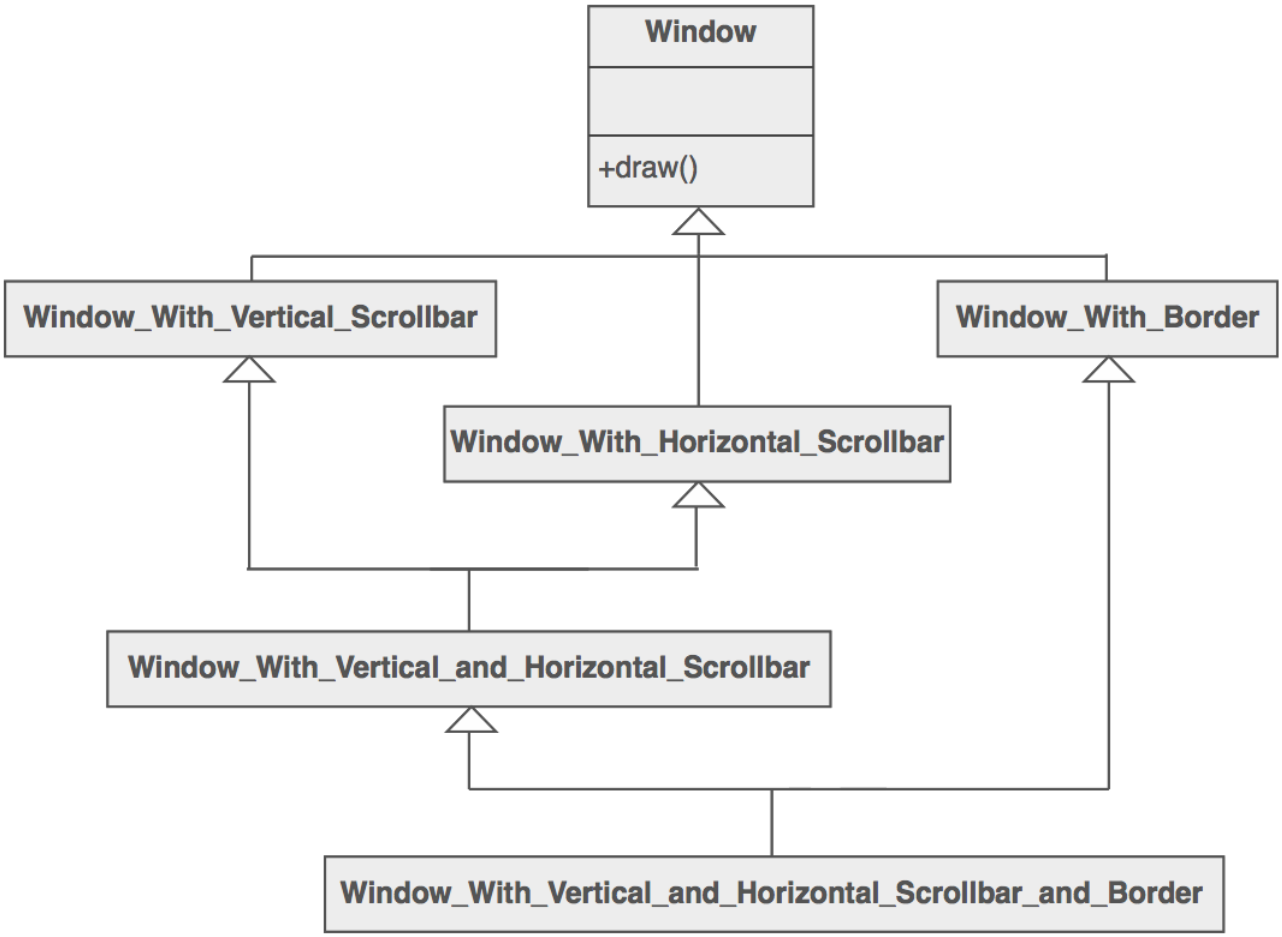


Check list

1. Ensure that object overhead is an issue needing attention, and, the client of the class is able and willing to absorb responsibility realignment.
2. Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
3. Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.
4. Create a Factory that can cache and reuse existing class instances.
5. The client must use the Factory instead of the new operator to request objects.
6. The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

Rules of thumb

- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Flyweight is often combined with Composite to implement shared leaf nodes.
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.

- Flyweight explains when and how State objects can be shared.

Proxy Design Pattern:-

Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Discussion

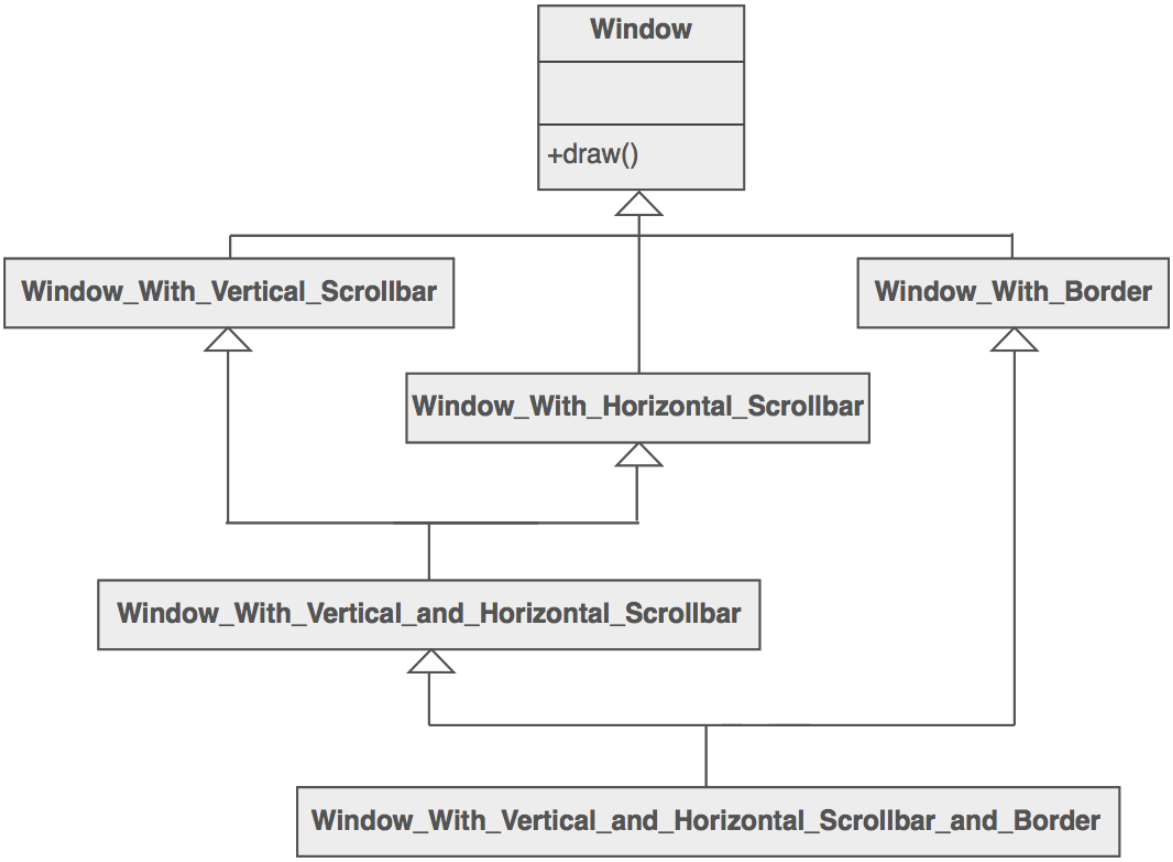
Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

1. A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
3. A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
4. A smart proxy interposes additional actions when an object is accessed. Typical uses include:
    - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
    - Loading a persistent object into memory when it's first referenced,
    - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

