CUGS SE Design Patterns part
Mediator, Memento, Interpreter

Peter Bunus (slides), Peter Fritzson (lecture)
Dept of Computer and Information Science
Linköping University, Sweden

pelab

Linköpings universitet
INSTITUTE OF TECHNOLOGY

Summary

- What make a good OO Design
- Behavioral Patterns

Mediator Pattern

Memento Pattern

Interpretor Pattern

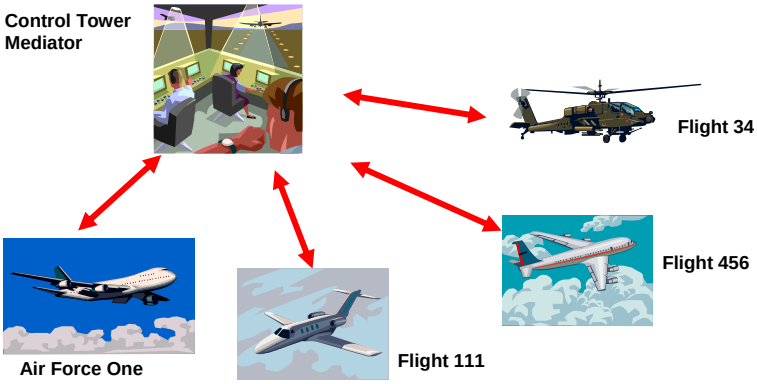
pelab

What Make a Good Object-Oriented Design?

- Encapsulate that varies.
- Favor composition over inheritance
- Program to interfaces not to implementation
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension but closed for modification
- Depend on abstraction. Do not depend on concrete classes
- Only talk to your friends
- Don't call us, we'll call you.
- A class should have only one reason to change

pelab

The Mediator – Non Software Example

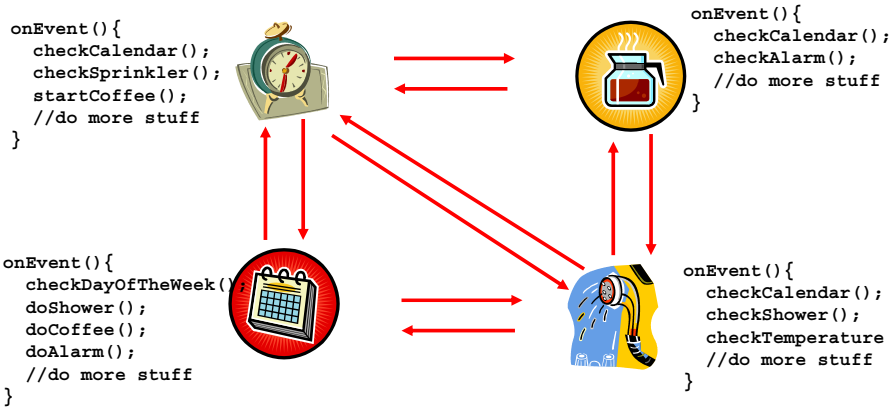


- The *Mediator* defines an object that controls how a set of objects interact.
- The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another.
- The constraints on who can take off or land are enforced by the tower.
- the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

