UNIT-V

Behavioral Patterns

Behavioural Patterns Part-I : Chain of Responsibility, Command, Interpreter, Iterator.

Behavioral Patterns (1)

• Deal with the way objects interact and distribute responsibility.

• Chain of Responsibility: 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.

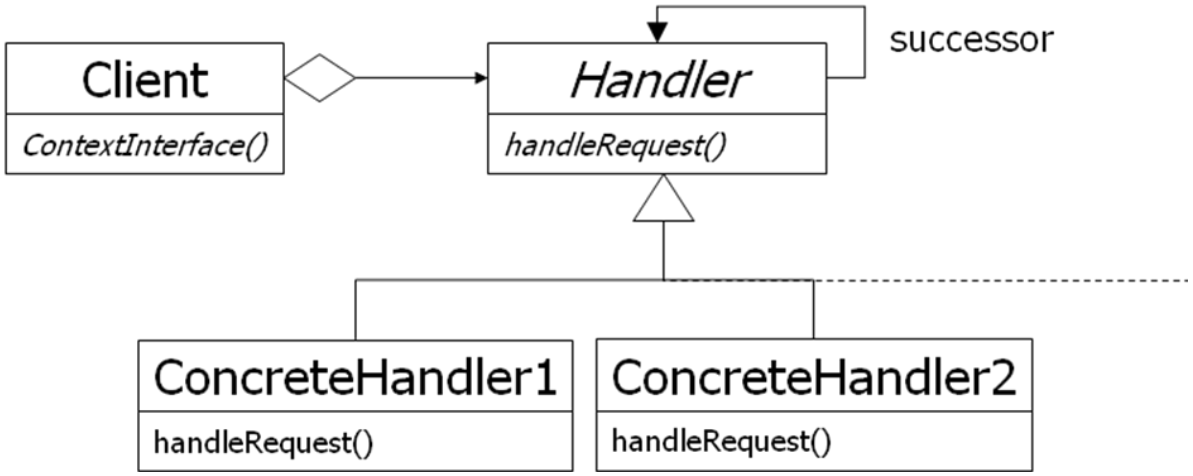
objects an dpass the request along the chain until an object handles it.

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

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

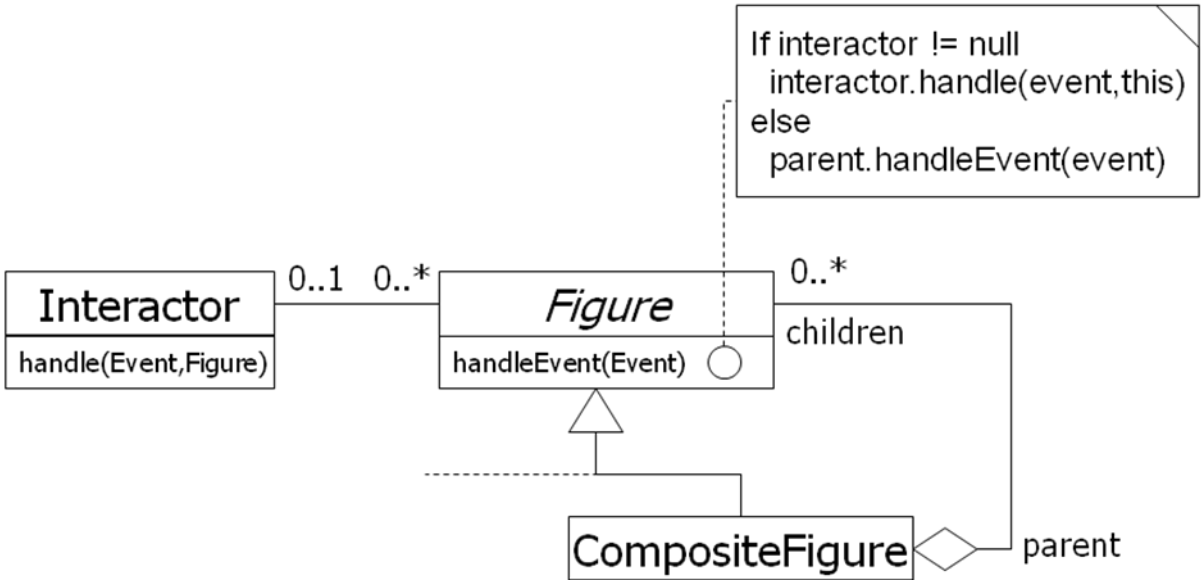
Chain of Responsibility:

- Decouple sender of a request from receiver.
- Give more than one object a chance to handle.
- Flexibility in assigning responsibility.
- Often applied with Composite.



Chain of Responsibility (2)

- Example: handling events in a graphical hierarchy



Command:Encapsulating Control Flow :

Name: Command design pattern

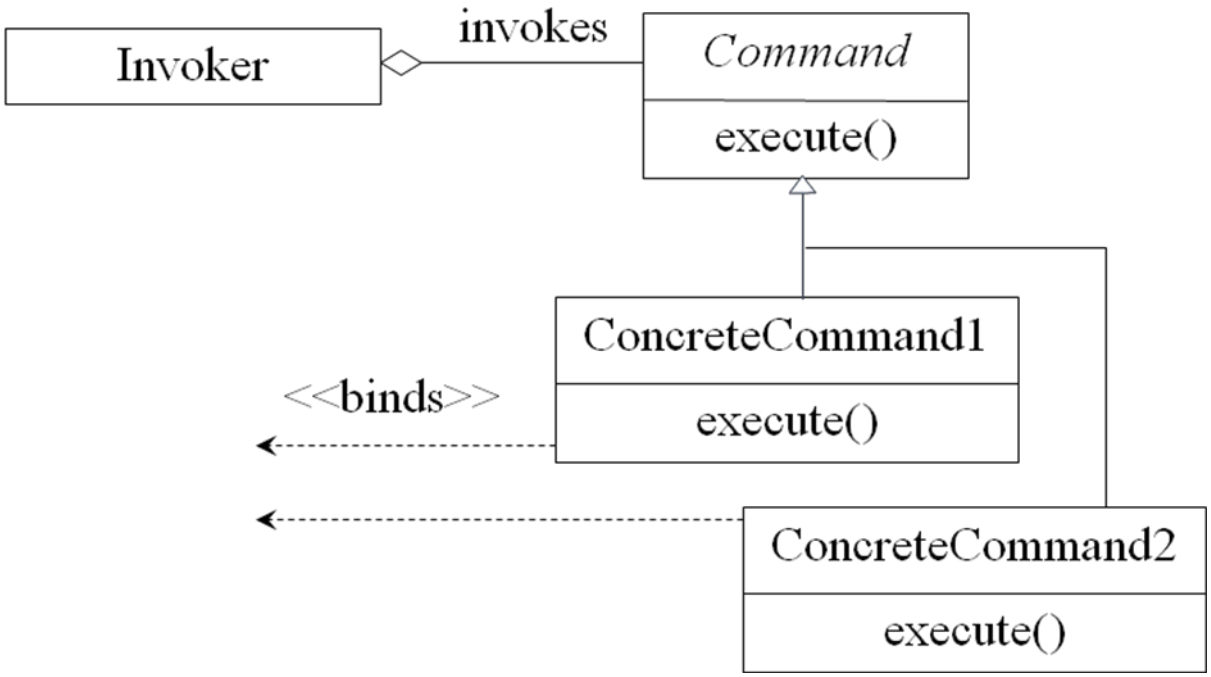
Problem description:

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

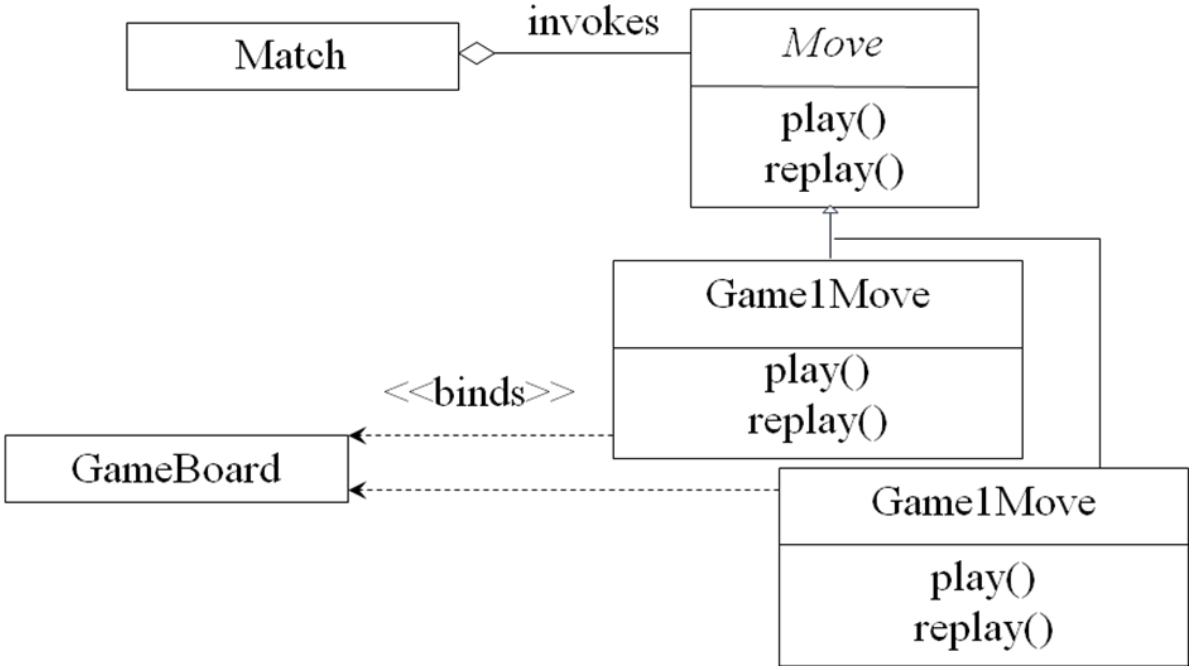
Solution:

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

Command: Class Diagram



Command: Class Diagram for Match



Command: Consequences

Consequences:

The object of the command (Receiver) and the algorithm of the command. (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

