

# Tecniche di Progettazione: Design Patterns

GoF: Mediator Memento Prototype Visitor

1

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

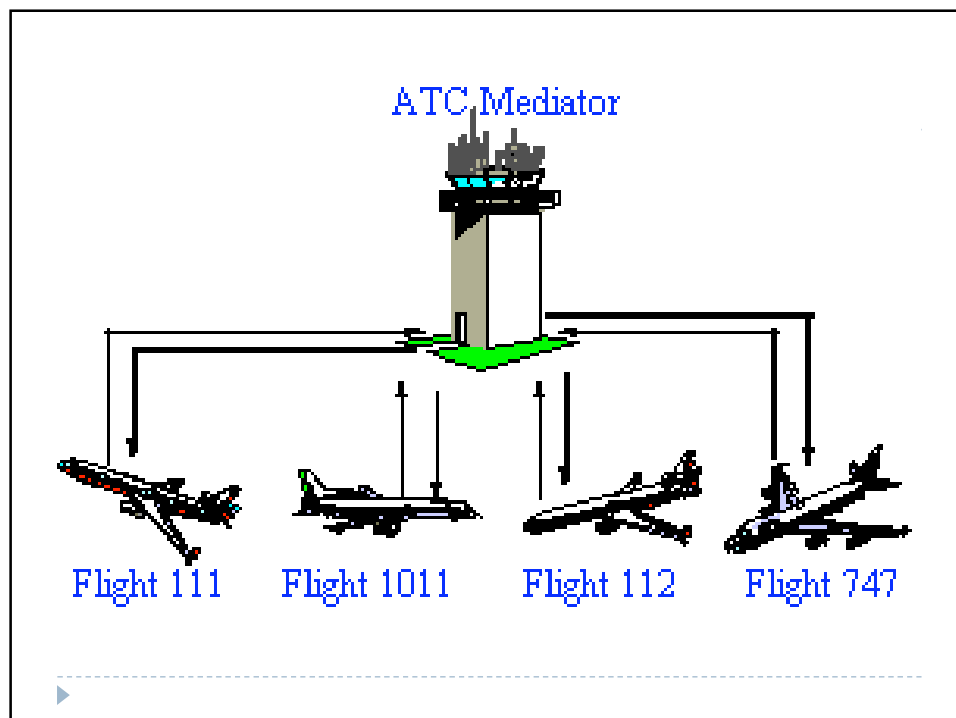
# Mediator

2

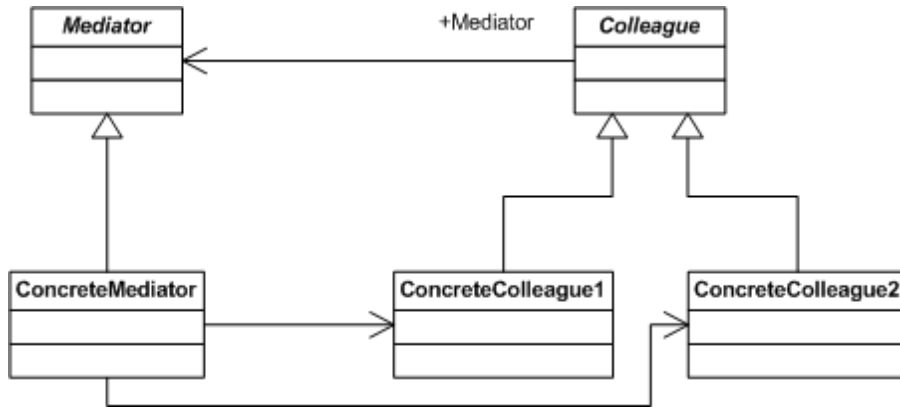
Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

## Applicability

- ▶ A set of objects communicates in well-defined, but complex ways. Often with unstructured dependencies.
- ▶ Reusing objects is difficult because it refers to and communicates with many other objects.
- ▶ A behavior that's distributed between several classes should be customizable without a lot of subclassing.