pelab

The Mediator – Another Example

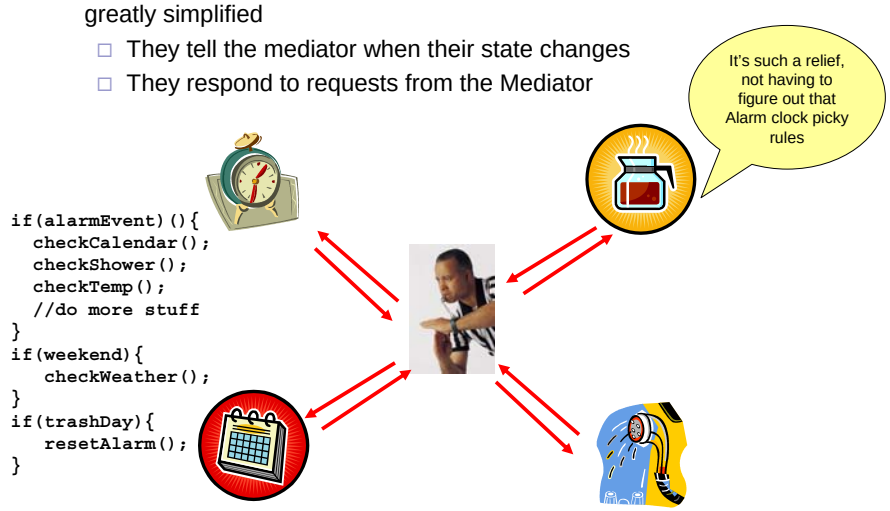
- Bob lives in the HouseOfFuture where everthing is automated:
  - When Bob hits the snooze button of the alarm the coffee maker starts brewing coffee
  - No coffee in weekends
  - .....



pelab

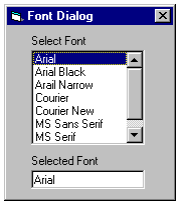
The Mediator in Action

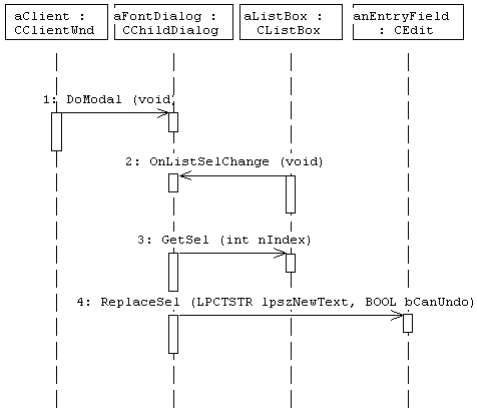
- With a Mediator added to the system all the appliance objects can be greatly simplified
  - They tell the mediator when their state changes
  - They respond to requests from the Mediator



pelab

Mediator and MFC (Microsoft Foundation Classes)

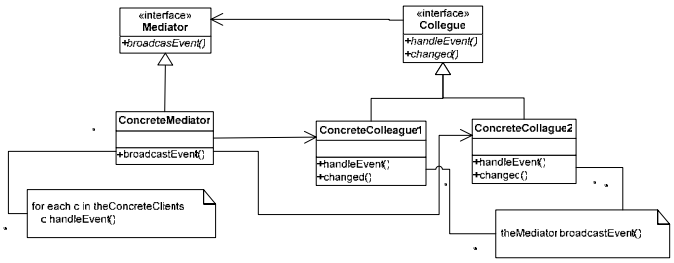




- The Client creates aFontDialog and invokes it.
- The list box tells the FontDialog (it's mediator) that it has changed
- The FontDialog (the mediator object) gets the selection from the list box
- The FontDialog (the mediator object) passes the selection to the entry field edit box

pelab

Actors in the Mediator Pattern



Mediator

defines an interface for communicating with Colleague objects

ConcreteMediator

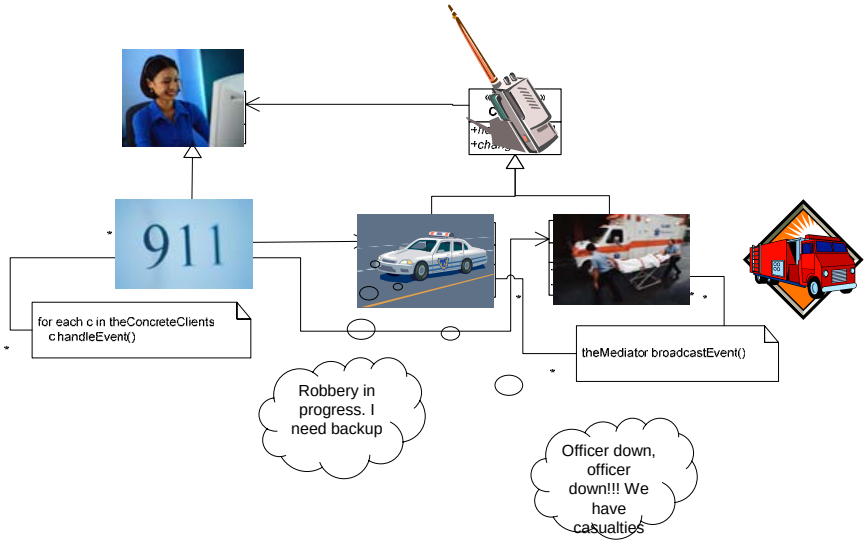
implements cooperative behavior by coordinating Colleague objects
knows and maintains its colleagues