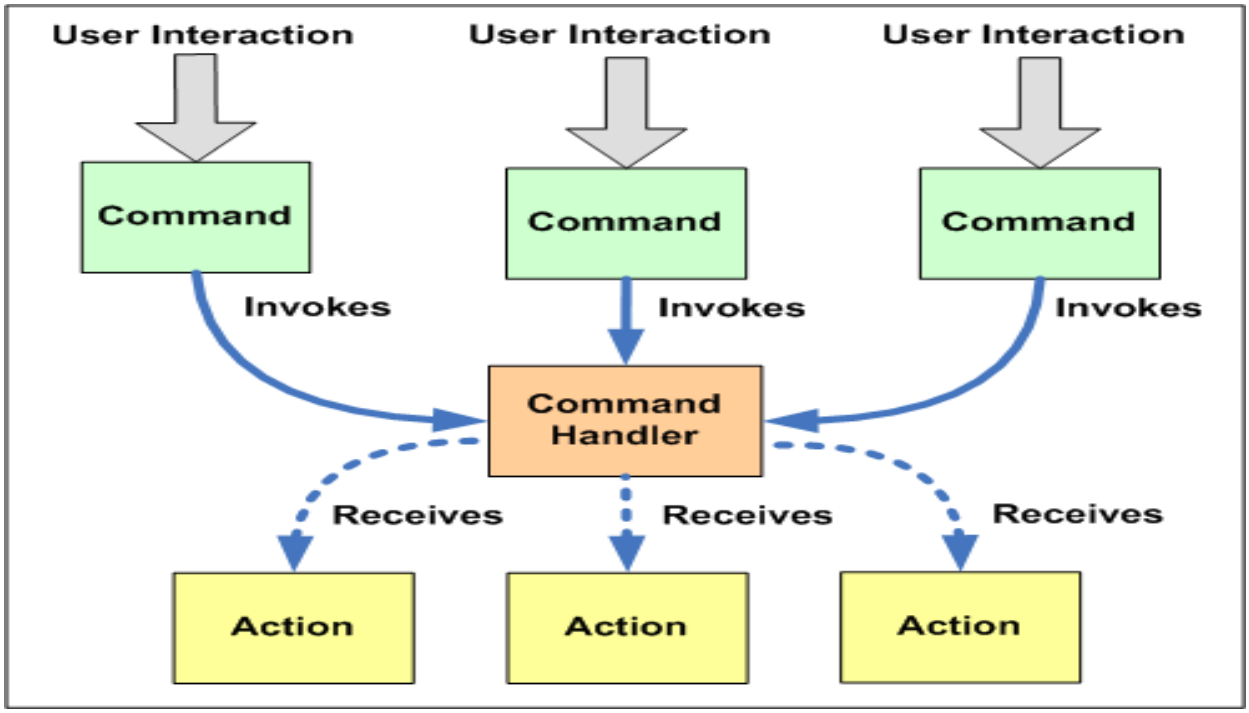
ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

Command:

- You have commands that need to be
  - executed,
  - undone, or
  - queued
- Command design pattern separates
  - Receiver from Invoker from Commands
- All commands derive from Command and implement do(), undo(), and redo().

Command Design Pattern :



- Separates command invoker and receiver.

Pattern: Interpreter:

- Intent: Given a language, interpret sentences.
- Participants: Expressions, Context, Client.
- Implementation: A class for each expression type

  An Interpret method on each class
  A class and object to store the global state (context)
- No support for the parsing process
  (Assumes strings have been parsed into exp trees)

Pattern: Interpreter with Macros:

- Example: Definite Clause Grammars.
- A language for writing parsers/interpreters.
- Macros make it look like (almost) standard BNF.
  Command(move(D)) -> “go”, Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp.
- Does parsing as well as interpretation.

- Builds tree structure only as needed.
  (Or, can automatically build complete trees)
- May or may not use expression classes.

Method Combination:

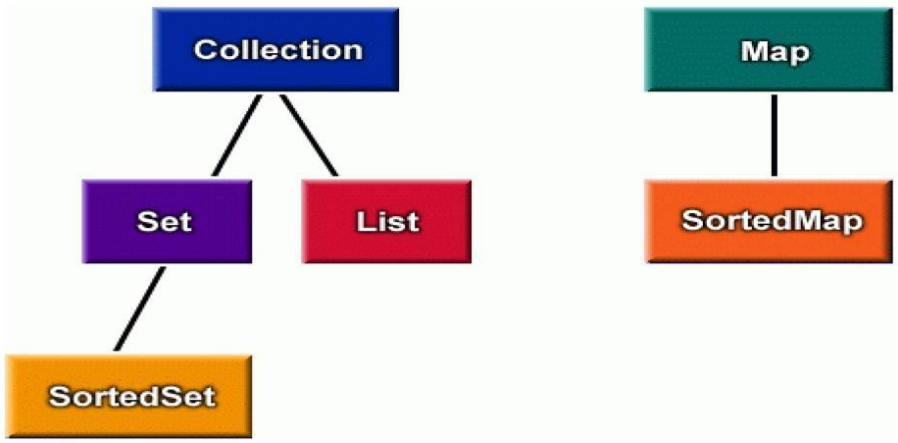
- Build a method from components in different classes
- Primary methods: the “normal” methods; choose the most specific one
- Before/After methods: guaranteed to run;
  No possibility of forgetting to call super
  Can be used to implement *Active Value* pattern
- Around methods: wrap around everything;
  Used to add tracing information, etc.
- Is added complexity worth it?
  Common Lisp: Yes; Most languages: No

Iterator pattern :

- **iterator**: an object that provides a standard way to examine all elements of any collection.
- uniform interface for traversing many different data structures without exposing their implementations.
- supports concurrent iteration and element removal.
- removes need to know about internal structure of collection or different methods to access data from different collections.

Pattern: Iterator

objects that traverse collections



Iterator interfaces in Java:

```
public interface java.util.Iterator {
public boolean hasNext();
  public Object next();
  public void remove();
}
public interface java.util.Collection {
  ... // List, Set extend Collection
  public Iterator iterator();
}
public interface java.util.Map {
  ...
  public Set keySet();        // keys,values are Collections
  public Collection values(); // (can call iterator() on them)
}
```

Iterators in Java:

- all Java collections have a method iterator that returns an iterator for the elements of the collection.
- can be used to look through the elements of any kind of collection (an alternative to for loop).

```
List list = new ArrayList();
... add some elements ...
for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

Adding your own Iterators :

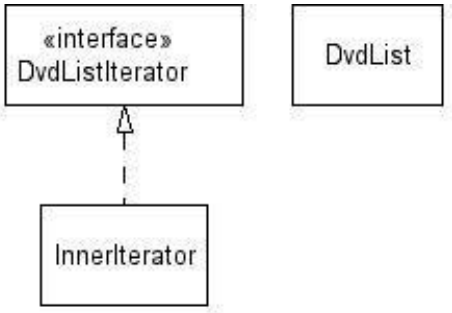
- when implementing your own collections, it can be very convenient to use Iterators

–**discouraged (has nonstandard interface):**

```
public class PlayerList {
  public int getNumPlayers() { ... }
  public boolean empty() { ... }
  public Player getPlayer(int n) { ... }
}
```

– **preferred:**