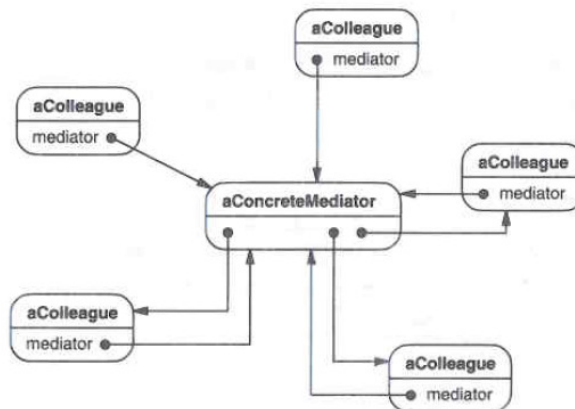
## Mediator: structure



▶ 5

Design patterns, Laura Semini,  
Università di Pisa, Dipartimento di

## Structure



▶

## Consequences

---

- ▶ **Limits subclassing**
    - ▶ Localizes behavior that would be otherwise distributed among many objects
    - ▶ Changes in behavior require changing only the Mediator class
  - ▶ **Decouples colleagues**
    - ▶ Colleagues become more reusable.
    - ▶ You can have multiple types of interactions between colleagues, and you don't need to subclass or otherwise change the colleague class to do that.
- 



## Consequences

---

- ▶ **Simplifies object protocols**
    - ▶ Many-to-many interactions replaced with one-to-many interactions
    - ▶ More intuitive
    - ▶ More extensible
    - ▶ Easier to maintain
  - ▶ **Abstracts object cooperation**
    - ▶ Mediation becomes an object itself
    - ▶ Interaction and individual behaviors are separate concepts that are encapsulated in separate objects
- 



## Consequences

---

- ▶ **Centralizes control**
  - ▶ Mediator can become very complex
  - ▶ With more complex interactions, extensibility and maintenance may become more difficult
  - ▶ Using a mediator may compromise performance



## Implementation Issues

---

- ▶ **Omitting the abstract Mediator class – possible when only one mediator exists**
- ▶ **Strategies for Colleague-Mediator communication**
  - ▶ Observer class
  - ▶ Pointer / other identifier to “self” passed from colleague to mediator, who pass it to the other colleague(s)



## Related Patterns

---

- ▶ **Façade**

- ▶ Unidirectional rather than cooperative interactions between object and subsystem
- ▶ Mediator is like a multi-way Façade pattern.

- ▶ **Observer**

- ▶ May be used as a means of communication between Colleagues and the Mediator



## Coordination

---

- ▶ Linda and tuple spaces

- ▶ BPEL (Business Process Execution Language) and web services (BPEL4WS o WS-BPEL)

# Memento

13

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

## Memento

---

### ▶ Intent

- ▶ “Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.”

### ▶ Motivation

- ▶ When we want to store off an object’s internal state without adding any complication to the object’s interface.
- ▶ Perhaps for an undo mechanism

---

▶

## Memento pattern

---

### ▶ Memento:

- ▶ a saved "snapshot" of the state of an object or objects for possible later use
- ▶ useful for:
  - ▶ writing an Undo / Redo operation
  - ▶ ensuring consistent state in a network
  - ▶ Persistency: save / load state between executions of program



15

## Applicability

---

### ▶ Use this

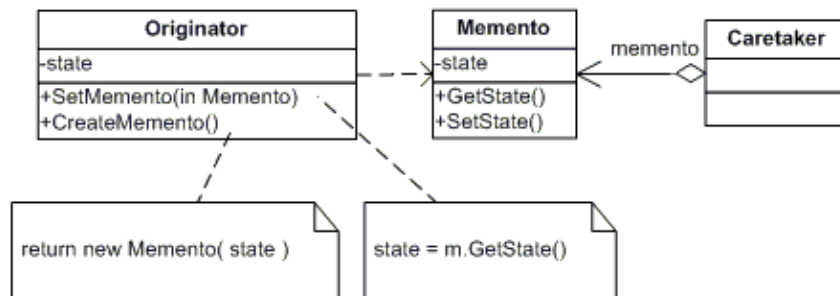
- ▶ when you want to save state on a hierarchy's elements.
- ▶ When the hierarchy's interface would be broken if implementation details were exposed.





## Structure

---



## Participants

---

- ▶ **Memento**
  - ▶ stores the state of the Originator
- ▶ **Originator**
  - ▶ Creates the memento
  - ▶ “Uses the memento to restore its internal state”
- ▶ **CareTaker**
  - ▶ Keeps track of the Memento
  - ▶ Never uses the Memento’s Interface to the Originator

## Collaboration

---

- ▶ Caretaker requests a memento from an Originator.
- ▶ Originator passes back memento.
- ▶ Originator uses it to restore state.