Colleague classes (Participant)

each Colleague class knows its Mediator object (has an instance of the mediator)
each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

pelab

Yet Another Example



pelab

Mediator advantages and disadvantages

++++ Changing the system behavior means just sub classing the mediator. Other objects can be used as is.

++ Since the mediator and its colleagues are only tied together by a loose coupling, both the mediator and colleague classes can be varied and reused independent of each other.

++++ Since the mediator promotes a One-to-Many relationship with its colleagues, the whole system is easier to understand (as opposed to a many-to-many relationship where everyone calls everyone else).

++ It helps in getting a better understanding of how the objects in that system interact, since all the object interaction is bundled into just one class - the mediator class.

--- Since all the interaction between the colleagues are bundled into the mediator, it has the potential of making the mediator class very complex and monolithically hard to maintain.

Issues

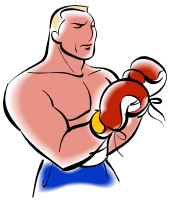
- When an event occurs, colleagues must communicate that event with the mediator. This is somewhat reminiscent of a subject communicating a change in state with an observer.
- One approach to implementing a mediator, therefore, is to implement it as an observer following the observer pattern.

Mediator versus Observer

pelab

Mediator versus Observer

receive notification of changes



Observer does so through an abstract mechanism. Allows source of notification to be independent of its observers.

Observer

Multiple observers

Each observer knows how to query the state change in its subject

receive notification of changes

In Mediator, the source must know its mediator. This makes it possible for mediator to define reactions to each stimulus.

One mediator per pattern

All the response actions are stored in the mediator. The mediator may modify states of the concrete colleagues

Mediator

pelab

Mediator versus Facade

Façade abstracts a subsystem of objects to provide a convenient interface. **Unidirectional**.

Façade

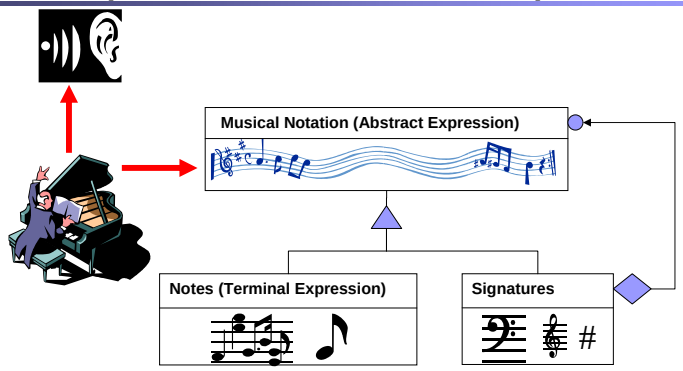
Façade objects make requests of the subsystem, but not vice-versa.

Enables cooperative behaviour, that colleagues don't or can't provide. **Multidirectional**.