```
public class PlayerList {
  public Iterator iterator() { ... }
  public int size() { ... }
  public boolean isEmpty() { ... }
}
```



Command:Encapsulating Control Flow:

Name: Command design pattern

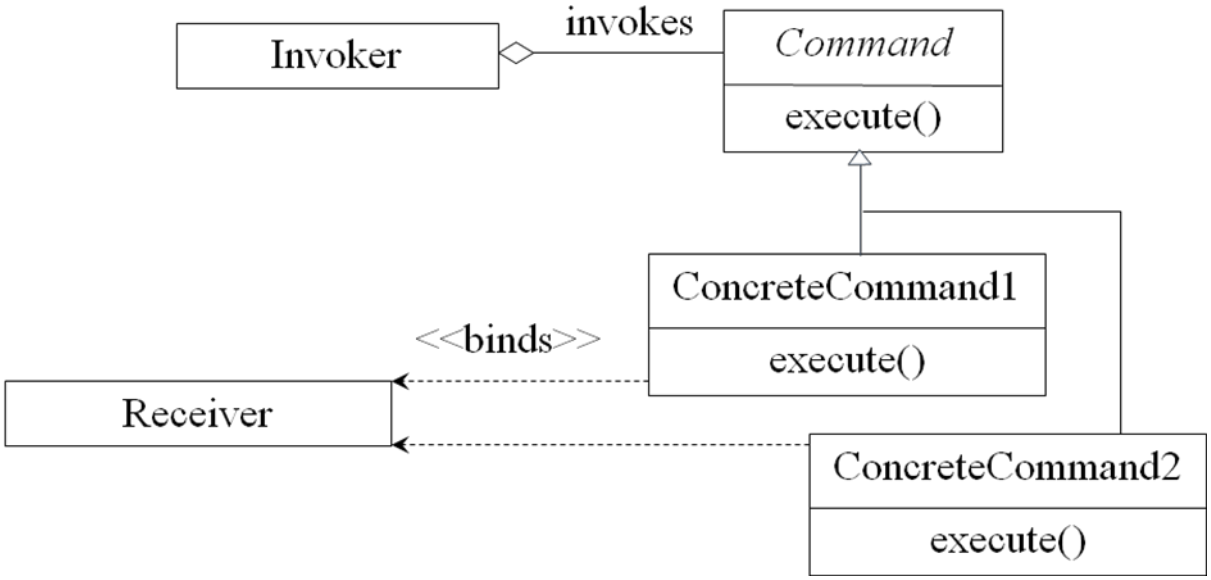
Problem description:

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

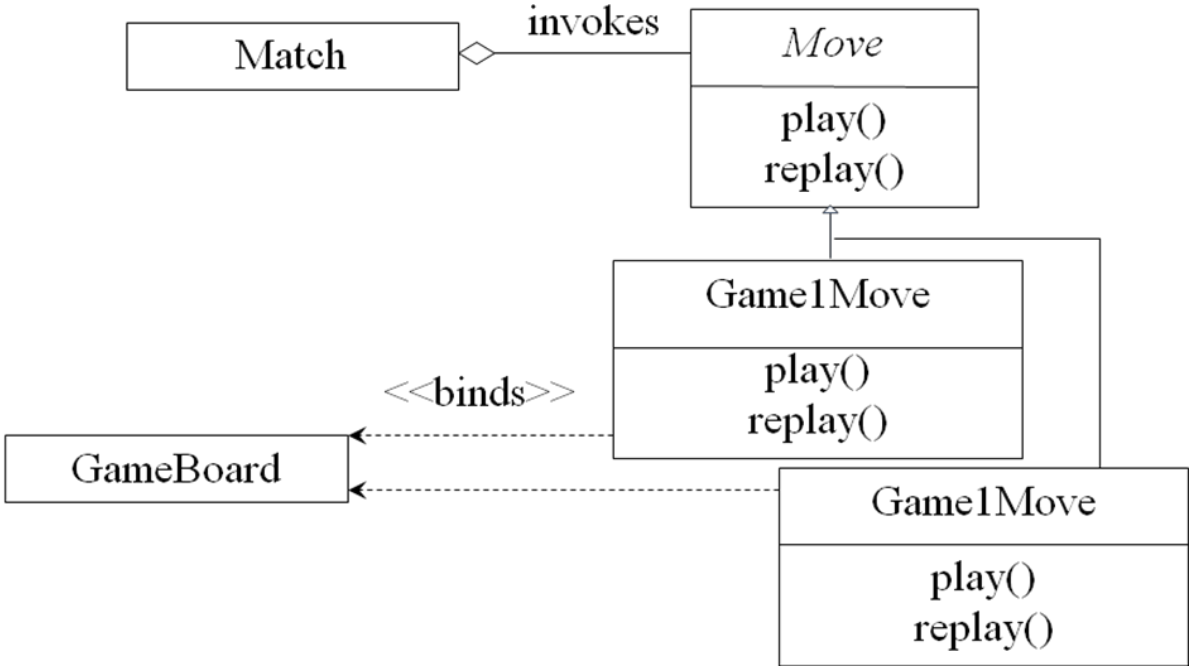
Solution:

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

Command: Class Diagram



Command: Class Diagram for Match



Command: Consequences

Consequences:

The object of the command (Receiver) and the algorithm of the command (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

- **Intent**: Given a language, interpret sentences
- **Participants**: Expressions, Context, Client
- **Implementation**: A class for each expression type
  An Interpret method on each class
  A class and object to store the global state (context)
- No support for the parsing process
  (Assumes strings have been parsed into exp trees)

Pattern: Interpreter with Macros

- **Example**: Definite Clause Grammars
- A language for writing parsers/interpreters
- Macros make it look like (almost) standard BNF
  Command(move(D)) -> “go”, Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp
- Does parsing as well as interpretation.
- Builds tree structure only as needed.
  (Or, can automatically build complete trees)
- May or may not use expression classes.

Method Combination:

- Build a method from components in different classes
- Primary methods: the “normal” methods; choose the most specific one
- Before/After methods: guaranteed to run;
  No possibility of forgetting to call super
  Can be used to implement *Active Value* pattern
- Around methods: wrap around everything;
  Used to add tracing information, etc.
- Is added complexity worth it?
  Common Lisp: Yes; Most languages: No

Behavioural Patterns Part-II

Part-II : Mediator, Memento, Observer

Behavioral Patterns (1):

• Deal with the way objects interact and distribute responsibility

• Chain of Responsibility: 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 an dpass the request along the chain until an object handles it.

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

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

Behavioral Patterns (2):

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

• Mediator: 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 lets you vary their interaction independently.

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

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

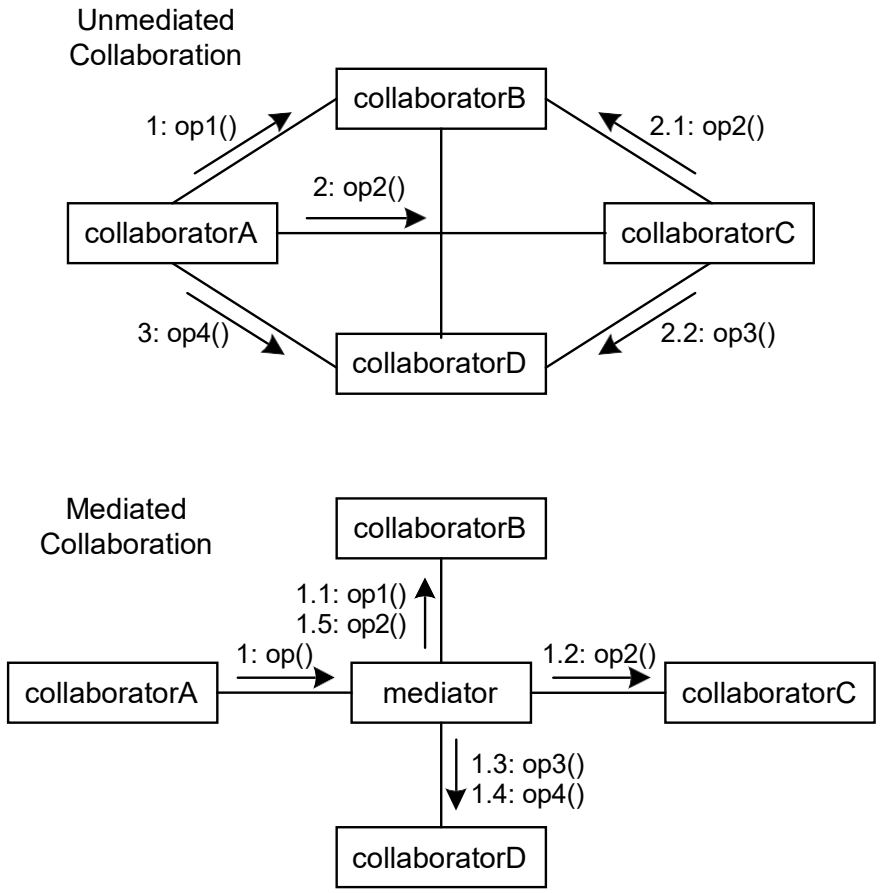
Behavioral Patterns (3)

- State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasseses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor: 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.

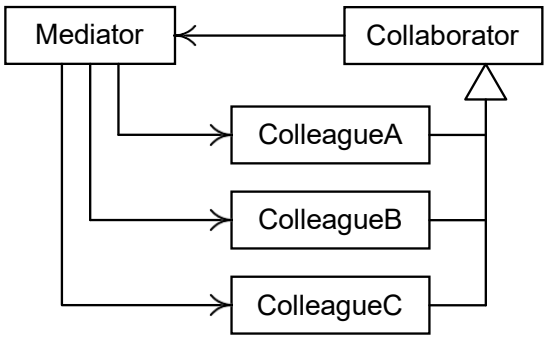
The Mediator Pattern:

- The Mediator pattern reduces coupling and simplifies code when several objects must negotiate a complex interaction.
- Classes interact only with a mediator class rather than with each other.
- Classes are coupled only to the mediator where interaction control code resides.
- Mediator is like a multi-way Façade pattern.

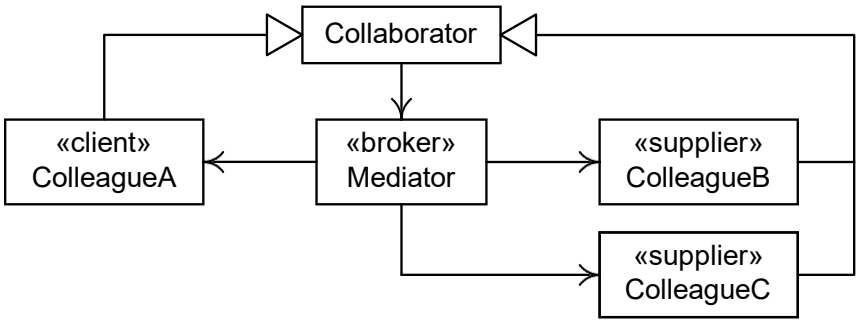
Analogy: a meeting scheduler.



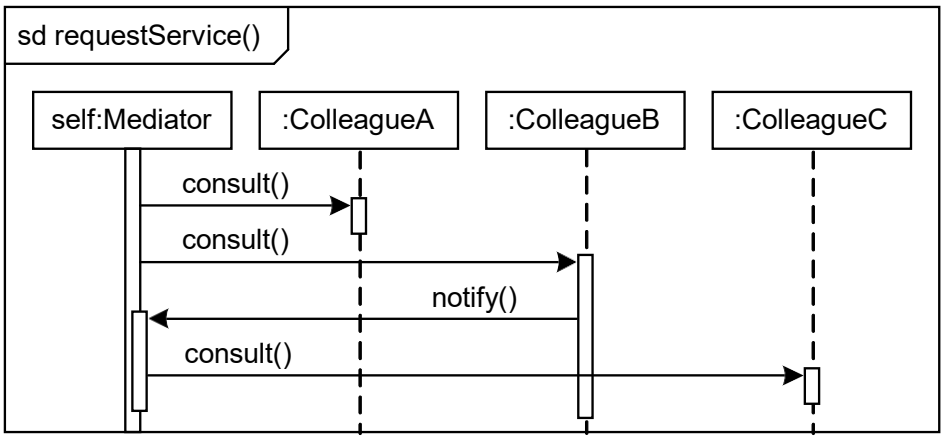
Mediator Pattern Structure:



