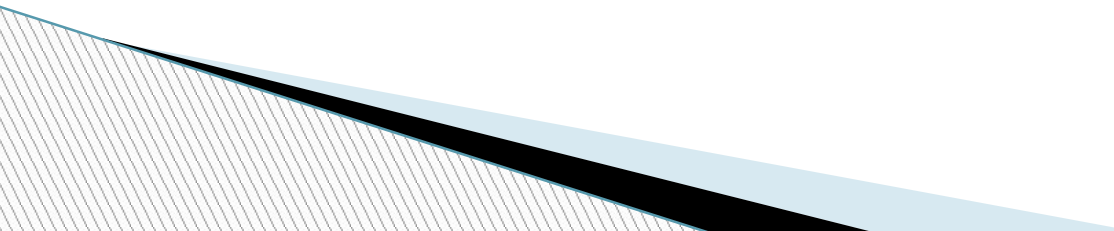
Applicability.....

**The Interpreter pattern works best when....**

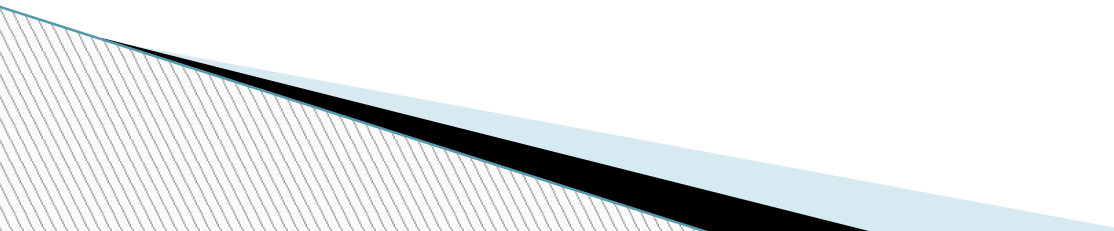
- **Efficiency is not a critical concern.**

The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines. But even then, the *translator can be* implemented by the Interpreter pattern, so the pattern is still applicable.

# Collaborations

- The client builds (or is given) the sentence as an abstract syntax tree of Non terminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
  - Each NonterminalExpmsion node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion.
  - The Interpret operations at each node use the context to store and access the state of the interpreter.
- 

## Consequences

1. *It's easy to change and extend the grammar.*
  2. *Implementing the grammar is easy, too.*
  3. *Complex grammars are hard to maintain.*
  4. *Adding new ways to interpret expressions.*
- 

**Thank You!**

A decorative graphic in the bottom-left corner consisting of several parallel diagonal lines in a light beige color, set against a dark blue background.