## Consequences (good)

---

- ▶ “Preserves Encapsulation Boundaries”
- ▶ “It simplifies Originator”



## Consequences (bad)

---

- ▶ Might be expensive
- ▶ Difficulty defining interfaces to keep Originator encapsulated
- ▶ Hidden costs in caring for mementos
  - ▶ Caretaker could have to keep track of a lot of information for the memento



## Storing Incremental Changes

---

- ▶ If storing state happens incrementally, then we can just record the changes of what's happened in a new memento object.
- ▶ This helps with memory difficulties.



Notes Practice

# Prototype

23 Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

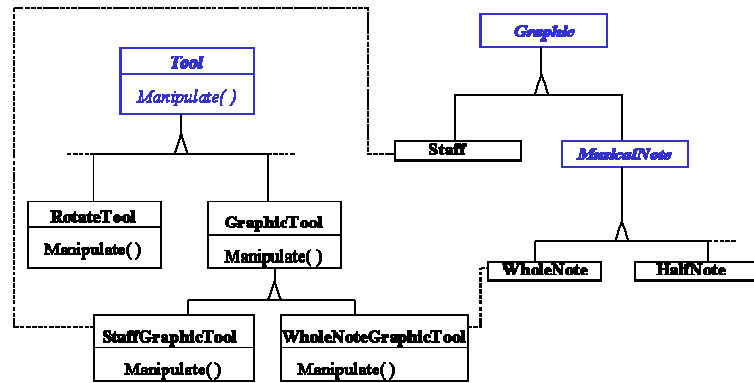
## Prototype Pattern

---

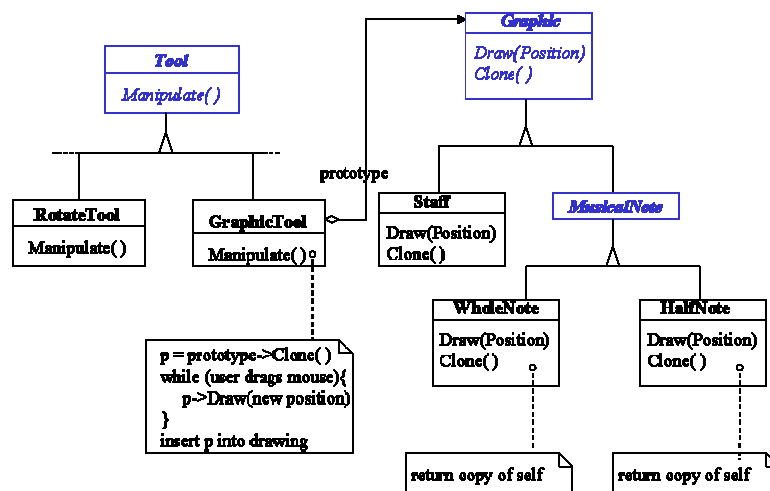
- ▶ A creational pattern
- ▶ Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype



## Problem



## Prototype solution

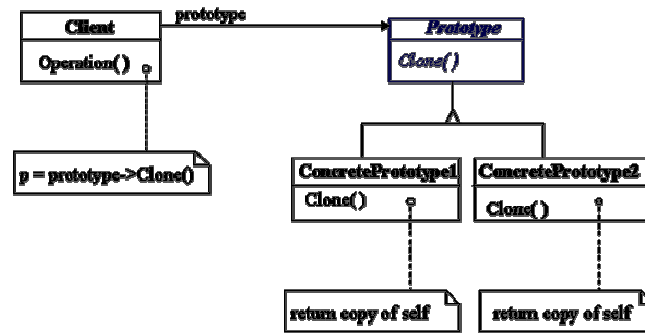


## Structure & Participants

**Prototype(Graphic)**  
-declares an interface  
for cloning itself.

**ConcretePrototype**  
(Staff,WholeNote,  
HalfNote)  
-implements an  
operation for cloning  
itself.

**Client(GraphicalTo  
ol)**  
- creates a new  
object by asking a  
prototype to clone  
itself.



java.lang Class Object  
protected Object **clone()** throws  
CloneNotSupportedException

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object *x*, the expression:

`x.clone() != x`

will be true, and that the expression:

`x.clone().getClass() == x.getClass()`

will be true, but these are not absolute requirements. While it is typically the case that:

`x.clone().equals(x)`

will be true, this is not an absolute requirement.

By convention, the returned object should be obtained by calling `super.clone`.