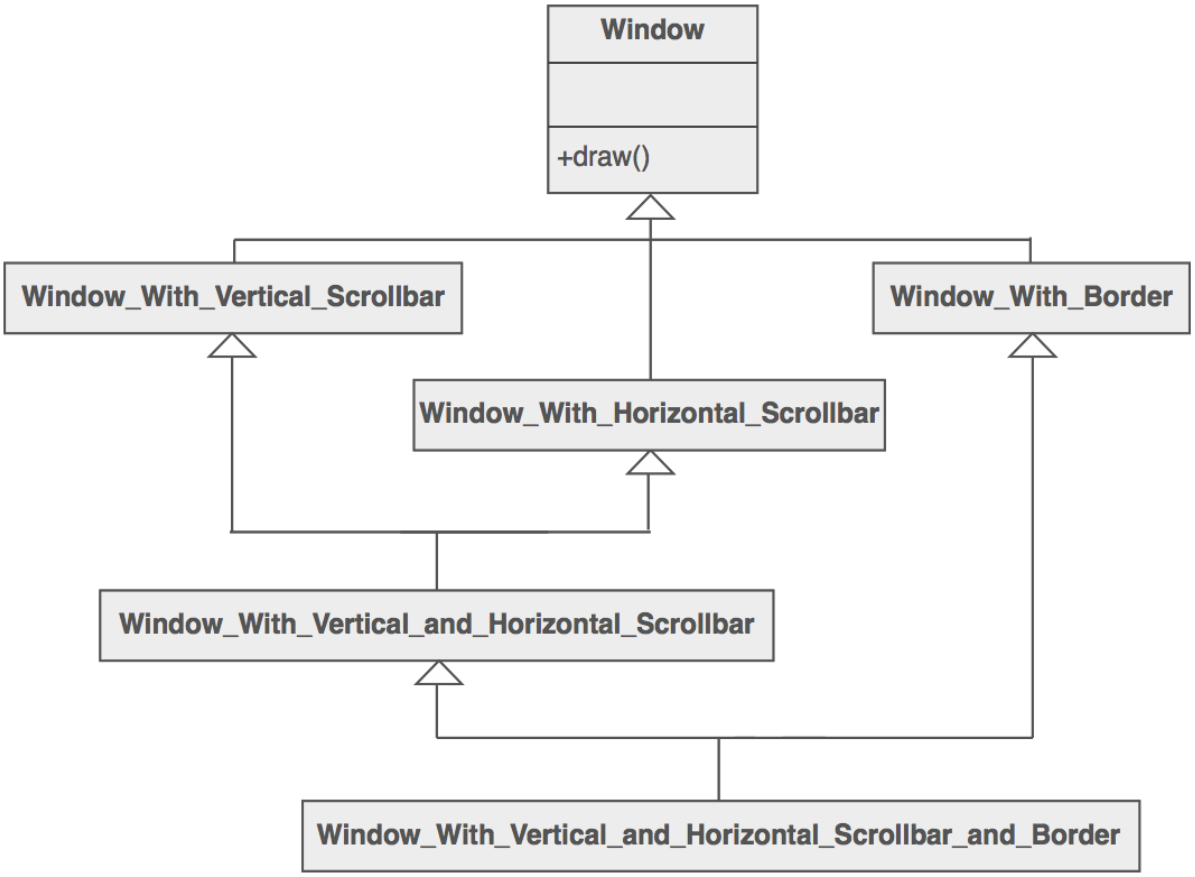
Structure

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Check list

1. Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.
2. Define an interface that will make the proxy and the original component interchangeable.
3. Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
4. The wrapper class holds a pointer to the real class and implements the interface.
5. The pointer may be initialized at construction, or on first use.
6. Each wrapper method contributes its leverage, and delegates to the wrappee object.

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

UNIT-IV

Behavioral Patterns

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

Behavioral Patterns (1)

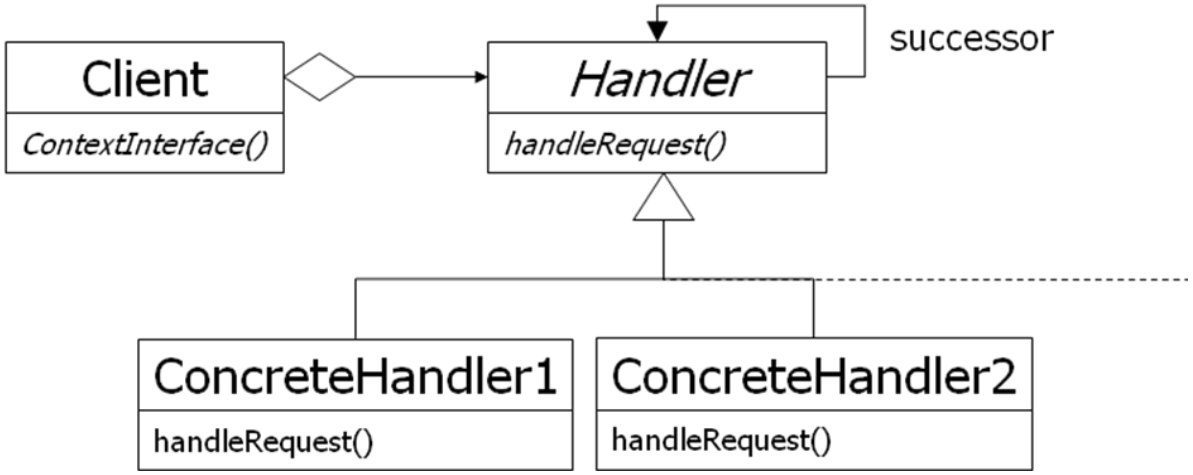
- Deal with the way objects interact and distribute responsibility.
- Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving.

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.

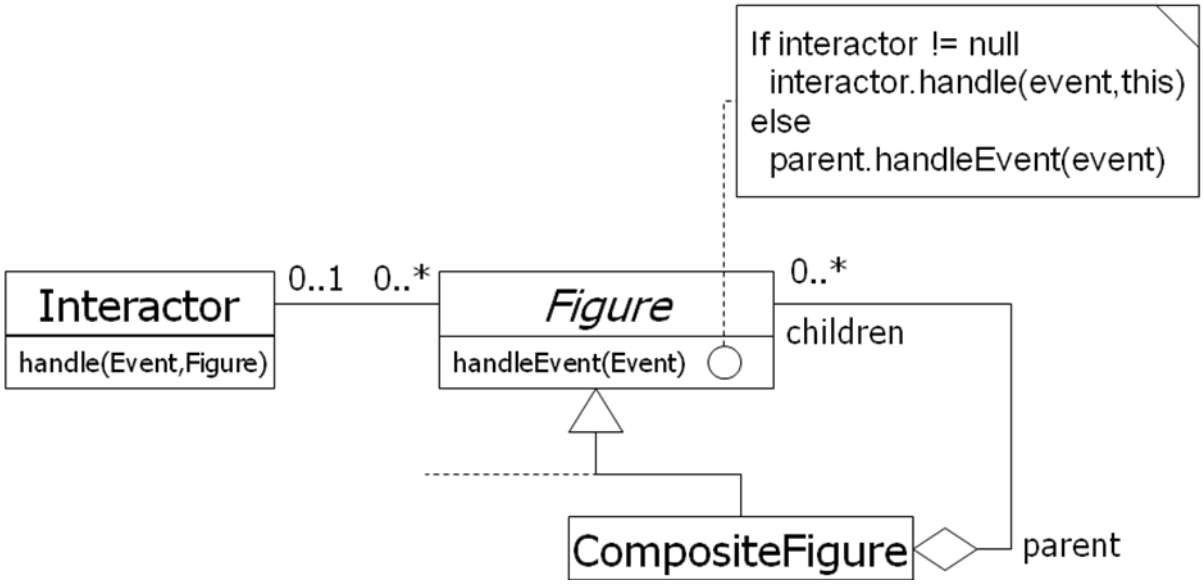
Chain of Responsibility:

- 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

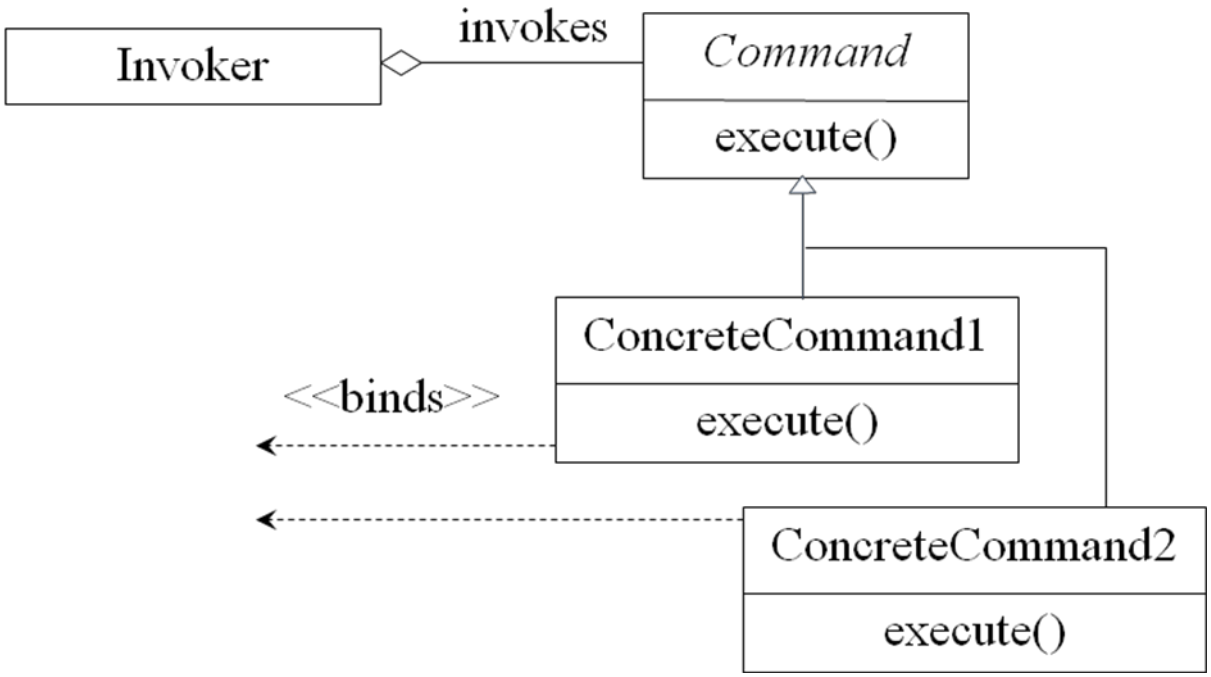
pattern Problem description:

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

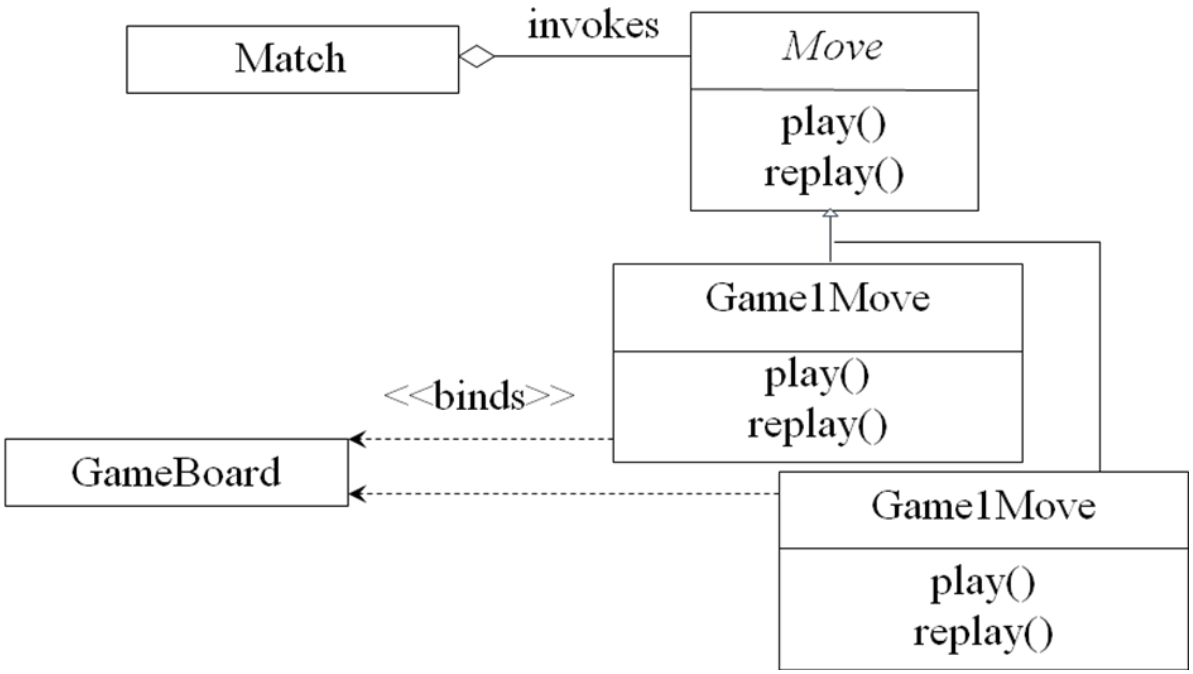
Solution:

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

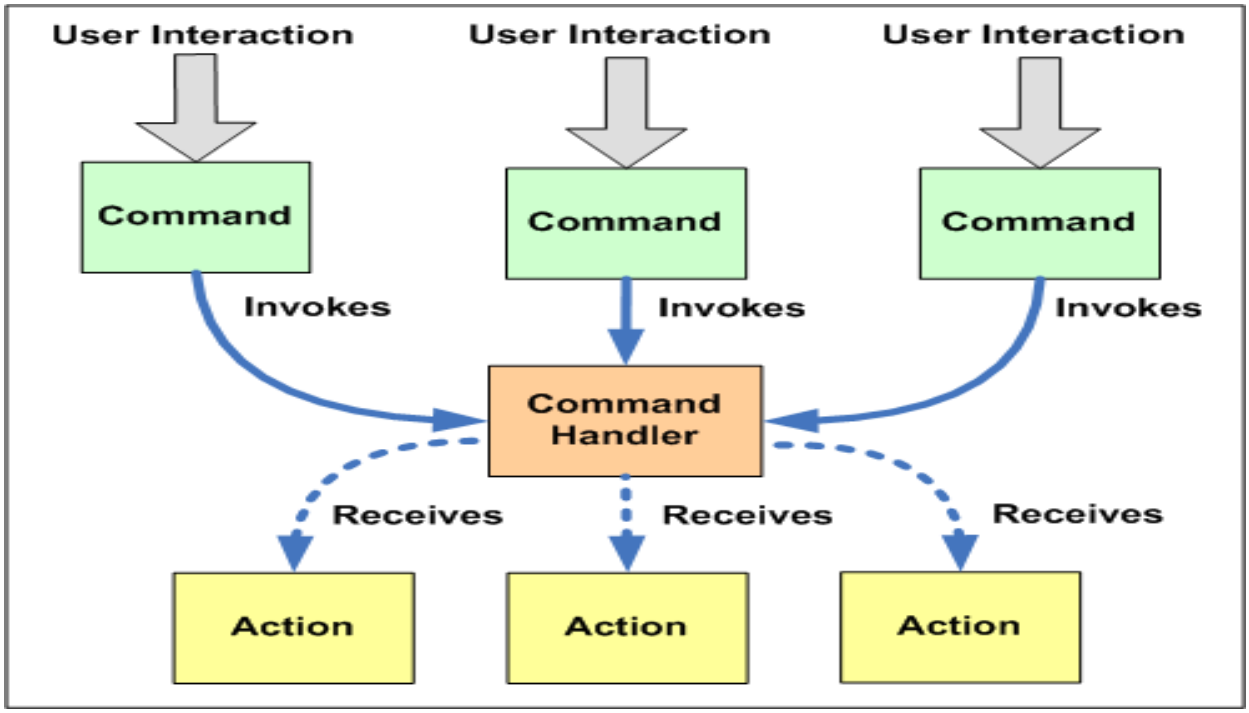
stored.