Mediator

pelab

The Interpreter – Non Software 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 score
- Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented

pelab

The Interpreter

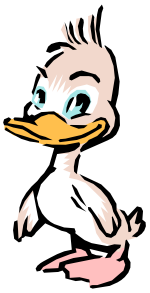
- Use the Interpreter to build and interpreter for a language
- Define a language grammar
  - Each grammar rule is represented by a class
- Represent sentences
  - Sentences within language can be represented by abstract syntax trees of instances of these classes

pelab

The Interpreter – How it Works

- We need to implement an educational tool for children to learn programming.
- Using our tool, each child gets to control Ugly Duckly with the help of a simple language.
- Here is a example of the language:

```
right;
while (daylight) fly;
quack;
```



pelab

The Ugly Dukly Language

```
right;
while (daylight) fly;
quack;
```

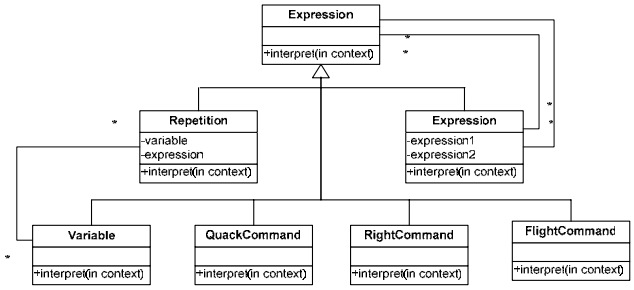
```
expression ::= <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition := while '(' <variable> ')' <expression>
variable :=[A-Z,a-z];
```

- We got the grammar, now all we need is a way to represent and interpret sentences in the grammar

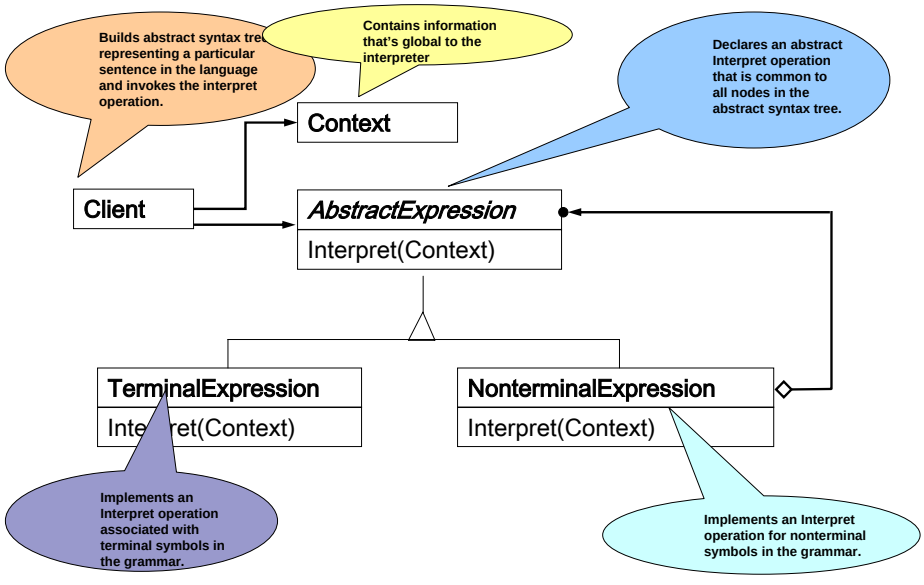
The Ugly Dukly Language Classes

- We define a class based representation for the grammar along with an interpreter to interpret the sentences

```
expression ::= <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition := while '(' <variable> ')' <expression>
variable :=[A-Z,a-z];
```