If a class and all of its superclasses (except `Object`) obey this convention, it will be the case that `x.clone().getClass() == x.getClass()`.

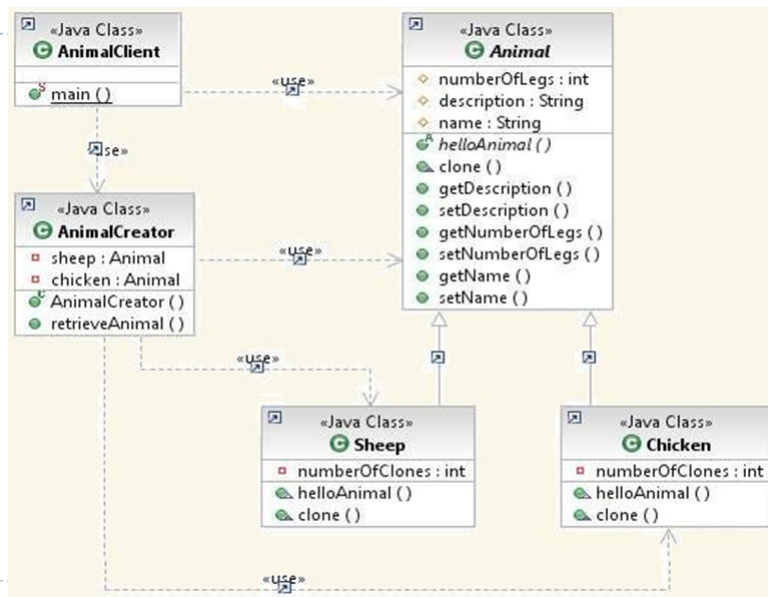
java.lang Class Object  
 protected Object **clone()** throws  
 CloneNotSupportedException

- ▶ By convention, the object returned by this method should be independent of this object (which is being cloned).
- ▶ To achieve this independence, it may be necessary to modify one or more fields of the object returned by super.clone before returning it.
  - ▶ Typically, this means copying any mutable objects that comprise the internal "deep structure" of the object being cloned and replacing the references to these objects with references to the copies.
  - ▶ If a class contains only primitive fields or references to immutable objects, then it is usually the case that no fields in the object returned by super.clone need to be modified.

▶ 29

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

## Ex. Animal farm



## Prototype Pattern Example code

---

```

public abstract class Animal implements Cloneable {
    protected int numberOfLegs = 0;
    protected String description = "";
    protected String name = "";

    public abstract String helloAnimal();

    public Animal clone() {
        Animal clonedAnimal = null;
        clonedAnimal = (Animal) super.clone();
        clonedAnimal.setName(name);
        return clonedAnimal;
    } // method clone

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
} // class Animal

```

---

## Prototype Pattern Example code

---

```

public class Chicken extends Animal {
    private int numberOfClones = 0;

    public String helloAnimal() {
        StringBuffer chickenTalk = new StringBuffer();
        chickenTalk.append("cluck cluck world. I am ");
        chickenTalk.append(name);
        return chickenTalk.toString();
    } // helloAnimal

    public Chicken clone() {
        Chicken clonedChicken = (Chicken) super.clone();
        String chickenName = clonedChicken.getName();
        numberOfClones++;
        clonedChicken.setName(chickenName + numberOfClones);
        return clonedChicken;
    } // method clone
}

```

---



## Prototype Pattern Example code

---

```
public class sheep extends Animal {
    private int numberOfClones = 0;

    public String helloAnimal() {
        StringBuffer sheepTalk = new StringBuffer();
        sheepTalk.append("Meeeeeee world. I am ");
        sheepTalk.append(name);
        return sheepTalk.toString();
    } // helloAnimal

    public Sheep clone() {
        Sheep clonedSheep = (Sheep) super.clone();
        String sheepName = clonedSheep.getName();
        numberOfClones++;
        clonedSheep.setName(sheepName + numberOfClones);
        return clonedSheep;
    } // method clone
}
```

---

## Prototype Pattern Example code

---

```
public class AnimalCreator {
    private Animal sheep = new Sheep();
    private Animal chicken = new Chicken();

    public AnimalCreator() {
        sheep.setName("sheep");
        chicken.setName("chicken");
    } // no-arg constructor

    public Animal retrieveAnimal(String kindOfAnimal) {
        if ("chicken".equals(kindOfAnimal)) {
            return (Animal) chicken.clone();
        }
        else if ("sheep".equals(kindOfAnimal)) {
            return (Animal) sheep.clone();
        } // if
        return null;
    } // method retrieveAnimal
} // class AnimalCreator
```