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:

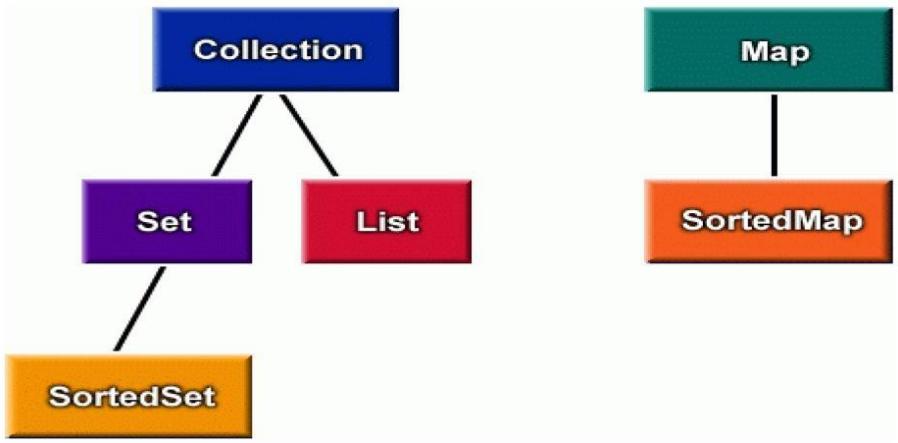
- Build a method from components in different classes
- Primary methods: the “normal” methods; choose the most specific one
- Before/After methods: guaranteed to
  run; No possibility of forgetting to
  call super
  Can be used to implement *Active Value* pattern
- Around methods: wrap around
  everything; 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:**

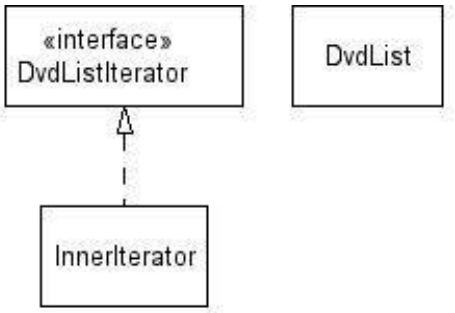
public class PlayerList {

public Iterator iterator() {

... } public int size() { ... }

public boolean isEmpty() { ... }

}



Command:Encapsulating Control Flow:

Name: Command design pattern

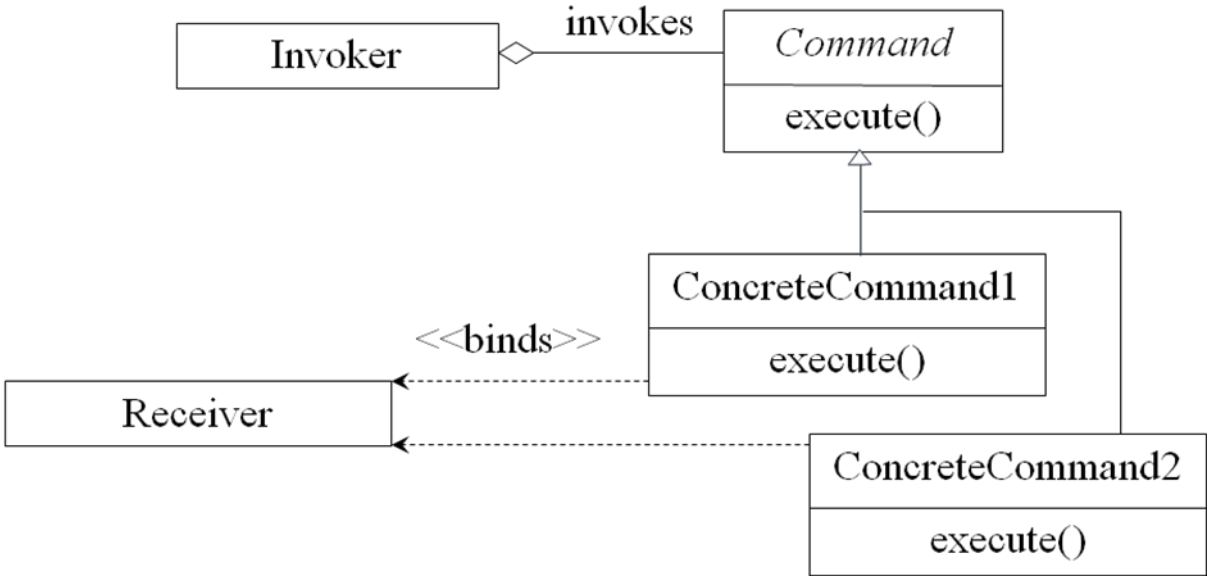
Problem description:

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

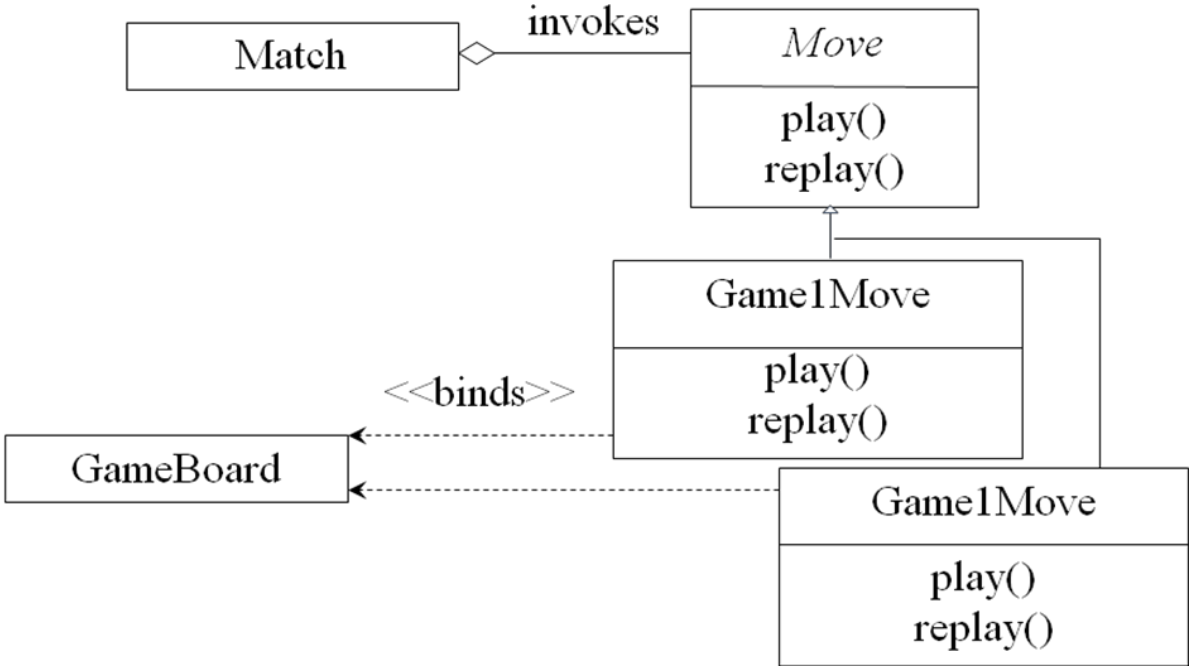
Solution:

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.