Mediator as a Broker:



Mediator Behavior:



When to Use a Mediator:

- Use the Mediator pattern when a complex interaction between collaborators must be encapsulated to
  - Decouple collaborators,
  - Centralize control of an interaction, and
  - Simplify the collaborators.
- Using a mediator may compromise performance.

Mediators, Façades, and Control Styles:

- The Façade and Mediator patterns provide means to make control more centralized.
- The Façade and Mediator patterns should be used to move from a dispersed to a delegated style, but not from a delegated to a centralized style.

Summary :

- Broker patterns use a Broker class to facilitate the interaction between a Client and a Supplier.
- The Façade pattern uses a broker (the façade) to provide a simplified interface to a complex sub-system.
- The Mediator pattern uses a broker to encapsulate and control a complex interaction among several suppliers.

Memento Pattern

Intent:

- Capture and externalize an object's state without violating encapsulation.
- Restore the object's state at some later time.
  - Useful when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
  - Entrusts other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

Forces:

- Application needs to capture states at certain times or at user discretion. May be used for:
  - Undue / redo
  - Log errors or events
  - Backtracking
- Need to preserve encapsulation
  - Don't share knowledge of state with other objects
- Object owning state may not know when to take state snapshot.

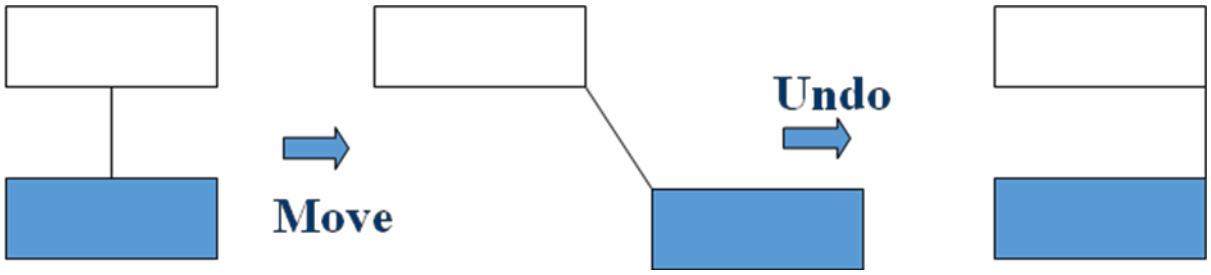
Motivation:

- Many technical processes involve the exploration of some complex data structure.
- Often we need to backtrack when a particular path proves unproductive.

Examples are graph algorithms, searching knowledge bases, and text navigation.

Memento stores a snapshot of another object's internal state, exposure of which would violate encapsulation and compromise the application's reliability and extensibility.

A graphical editor may encapsulate the connectivity relationships between objects in a class, whose public interface might be insufficient to allow precise reversal of a move operation.



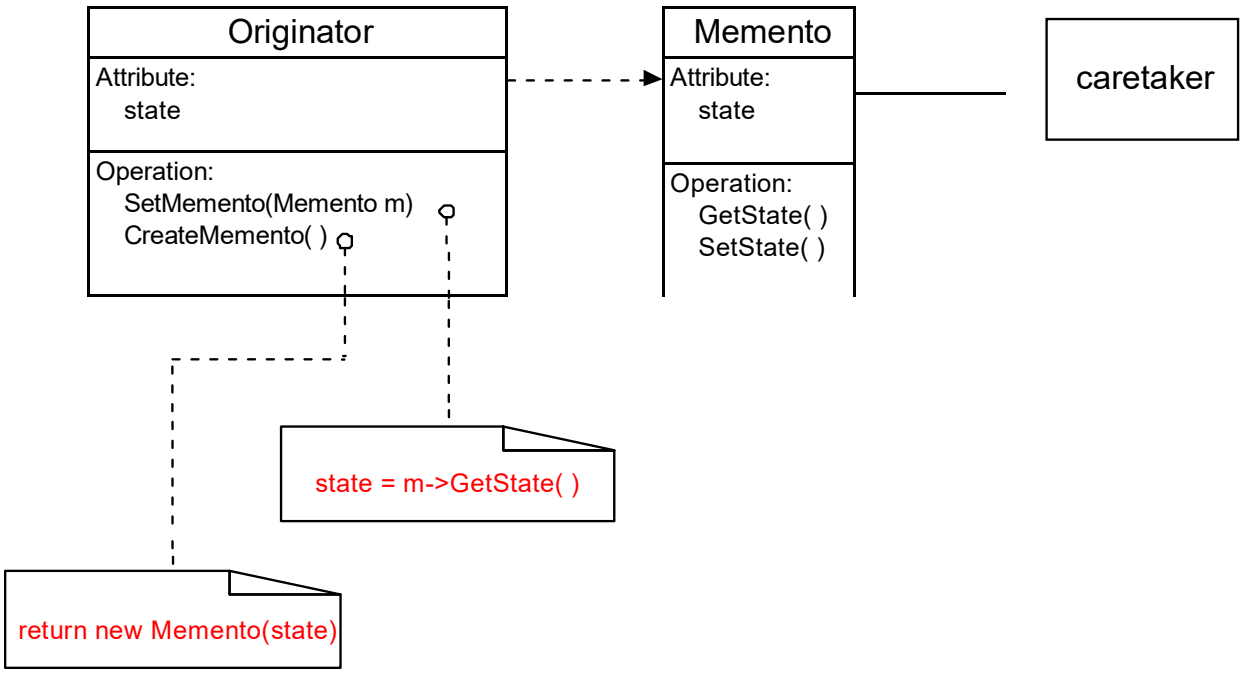
Memento pattern solves this problem as follows:

- The editor requests a memento from the object before executing *move* operation.
- Originator creates and returns a memento.
- During *undo* operation, the editor gives the memento back to the originator.
- Based on the information in the memento, the originator restores itself to its previous state.

Applicability:

- Use the Memento pattern when:
  - A snapshot of an object's state must be saved so that it can be restored later, and direct access to the state would expose implementation details and break encapsulation.

Structure:



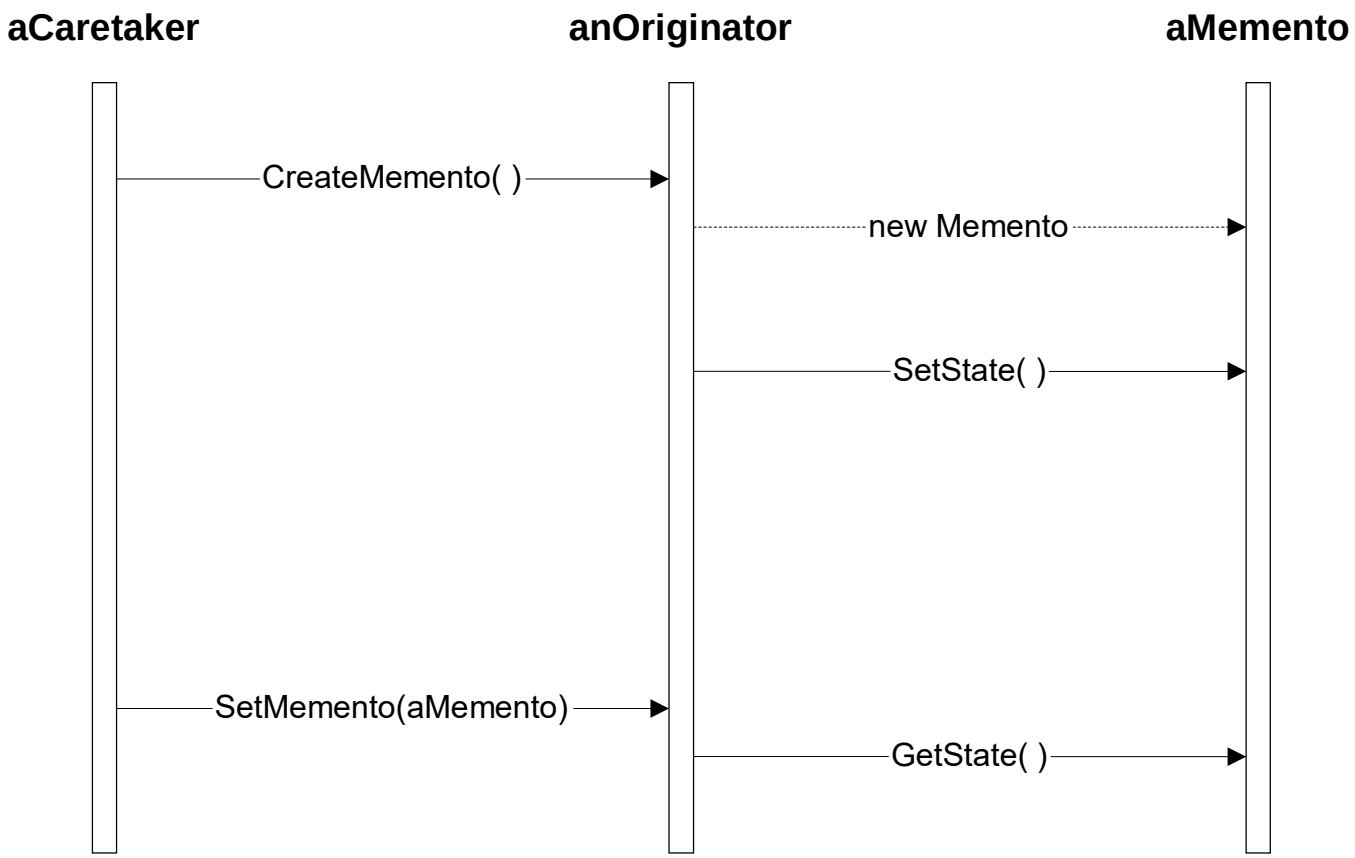
Participants:

- Memento
  - Stores internal state of the Originator object. Originator decides how much.
  - Protects against access by objects other than the originator.
  - Mementos have two interfaces:
    - Caretaker sees a narrow interface.
    - Originator sees a wide interface.
- Originator
  - Creates a memento containing a snapshot of its current internal state.
  - Uses the memento to restore its internal state.

Caretaker:

- Is responsible for the memento's safekeeping.