---

## Prototype Pattern Example code

---

```

public class AnimalClient {
    public static void main(String[] args) {
        AnimalCreator animalCreator = new AnimalCreator();
        Animal[] animalFarm = new Animal[8];

        animalFarm[0] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[1] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[2] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[3] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[4] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[5] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[6] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[7] = animalCreator.retrieveAnimal("Sheep");

        for (int i= 0; i<=7; i++) {
            System.out.println(animalFarm[i].helloAnimal());
        } // for
    } // main method
} // class AnimalClient

Cluck cluck world. I am Chicken1.
Cluck cluck world. I am Chicken2.
Cluck cluck world. I am Chicken3.
Cluck cluck world. I am Chicken4.
Meeeeeeee world. I am Sheep1.
Meeeeeeee world. I am Sheep2.
Meeeeeeee world. I am Sheep3.
Meeeeeeee world. I am Sheep4.

```

## Prototype Pattern

---

- ▶ **When to Use**
  - ▶ When product creation should be decoupled from system behavior
  - ▶ When to avoid subclasses of an object creator in the client application
  - ▶ When creating an instance of a class is time-consuming or complex in some way.

## Consequences of Prototype Pattern

---

- ▶ Hides the concrete product classes from the client
- ▶ Adding/removing of prototypes at run-time
- ▶ Allows specifying new objects by varying values or structure
- ▶ Reducing the need for sub-classing

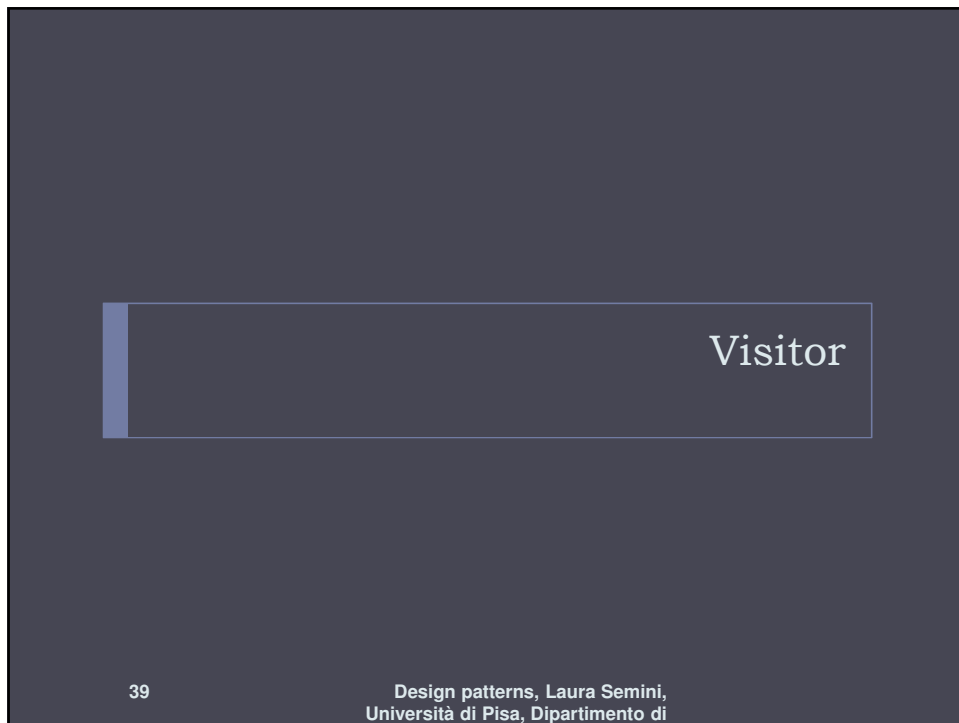


## Drawbacks of Prototype Pattern

---

- ▶ It is built on the method `.clone()`, which could be complicated sometimes in terms of shallow copy and deep copy. Moreover, classes that have circular references to other classes cannot really be cloned.





## Visitor Pattern

---

### ▶ Intent

- ▶ Lets you define a new operation without changing the classes on which they operate.

### ▶ Motivation

- ▶ Allows for increased functionality of a class(es) while streamlining base classes.
- ▶ A primary goal of designs should be to ensure that base classes maintain a minimal set of operations.
- ▶ Encapsulates common functionality in a class framework.