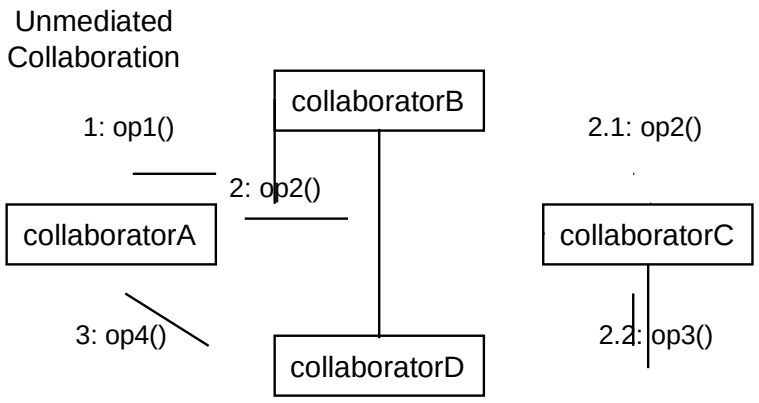
• Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

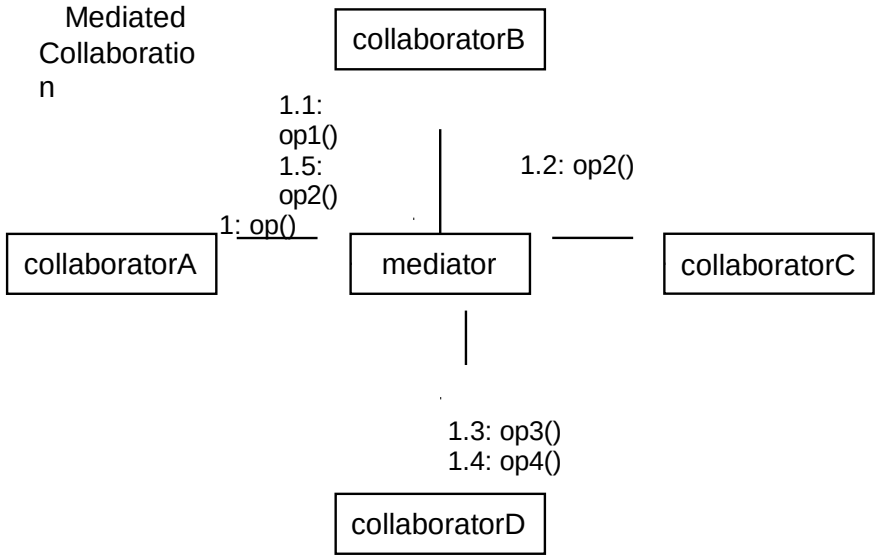
• Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasseses redefine certain steps of an algorithm without changing the algorithm's structure.

• 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.

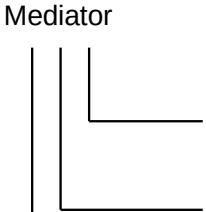
The Mediator Pattern:

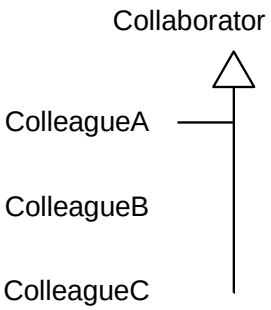
- The Mediator pattern reduces coupling and simplifies code when several objects must negotiate a complex interaction.
- Classes interact only with a mediator class rather than with each other.
- Classes are coupled only to the mediator where interaction control code resides.
- Mediator is like a multi-way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

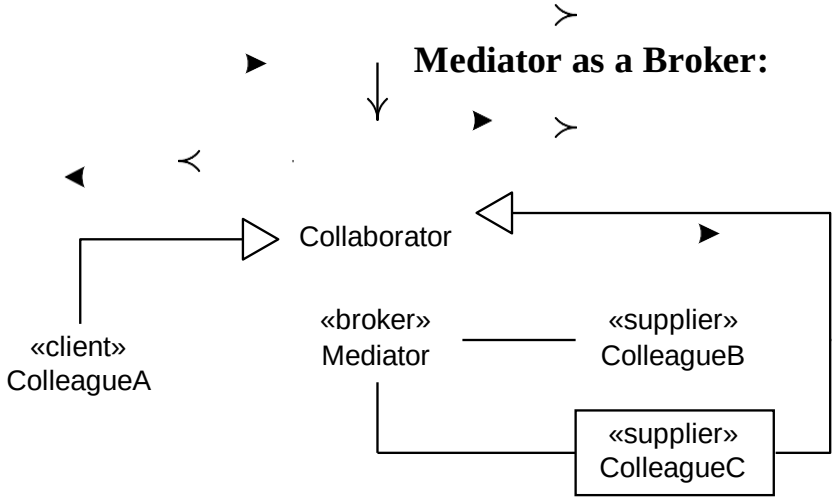




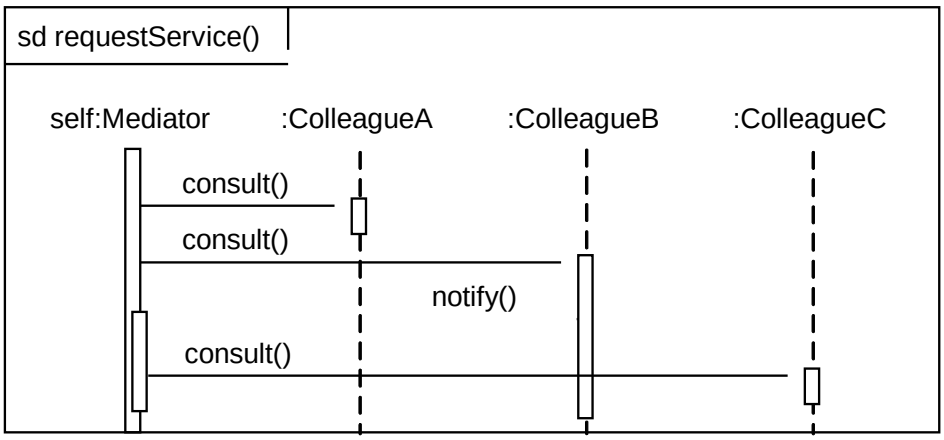
Mediator Pattern Structure:




Collaborator
ColleagueA
ColleagueB
ColleagueC



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.

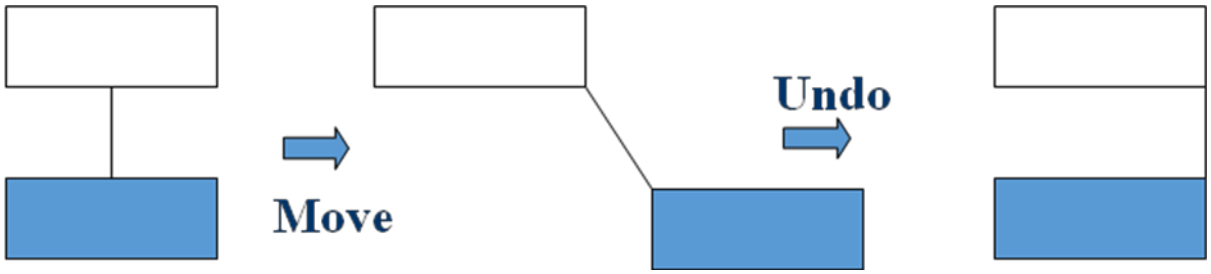
Motivation:

- 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.