- Never operates on or examines the contents of a memento.

Event Trace:



Collaborations:

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

Consequences:

Memento has several consequences:

- Memento avoids exposing information that only an originator should manage, but for simplicity should be stored outside the originator.
- Having clients manage the state they ask for simplifies the originator.

- Using mementos may be expensive, due to copying of large amounts of state or frequent creation of mementos.
- A caretaker is responsible for deleting the mementos it cares for.

A caretaker may incur large storage costs when it stores mementos.

Implementation:

When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just incremental changes to originator's state.

Known Uses:

Memento is a 2000 film about Leonard Shelby and his quest to revenge the brutal murder of his wife. Though Leonard is hampered with short-term memory loss, he uses notes and tatoos to compile the information into a suspect.

Known Use of Pattern

- Dylan language uses memento to provide iterators for its collection facility.
  - Dylan is a dynamic object oriented language using the functional style.
  - Development started by Apple, but subsequently moved to open source.

Related Patterns

- Command

  Commands can use mementos to maintain state for undo mechanisms.

- Iterator

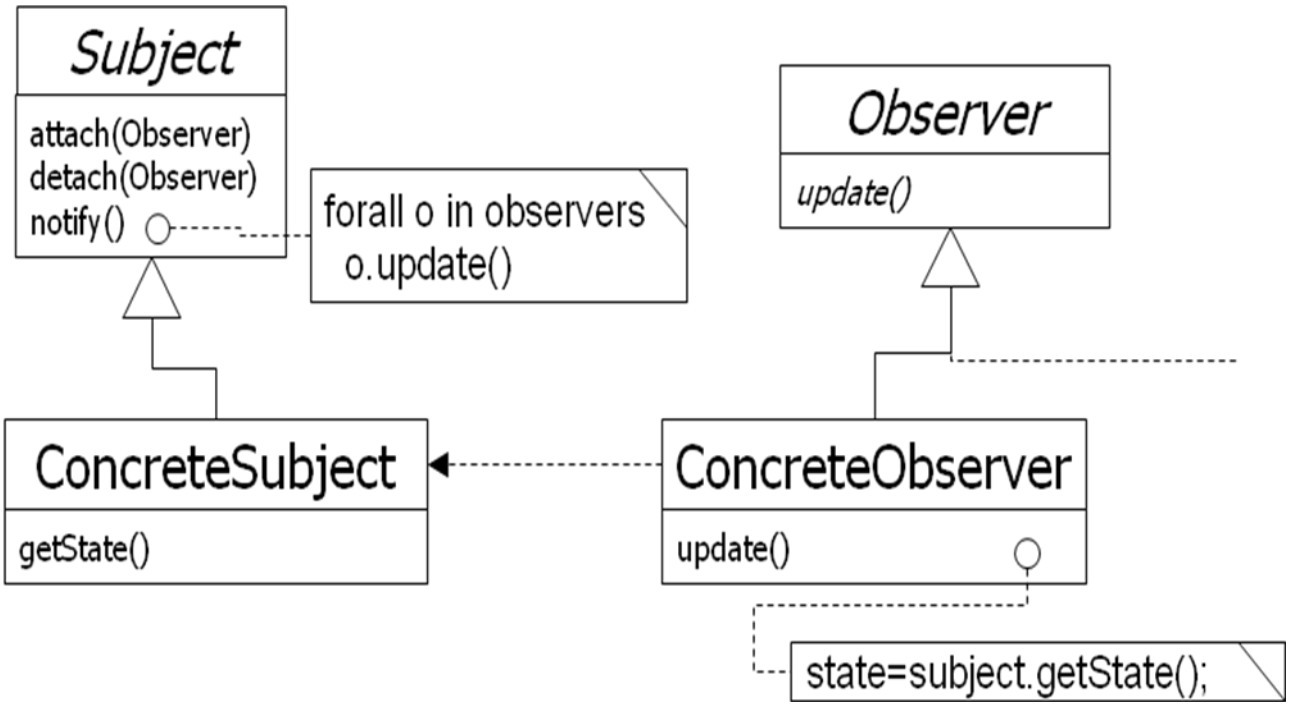
  Mementos can be used for iteration.

Observer Pattern

- Define a one-to-many dependency, all the dependents are notified and updated automatically
- The interaction is known as **publish-subscribe** or **subscribe-notify**
- Avoiding observer-specific update protocol: **pull model** vs. **push model**
- Other consequences and open issues
- **Intent:**
    - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Key forces:**
    - There may be many observers
    - Each observer may react differently to the same notification
    - The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject

Observer

- Many-to-one dependency between objects
- Use when there are two or more views on the same "data"
- aka "Publish and subscribe" mechanism
- Choice of "push" or "pull" notification styles



Observer: Encapsulating Control Flow

Name: Observer design pattern

Problem description:

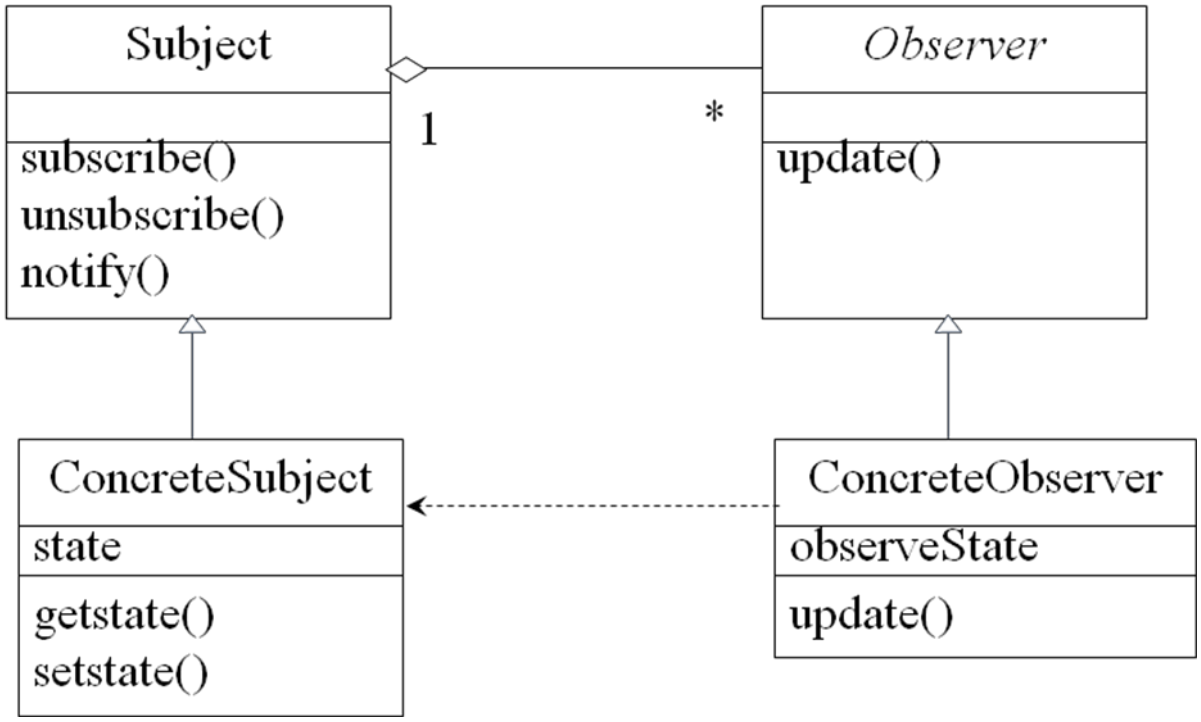
Maintains consistency across state of one Subject and many Observers.

Solution:

A Subject has a primary function to maintain some state (e.g., a data structure). One or more Observers use this state, which introduces redundancy between the states of Subject and Observer.

Observer invokes the subscribe() method to synchronize the state. Whenever the state changes, Subject invokes its notify() method to iteratively invoke each Observer.update() method.

Observer: Class Diagram



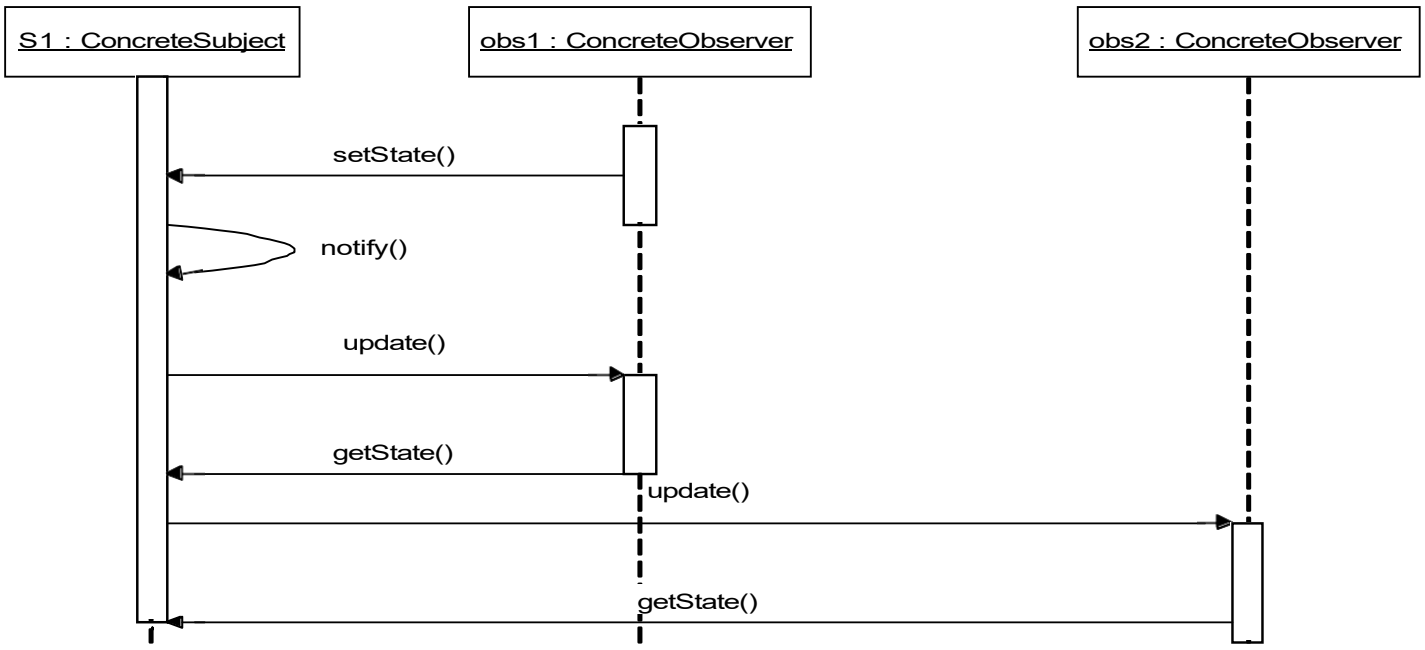
Observer: Consequences

Consequences:

Decouples Subject, which maintains state, from Observers, who make use of the state.

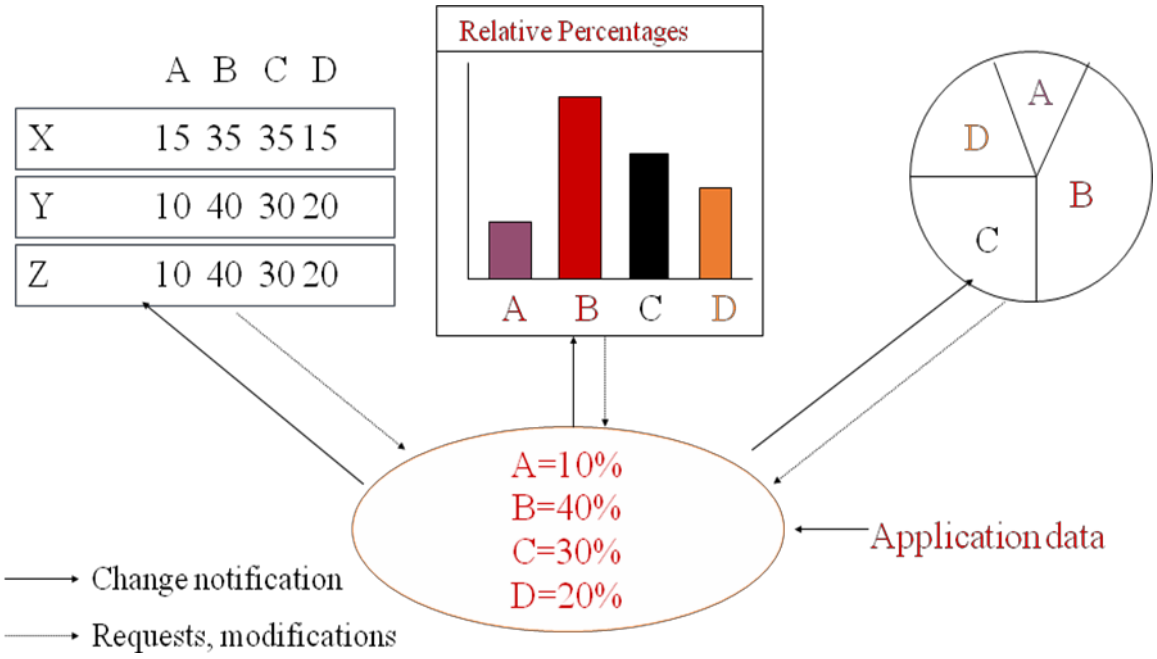
Can result in many spurious broadcasts when the state of Subject changes.

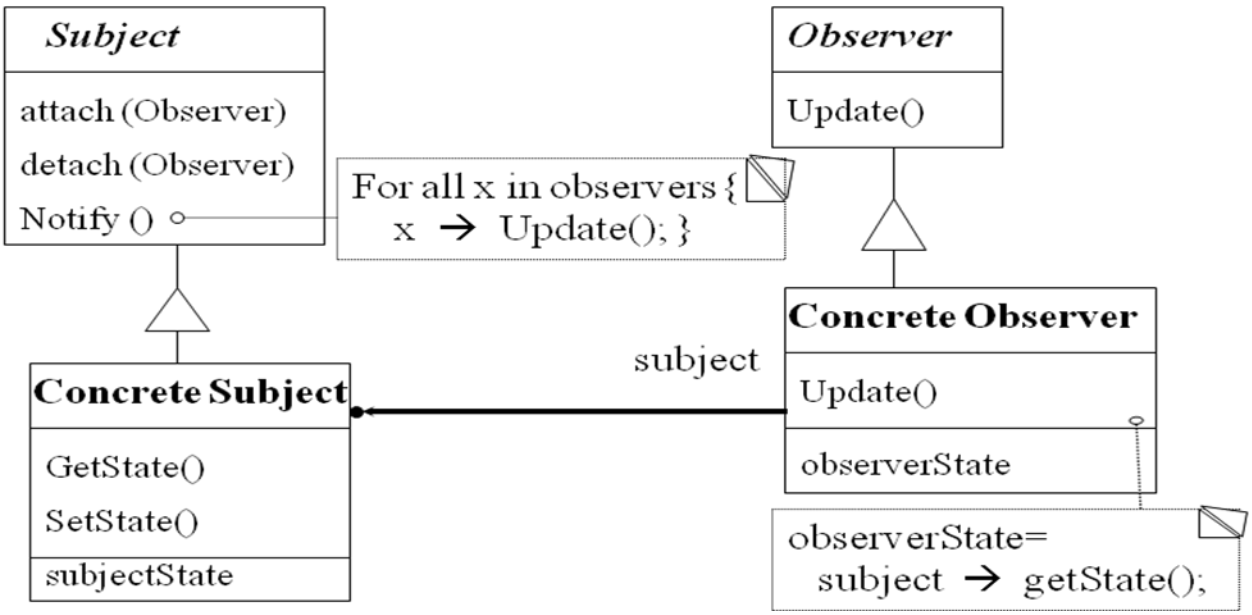
Collaborations in Observer Pattern



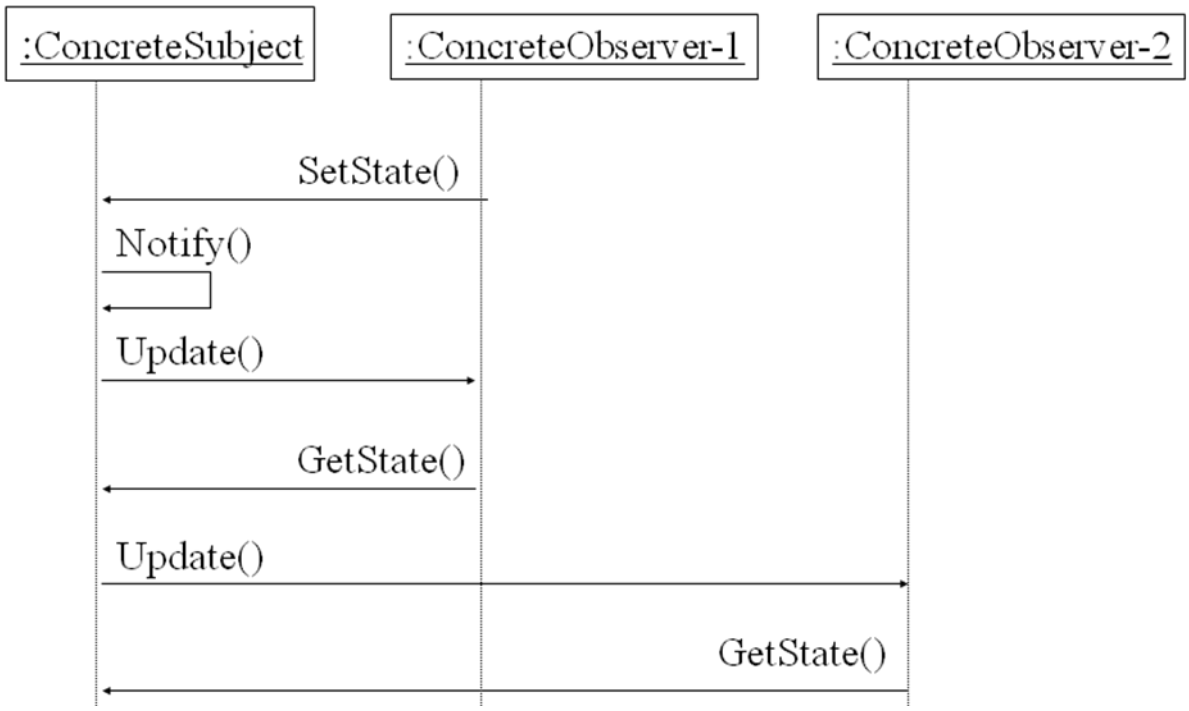
Observer Pattern [1]

- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.





Class collaboration in Observer



Observer Pattern: Observer code