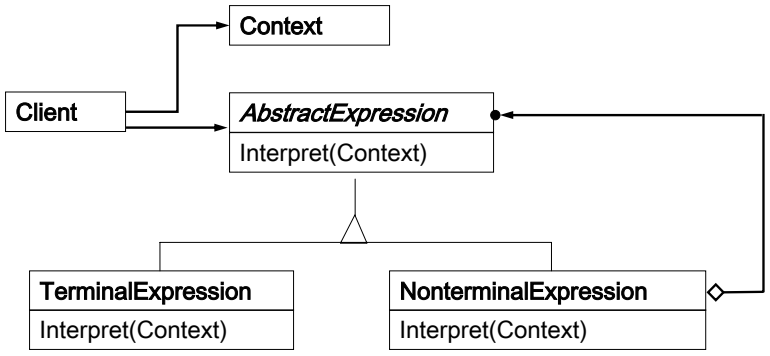


Interpreter UML



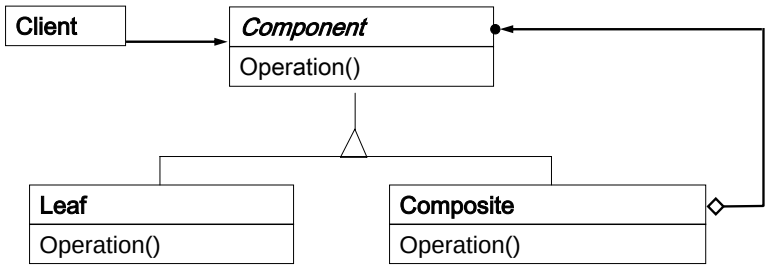
pelab

Interpreter – Looks familiar?



pelab

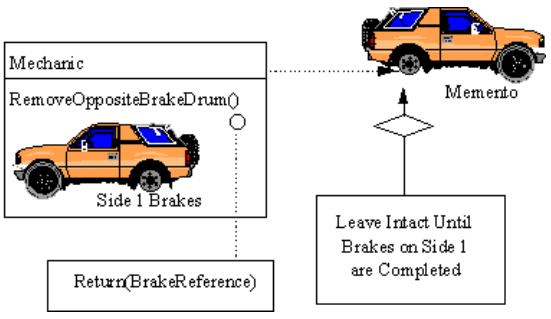
Composite (Structural Pattern)



Interpretor Advantages and Disadvantages

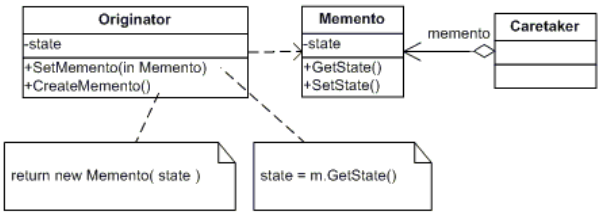
- +++ Representing each grammar rule in a class makes the language easy to implement
- ++ Because the grammar is represented by classes, you can easily change or extend the language
- ++By adding additional methods to the class structure, you can add new behaviors beyond interpretation (pretty printing)
- --- Use Interpreter when you need to implement a simple language
- ---- Appropriate when you have a simple grammar and simplicity is more important than efficiency
- ++ Use for scripting and programming languages
- It can become cumbersome when the number of grammar rule is large. In this cases a parser/compiler generator may be more appropiate.

The Memento – Non Software 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*

Memento



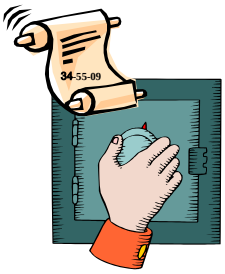

```
Originator o = new Originator();
o.State = "On";

// Store internal state
Caretaker c = new Caretaker();
c.Memento = o.CreateMemento();

// Continue changing originator
 o.State = "Off";

 // Restore saved state
 o.SetMemento( c.Memento );
```

• The Originator has a “CreateMemento” method, which creates a new Memento object and initializes it with the Originator’s internal state.

•The Memento object is stored in a Caretaker object, which acts as a supervisor to the storing and restoring of the state of Originator objects. A Caretaker object stores the Memento object, but cannot access its internal data.