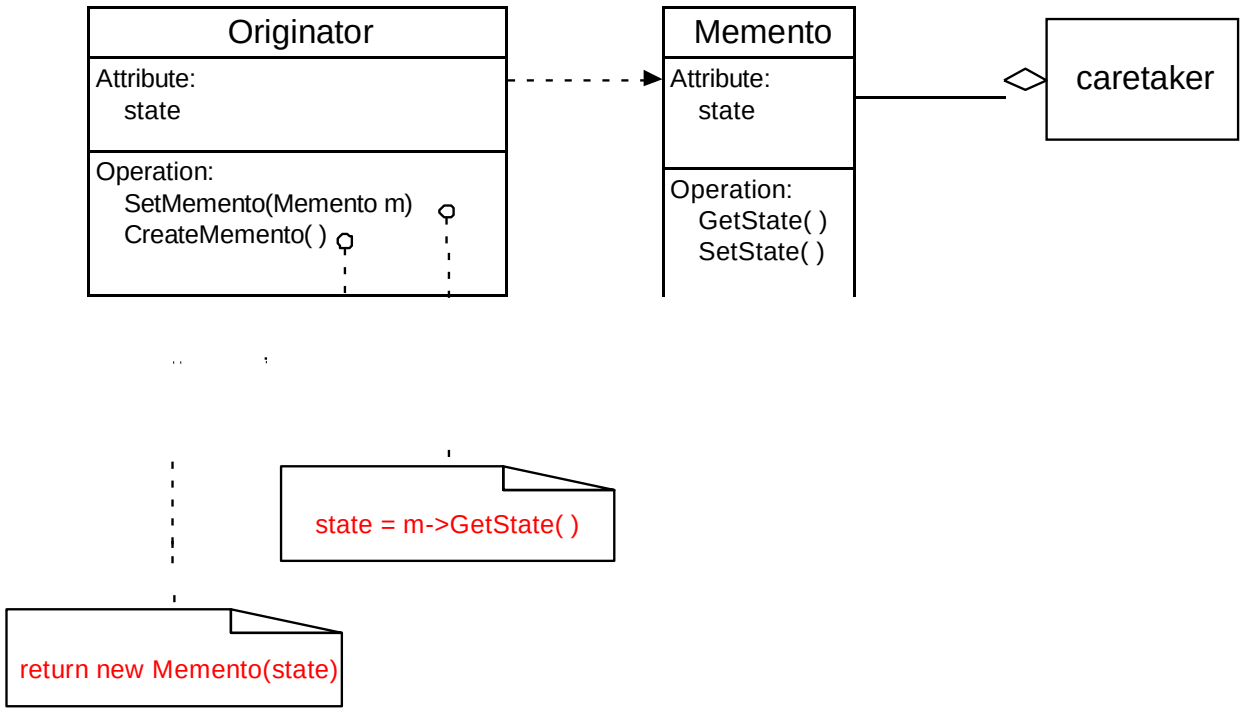
Memento pattern solves this problem as follows:

- 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:



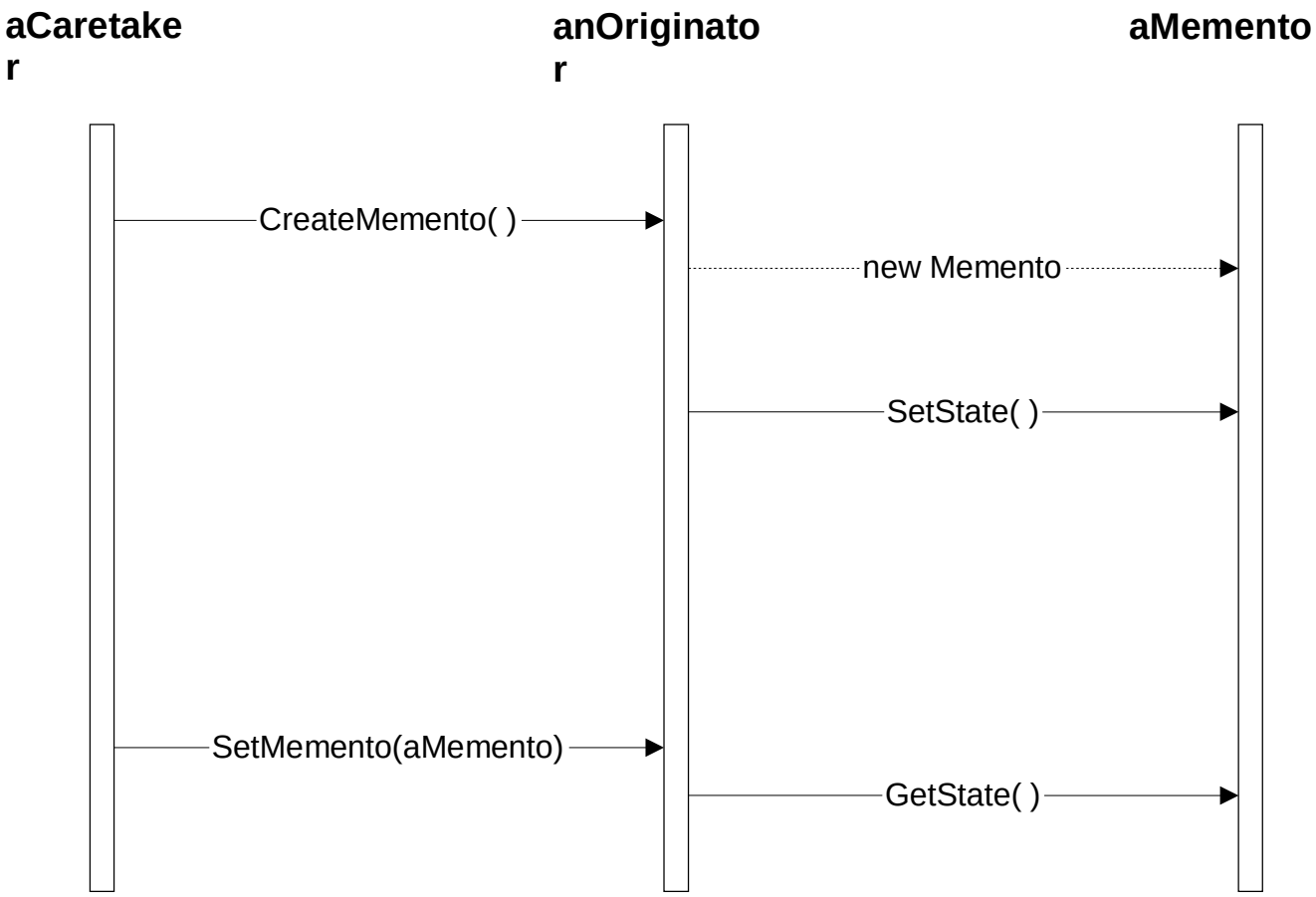
Participants:

- 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

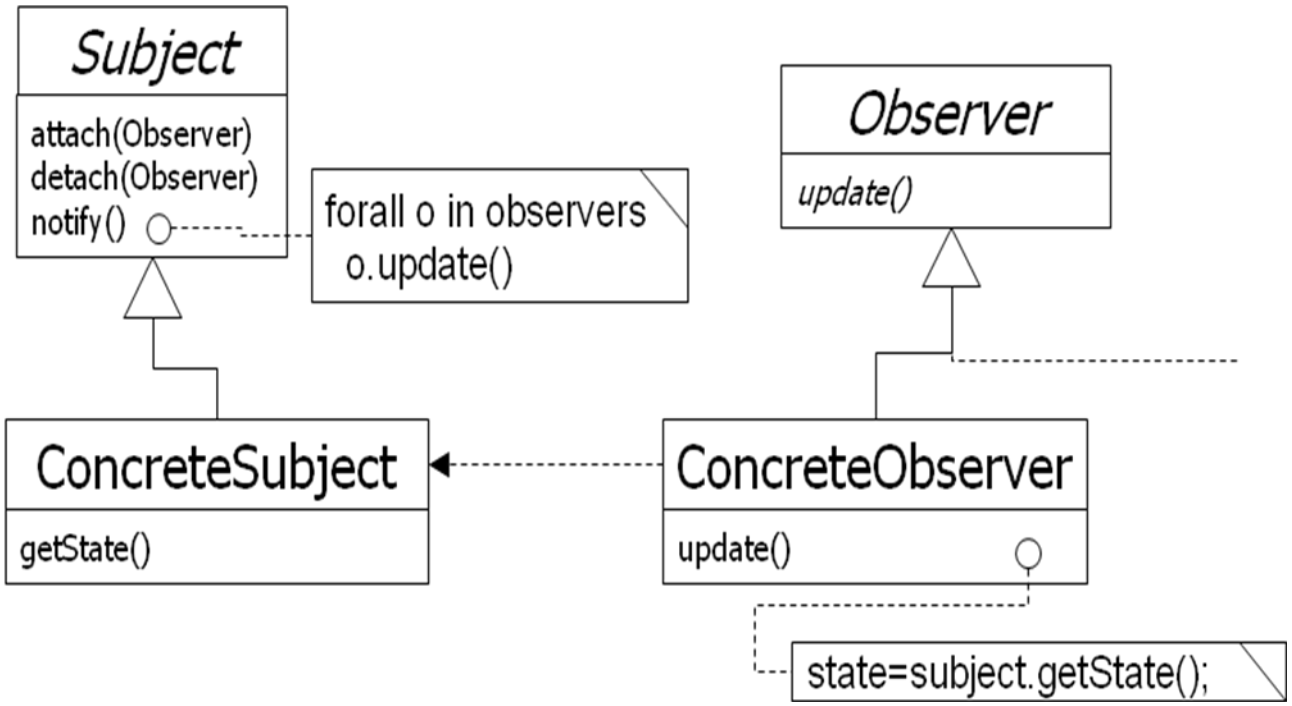
  Mementos can be used for iteration.

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:

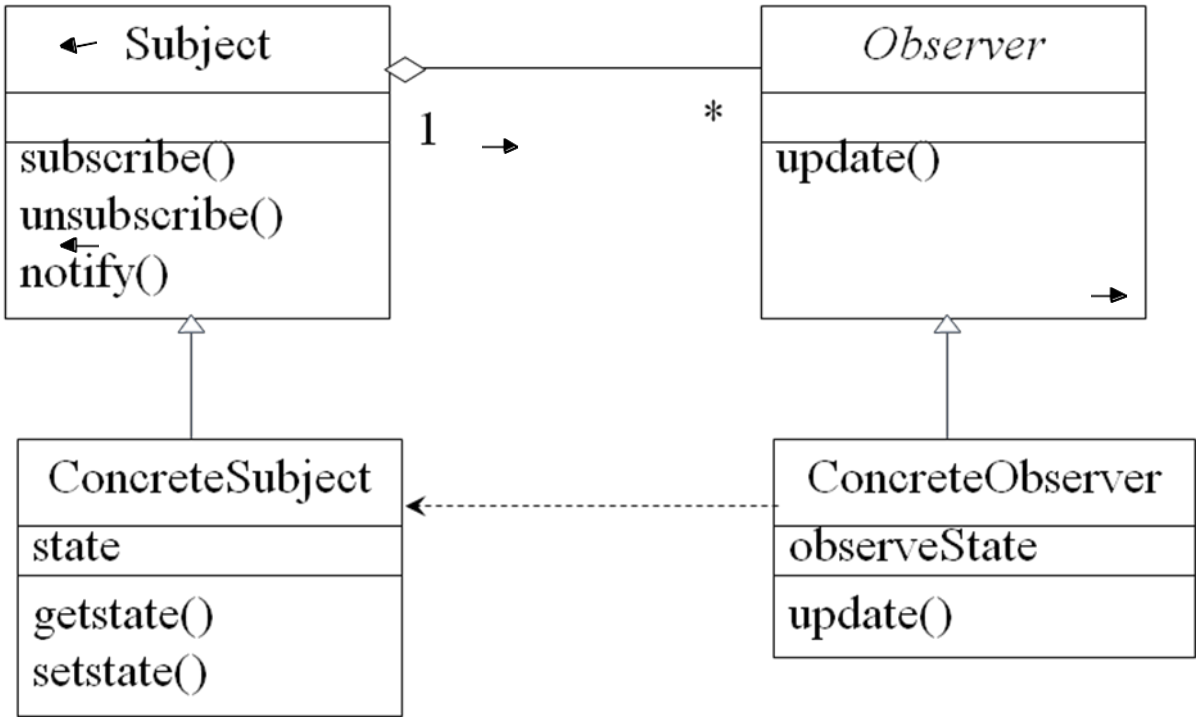
Maintains consistency across state of one Subject and many Observers.

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

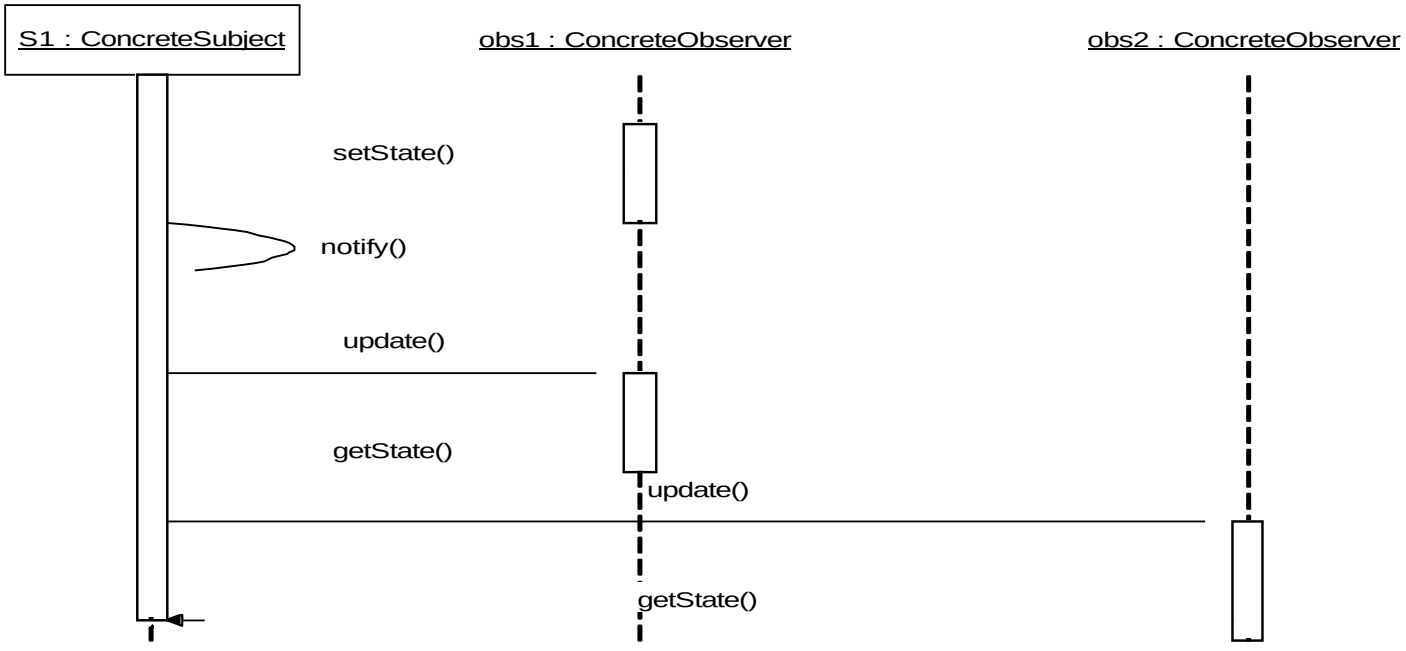


Observer: Consequences

Consequences:

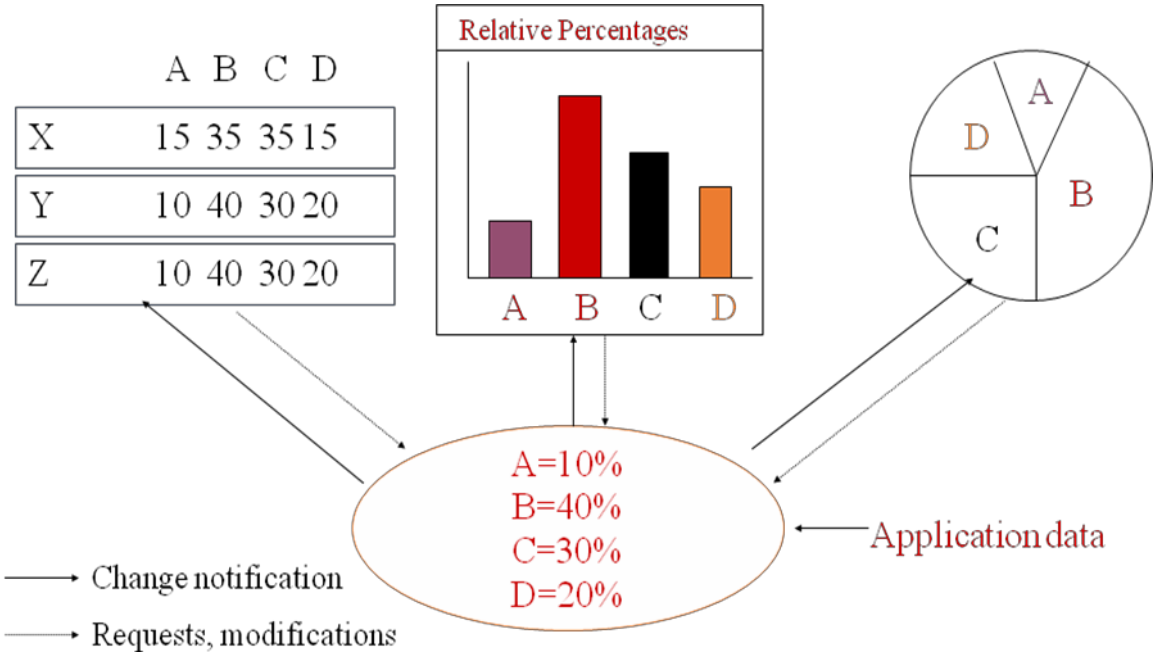
Decouples Subject, which maintains state, from Observers, who make use of the state. Can result in many spurious broadcasts when the state of Subject changes.

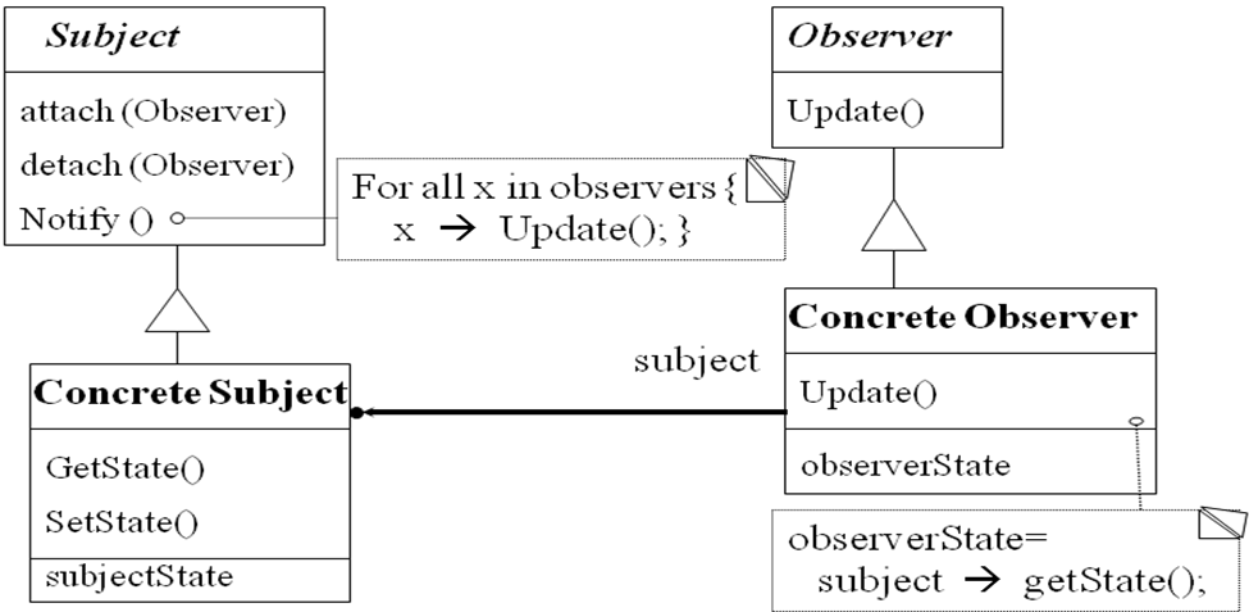
Collaborations in Observer Pattern



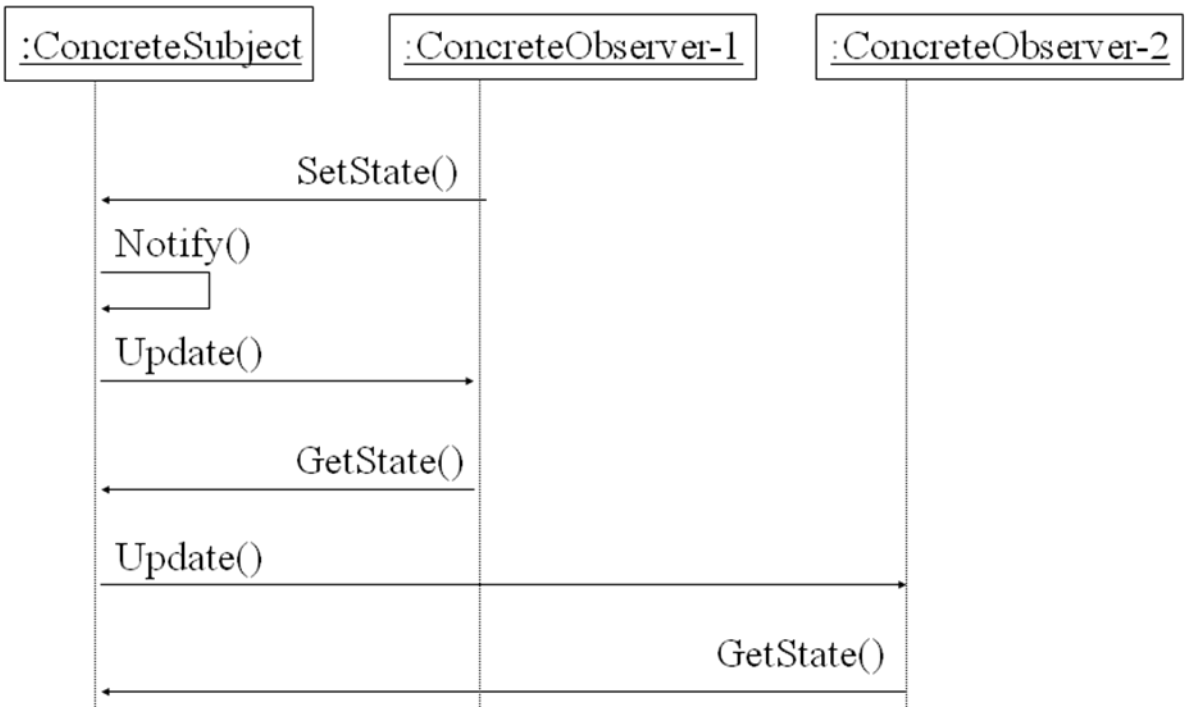
Observer Pattern [1]

- 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