```
class Subject;

class observer {
public:
        virtual ~observer;

        virtual void Update (Subject* theChangedSubject)=0;

protected:

        observer ();
};
```

Abstract class defining the Observer interface.

Note the support for multiple subjects.

Observer Pattern: Subject Code

```
class Subject {

public:
        virtual ~Subject;
        virtual   void Attach (observer*);
        virtual   void Detach (observer*) ;
        virtual   void Notify();
protected:
        Subject ();
private:
        List <Observer*> *_observers;
};


void Subject :: Attach (Observer* o){
      _observers -> Append(o);
}
void Subject :: Detach (Observer* o){
      _observers -> Remove(o);
}
void Subject :: Notify (){

                  iter.CurrentItem() -> Update(this);
      }
}
```

Abstract class defining the Subject interface.

Observer Pattern: A Concrete Subject [1]

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinutes();
        virtual int GetSecond();
        void Tick ();
}

ClockTimer :: Tick {
    // Update internal time keeping state.
    // gets called on regular intervals by an internal
       timer.
        Notify();
 }
```

Observer Pattern: A Concrete Observer [1]

```
class DigitalClock: public Widget, public
    Observer {
public:
        DigitalClock(ClockTimer*);
        virtual ~DigitalClock();
        virtual void  Update(Subject*);
        virtual void  Draw();
private:
        ClockTimer* _subject;
        }
```

Override Observer operation.

Override Widget operation.

```
DigitalClock ::DigitalClock (ClockTimer* s) {
            _subject = s;
            _subject→Attach(this);
}

DigitalClock ::~DigitalClock() {
            _subject->Detach(this);

}
```

```
void DigitalClock ::Update (subject* theChangedSubject ) {
    If (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock ::Draw () {
  int hour = _subject->GetHour();
  int minute = _subject->GeMinute(); // etc.
  // Code for drawing the digital clock.
}
```

Check if this is the clock's subject.

Observer Pattern: Main (skeleton)

```
ClockTimer* timer = new ClockTimer;

DigitalClock* digitalClock = new DigitalClock (timer);
```

Observer Pattern: Consequences

- Abstract coupling between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.
- Unexpected updates: Observers need not be concerned about when then updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

When to use the Observer Pattern?

- *When* an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.
- *When* a change to one object requires changing others and the number of objects to be changed is not known.
- When an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

UNIT-V

Behavioral Patterns

Behavioural Patterns Part-II(cont'd) : State, Strategy, Template Method, Visitor, Discussion of Behavioural Patterns.

General Description

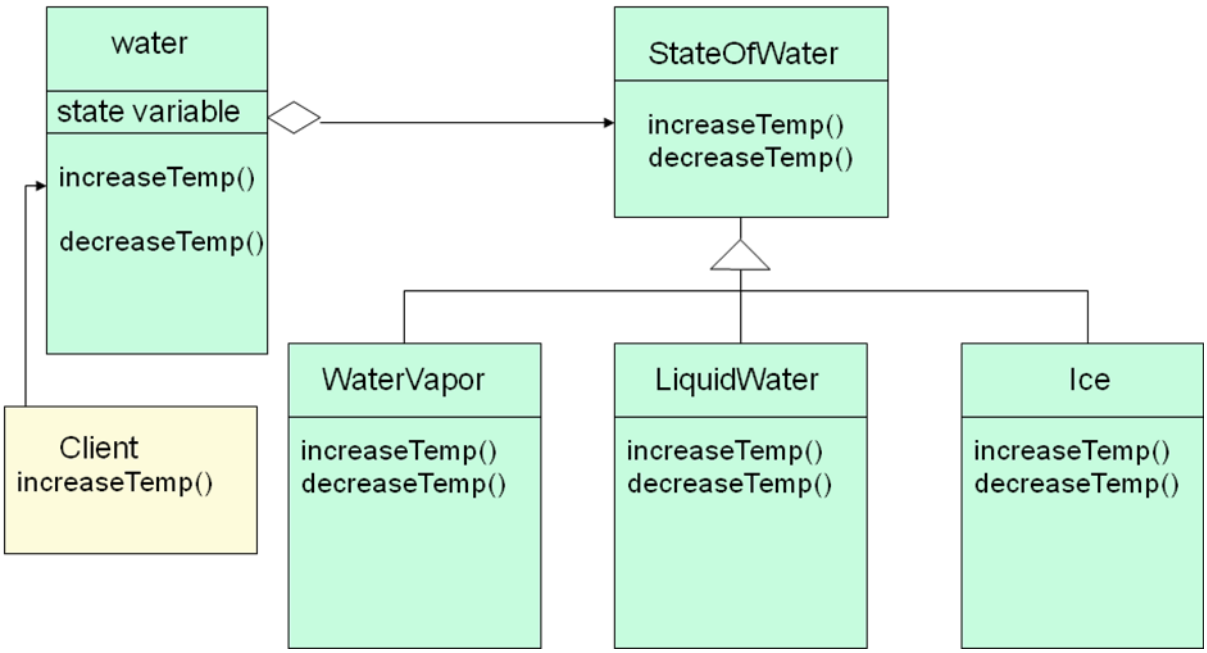
- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

```
if (myself = happy) then
{
                    eatIceCream();
                    ....
                    }
else if (myself = sad) then
{
                    goToPub();
                    ....
  }
else if (myself = ecstatic) then
{
                    ....
```

Example I



How is STATE pattern implemented ?

- "Context" class:

  Represents the interface to the outside world.

- "State" abstract class:

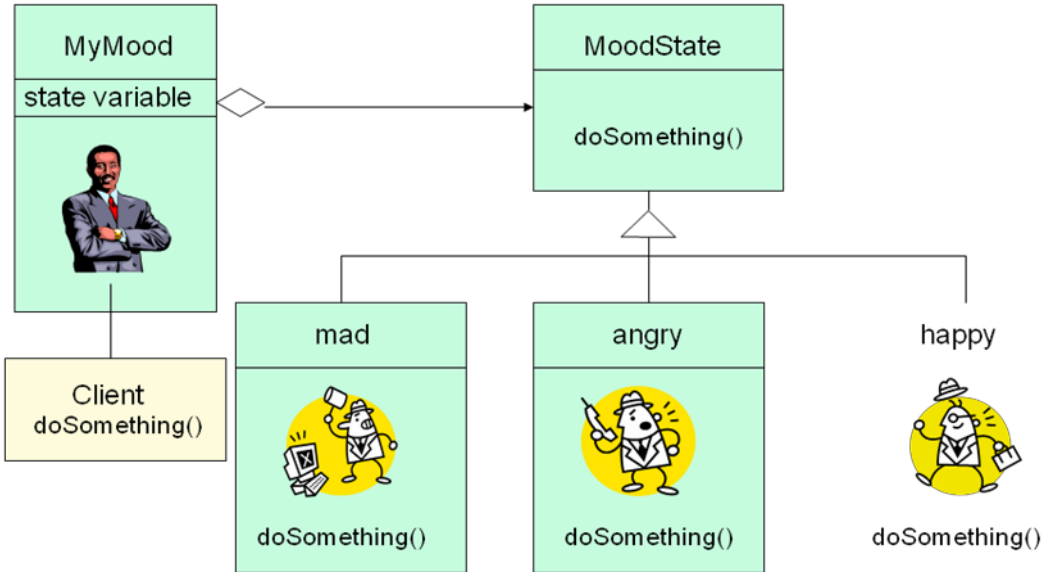
  Base class which defines the different states of the "state machine".

- "Derived" classes from the State class:

  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

Example II



Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.
- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.
  - **State objects can be shared.**
  - Context can share State objects if there are no instance variables.