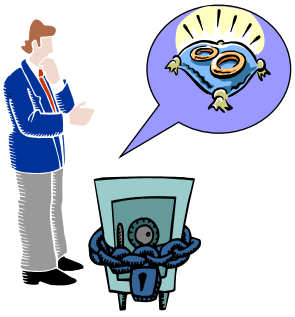
•When an undo or similar operation is called, the Caretaker calls the “Set Memento” method of the Originator and passes in the appropriate Memento object. The Originator’s “Set Memento” method sets all of its internal variables to the ones recorded earlier in that Memento object. Once all of the variables are reset its earlier state is restored and it will behave as it did before the Memento was originally recorded

Memento – Another Non Software Example



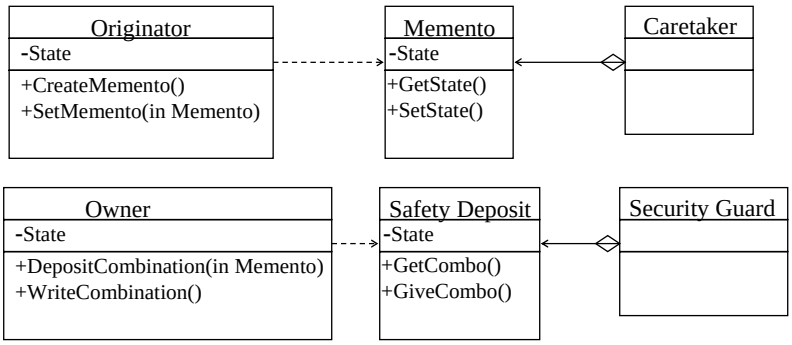
Memento

Caretaker



Originator

Memento Example




```
Owner o = new Owner();
o.State = "123-45-78-45";

// Store code
SecurityG c = new SecurityG();
c.SafetyDeposit = o.DepositCombination();

// Continue changing code
o.State = "4444444444";

 // Restore saved code
 o.writeCombination( c.SafetyDeposit );
```


Memento – Rules of Thumb

Rules of thumb

If you only need one Memento, combine the Originator and Caretaker into one object (Brown, 1998).

If you need many Mementos, store only incremental changes (Brown, 1998) . This will help to save space.

Memento often used in conjunction with Command, Iterator and Singleton.

Implementation of the Memento design pattern varies depending on the programming language. Implement the Originator as a friend class to the Memento in C++. Implement the Memento as an inner-class of the Originator in Java (Achtziger, 1999) .

A Java Implementation of Memento

```
public class Originator {
   private T state;
   private class Memento { // value object
         private T mstate;
         private Memento(T state) { mstate = copy_of(state); }
         private T getState() { return mstate; }
   }
   public Memento createMemento() {
         return new Memento(state);
   }
   // continued...
```

A C++ Implementation of Memeto

```
class Originator {
   public:
         Memento* CreateMemento();
         void SetMemento(const Memento*);
   private:
         State* _state; // internal data structures
};
class Memento {
   public:
         virtual ~Memento();
   private:
        // private members accessible only to Originator
         friend class Originator;
         Memento();
         void SetState(State*);
         State* GetState();
         // ...
   private:
         State* _state; // ... };
```

- Only talk to your friends

pelab

Memento Advantages and Disadvantages

- +++Since object oriented programming dictates that objects should encapsulate their state it would violate this law if objects' internal variables were accessible to external objects. The memento pattern provides a way of recording the internal state of an object in a separate object without violating this law
- ++It eliminates the need for multiple creation of the same object (i.e. Originator) for the sole purpose of saving its state. Since a scaled down version of the Originator is saved instead of the full Originator object, space is saved.
- +++It simplifies the Originator since the responsibility of managing Memento storage is no longer centralized at the Originator but rather distributed among the Caretakers.
- --The Memento object must provide two types of interfaces: a narrow interface to the Caretaker and a wide interface to the Originator. That is, it must acts like a black box to everything except for the class that created it
- --Using Mementos might be expensive if the Originator must store a large portion of its state information in the Memento or if the Caretakers constantly request and return the Mementos to the Originator. Therefore, this pattern should only be used if the benefit of using the pattern is greater than the cost of encapsulation and restoration

pelab

GOF 10th Anniversary

Additional Reading