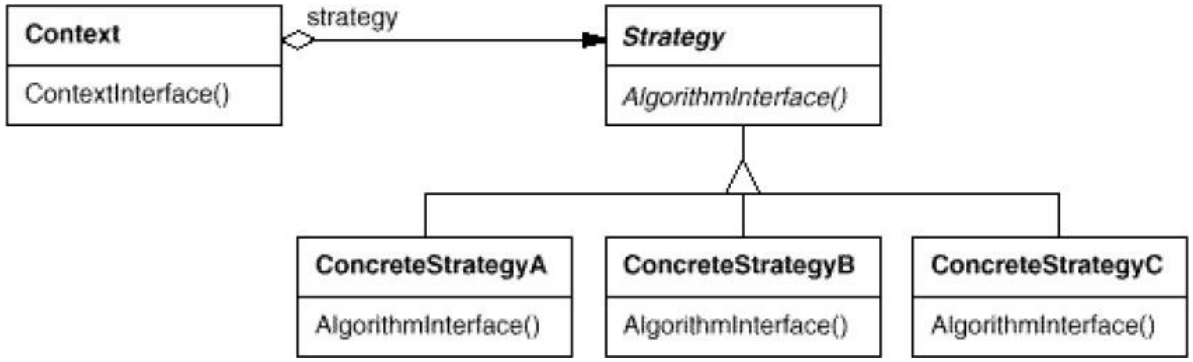
Food for thought…

- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.
  - **Where to define the state transitions ?**

- If criteria is fixed, transition can be defined in the context.
- More flexible if transition is specified in the State subclass.
- Introduces dependencies between subclasses.
- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
- First is desirable if the context changes state infrequently.
- Later is desirable if the context changes state frequently.

Pattern: Strategy

objects that hold alternate algorithms to solve a problem



Strategy pattern

- pulling an algorithm out from the object that contains it, and encapsulating the algorithm (the "strategy") as an object
- each strategy implements one behavior, one implementation of how to solve the same problem
  - how is this different from **Command** pattern?
- separates algorithm for behavior from object that wants to act
- allows changing an object's behavior dynamically without extending / changing the object itself
- **examples**:
  - file saving/compression
  - layout managers on GUI containers
  - AI algorithms for computer game players

Strategy example: Card player

```
// Strategy hierarchy parent
// (an interface or abstract class)
public interface Strategy {
public Card getMove();
}
// setting a strategy
player1.setStrategy(new SmartStrategy());
// using a strategy
Card p1move = player1.move(); // uses strategy
```

Strategy: Encapsulating Algorithms

Name: Strategy design pattern

Problem description:

Decouple a policy-deciding class from a set of mechanisms, so that different mechanisms can be changed transparently.

Example:

A mobile computer can be used with a wireless network, or connected to an Ethernet, with dynamic switching between networks based on location and network costs.

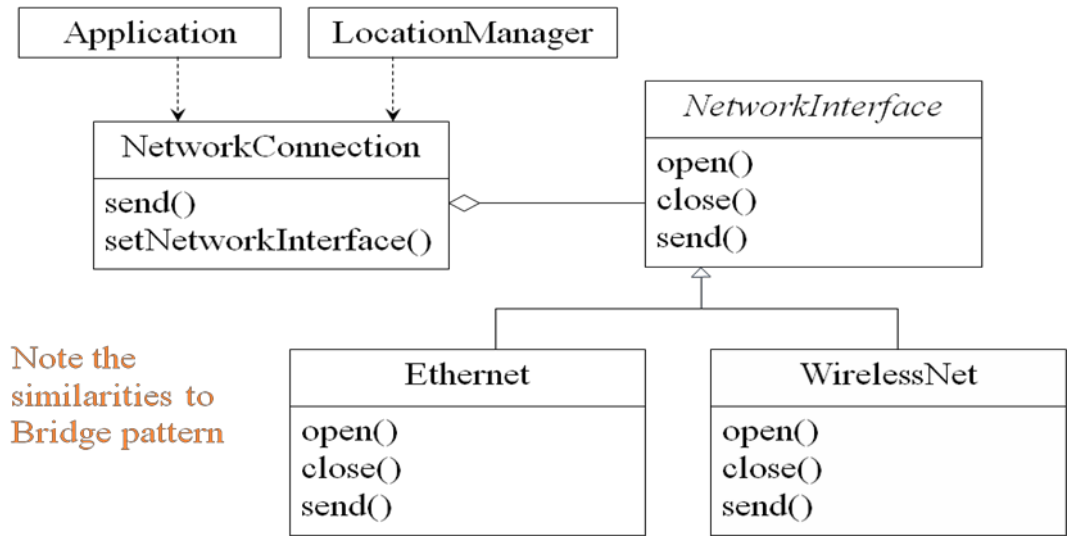
Solution:

A Client accesses services provided by a Context.

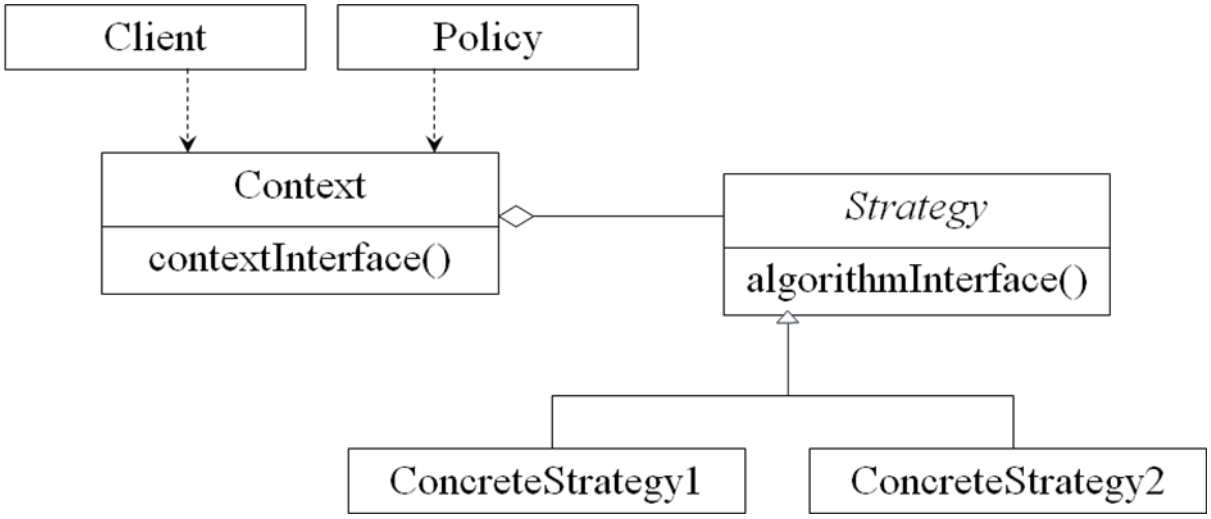
The Context services are realized using one of several mechanisms, as decided by a Policy object.

The abstract class Strategy describes the interface that is common to all mechanisms that Context can use. Policy class creates a ConcreteStrategy object and configures Context to use it.

Strategy Example: Class Diagram for Mobile Computer



Strategy: Class Diagram



Strategy: Consequences

Consequences:

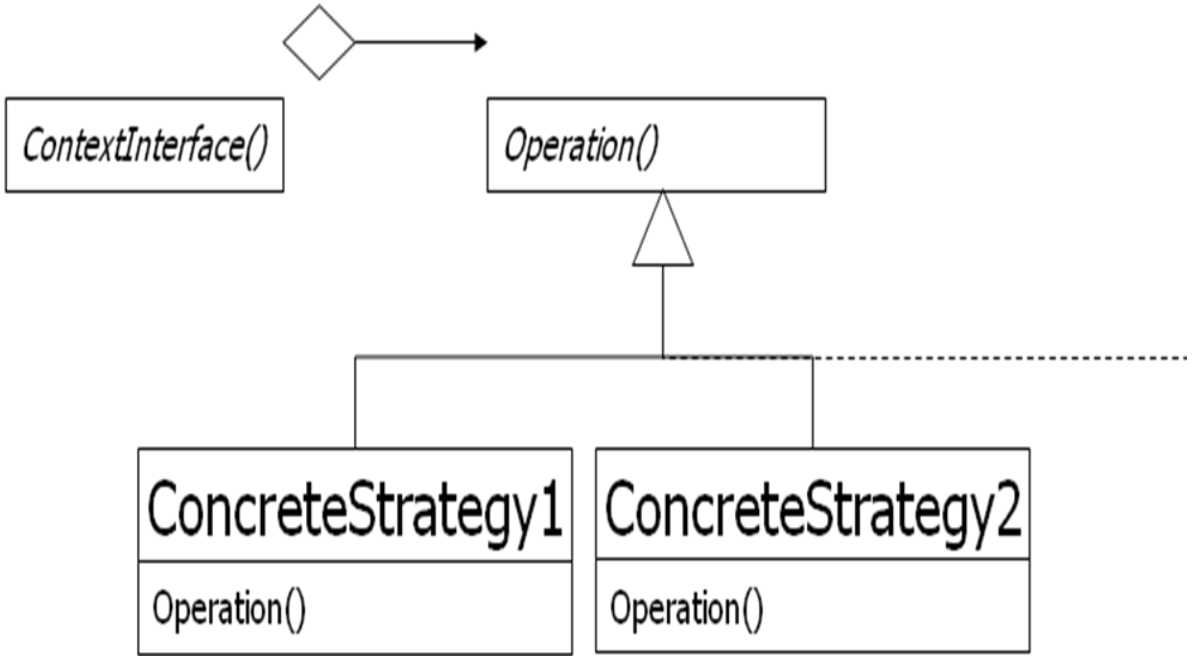
ConcreteStrategies can be substituted transparently from Context.

Policy decides which Strategy is best, given the current circumstances.

New policy algorithms can be added without modifying Context or Client.

Strategy

- **You want to**
  - use different algorithms depending upon the context
  - avoid having to change the context or client
- **Strategy**
  - decouples interface from implementation
  - shields client from implementations
  - Context is not aware which strategy is being used; Client configures the Context
  - strategies can be substituted at runtime
  - example: interface to wired and wireless networks
- Make algorithms interchangeable---”changing the guts”
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



Template Method

Conducted By Raghavendar Japala

Topics – Template Method

- Introduction to Template Method Design Pattern
- Structure of Template Method
- Generic Class and Concrete Class
- Plotter class and Plotter Function Class

Introduction

The DBAnimationApplet illustrates the use of an **abstract class** that serves as a template for classes with shared functionality.

An abstract class contains behavior that is common to all its subclasses. This behavior is encapsulated in nonabstract methods, which may even be declared ***final*** to prevent any modification. This action ensures that all subclasses will inherit the same common behavior and its implementation.

The abstract methods in such templates ensure the interface of the subclasses and require that context specific behavior be implemented for each concrete subclass.

Hook Method and Template Method

The abstract method paintFrame() acts as a placeholder for the behavior that is implemented differently for each specific context.

We call such methods, *hook* methods, upon which context specific behavior may be hung, or implemented.

The paintFrame() hook is placed within the method update(), which is common to all concrete animation applets. Methods containing hooks are called *template* methods.

Hook Method and Template Method (Con't)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

Frozen Spots and Hot Spots

A template method uses hook methods to define a common behavior.

Template method describes the fixed behaviors of a generic class, which are sometimes called **frozen spots**.

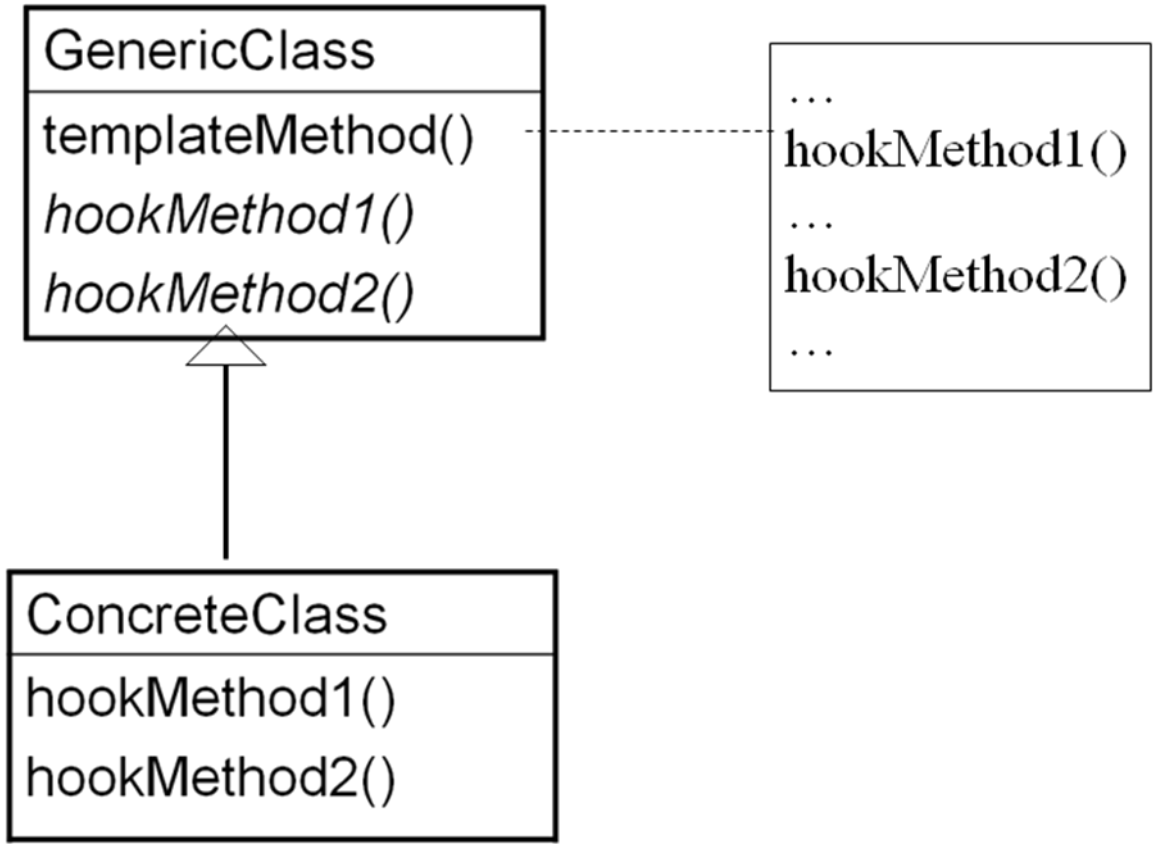
Hook methods indicate the changeable behaviors of a generic class, which are sometimes called **hot spots**.

Hook Method and Template Method (Con't)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

Structure of the Template Method Design Pattern



Structure of the Template Method Design Pattern (Con't)

GenericClass (e.g., DBAnimationApplet), which defines abstract hook methods (e.g., paintFrame()) that concrete subclasses (e.g., Bouncing-Ball2) override to implement steps of an algorithm and implements a template method (e.g., update()) that defines the skeleton of an algorithm by calling the hook methods;

ConcreteClass (e.g., Bouncing-Ball2) which implements the hook methods (e.g., paintFrame()) to carry out subclass specific steps of the algorithm defined in the template method.

Structure of the Template Method Design Pattern (Con't)

In the Template Method design pattern, *hook methods* **do not** have to be abstract.

The generic class may provide default implementations for the hook methods.

Thus the subclasses have the option of overriding the hook methods or using the default implementation.

The initAnimator() method in DBAnimationApplet is a

nonabstract hook method with a default implementation.

The init() method is another template method.

A Generic Function Plotter

The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

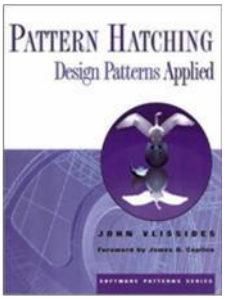
A concrete plotter PlotSine will be implemented to plot the function

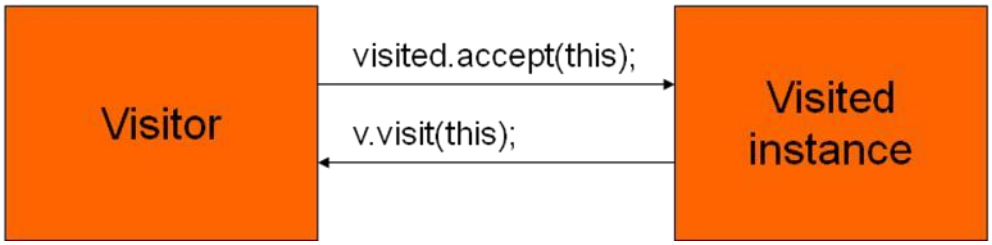
$y = sin\ x$



Pattern Hatching

Visitor pattern

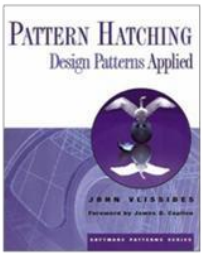






Pattern Hatching

Visitor Pattern




```
Class Visitor {
public:
        Visitor();
        void visit(File*);
        void visit(Directory*);
        void visit (Link*);
};

        }
Visitor cat;
node->accept(cat);
```

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick ();
```

```
void Visitor::visit (File* f)
        {f->streamOut(cout);}

void Visitor::visit (Directory* d)
        {cerr << "no printout for a
        directory";}

void Visitor::visit (Link* l)
        {l->getSubject()->accept(*this);}
```

```
void File::accept (Visitor& v)
        {v.visit(this);}
void Directory::accept (Visitor& v)
        {v.visit(this);}
void Link::accept (Visitor& v)
        {v.visit(this);}
```

What to Expect from Design Patterns, A Brief History, The Pattern Community An Invitation, A Parting Thought.

What to Expect from Design Patterns?

- A Common Design Vocabulary.
- A Documentation and Learning Aid.
- An Adjunct to Existing Methods.
- A Target for Refactoring.

A common design vocabulary

1. Studies of expert programmers for conventional languages have shown that knowledge and experience isn't organized simply around syntax but in larger conceptual structures such as algorithms, data structures and idioms [AS85, Cop92, Cur89, SS86], and plans for fulfilling a particular goal [SE84].
2. Designers probably don't think about the notation they are using for recording the designing as much as they try to match the current design situation against plans, data structures, and idioms they have learned in the past.
3. Computer scientists name and catalog algorithms and data structures, but we don't often name other kinds of patterns. Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.

A document and learning aid:

1. Knowing the design patterns makes it easier to understand existing systems.
2. Most large object-oriented systems use this design patterns people learning object-oriented programming often complain that the systems they are working with use inheritance in convoluted ways and that it is difficult to follow the flow of control.
3. In large part this is because they do not understand the design patterns in the system learning these design patterns will help you understand existing object-oriented system**.**

An adjacent to existing methods:

1. Object-oriented design methods are supposed to promote good design, to teach new designers how to design well, and standardize the way designs are developed.
2. A design method typically defines a set of notations (usually graphical) for modeling various aspects of design along with a set of rules that govern how and when to use each notation.
3. Design methods usually describe problems that occur in a design, how to resolve them and how to evaluate design. But then have not been able to capture the experience of expert designers.
4. A full fledged design method requires more kinds of patterns than just design patterns there can also be analysis patterns, user interface design patterns, or performance tuning patterns but the design patterns are an essential part, one that's been missing until now.

A target for refactoring:

1. One of the problems in developing reusable software is that it often has to be recognized or refactored [OJ90].
2. Design patterns help you determine how to recognize a design and they can reduce a amount of refactoring need to later.

The life cycle of object-oriented software has several faces. Brain Foote identifies these phases as the prototyping expansionary, and consolidating phases [Foo92**].**

Design Patterns Applied:

Example: An Hierarchical File System

Tree Structure → Composite

Patterns Overview

Symbolic Links → Proxy

Extending Functionality → Visitor

Single User Protection → Template Method

Multi User Protection → Singleton

User and Groups → Mediator

A Brief History of Design Patterns

- 1979--Christopher Alexander pens The Timeless Way of Building
  - Building Towns for Dummies
  - Had nothing to do with software
- 1994--Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four, or GoF) publish Design patterns: Elements of Reusable Object-Oriented Software
  - Capitalized on the work of Alexander
  - The seminal publication on software design patterns.

What's In a Design Pattern—1994

- The GOF book describes a pattern using the following four attributes:
  - The *name* to describes the pattern, its solutions and consequences in a word or two
  - The *problem* describes when to apply the pattern
  - The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The *consequences* are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk.

What's In a Design Pattern – 2002

- Grand's book is the latest offering in the field and is very Java centric. He develops the GOF attributes to a greater granularity and adds the Java specifics
  - Pattern name—same as GOF attribute
  - Synopsis—conveys the essence of the solution
  - Context—problem the pattern addresses
  - Forces—reasons to, or not to use a solution
  - Solution—general purpose solution to the problem
  - Implementation—important considerations when using a solution
  - Consequences—implications, good or bad, of using a solution
  - Java API usage—examples from the core Java API
  - Code example—self explanatory
  - Related patterns—self explanatory

Grand's Classifications of Design Pattern:

- Fundamental patterns
- Creational patterns
- Partitioning patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

The Pattern Community An Invention

- ❑ Christopher Alexander is the architect who first studied

Patterns in buildings and communities and developed

A PATTERN LANGUAGE for generating them.

- ❑ His work has inspired time and again. So it's fitting worth

while To compare our work to his.

- ❑ Then we'll look at other's work in software-related patterns.

Alexander's Pattern Languages

There are many ways in which our work is like Alexander's

Both are based on observing existing systems and looking for patterns in them.

Both have templates for describing patterns although

our templates are quite different)..

But there are just as many ways in which our work different.

- ➢ People have been making buildings for thousands of years, and there are many

classic examples to draw upon. We have been making Software systems for a

Relatively short time, and few are considered classics.

- ➢ Alexander gives an order in which his patterns should be used; we have not.
- ➢ Alexander's patterns emphasize the problems they adderss
- ➢ where as design patterns describes the solutions in more detail.
- ➢ Alexander claims his patterns will generate complete buildings.

We do not claim that our patterns will generate complete programs.

When Alexander claims you can design a house simply applying his patterns one after

Another ,he has goals similar to those of object-oriented design methodologies who

Gives step-by-step rules for design,

In fact ,we think it’s unlikely that there will ever be a compete pattern language for soft

-ware.

But certainly possible to make one that is more complete.

A Parting Thought.

The best designs will use many design patterns that dovetail And intertwine to produce a greater whole.

As Alexander says:

- It is possible to make buildings by stringing together pattern’s In a rather loose way,
- A building made like this , is an assembly of patterns. it is not

  Dense. It is not profound. but it is also possible to put pattern’s

  together
- In such a way that many patterns overlap in the same physical Space: the building is very dense; it has many meaning captured In a small space; and through this density, it becomes profound.