## Visitor Pattern

---

### ▶ Motivation (cont)

- ▶ Visitors avoid type casting that is required by methods that pass base class pointers as arguments. The following code describes how a typical class could expand the functionality of an existing composite.

```

Void MyAddition::execute( Base* basePtr) {
    if( dynamic_cast<ChildA*>(basePtr)){
        // Perform task for child type A.
    } else if ( dynamic_cast<ChildB*>(basePtr)){
        // Perform task for child type B.
    } else if( dynamic_cast<ChildC*>(basePtr)){
        // Perform task for child type C.
    }
}

```

---



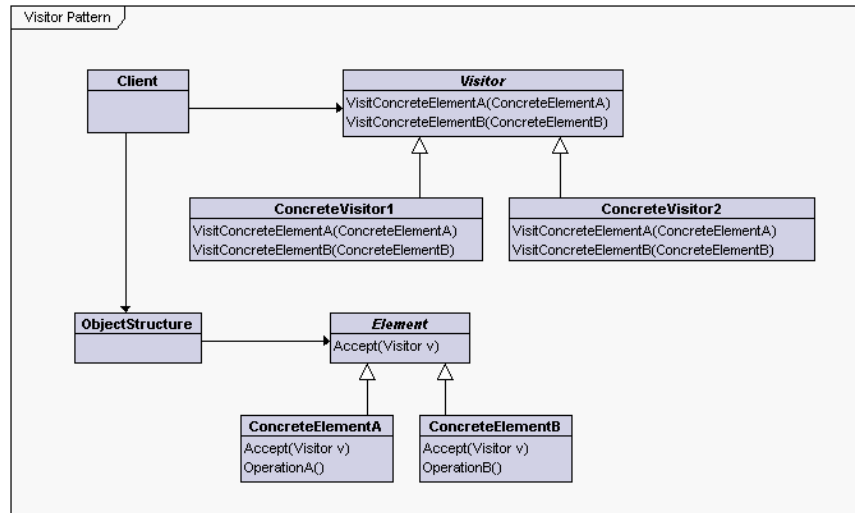
## Visitor Pattern: Applicability

---

- ▶ The following situations are prime examples for use of the visitor pattern.
    - ▶ When an object structure contains many classes of objects with different interfaces and you want to perform functions on these objects that depend on their concrete classes.
    - ▶ When you want to keep related operations together by defining them in one class.
    - ▶ When the class structure rarely change but you need to define new operations on the structure.
- 



## Structure



▶ 43

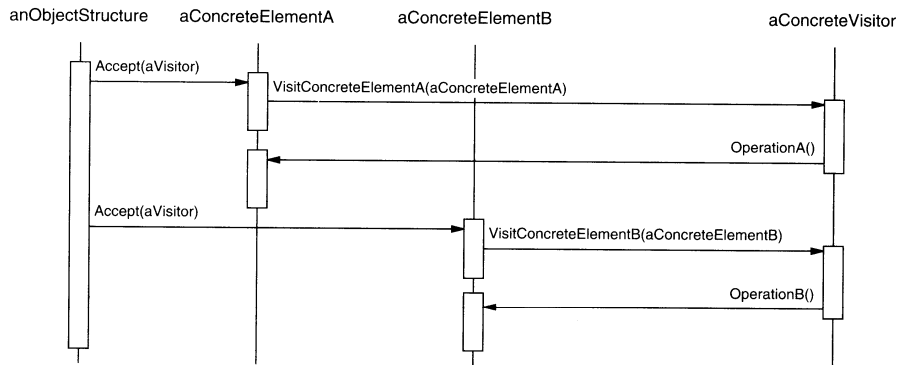
Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

## Visitor Pattern: Participants

- ▶ **Visitor**
  - ▶ Declares a Visit Operation for each class of Concrete Elements in the object structure.
- ▶ **Concrete Visitor**
  - ▶ Implements each operation declared by Visitor.
- ▶ **Element**
  - ▶ Defines an Accept operation that takes the visitor as an argument.
- ▶ **Concrete Element**
  - ▶ Implements an accept operation that takes the visitor as an argument.
- ▶ **Object Structure**
  - ▶ Can enumerate its elements.
  - ▶ May provide a high level interface to all the visitor to visit its elements.
  - ▶ May either be a composite or a collection.

▶

## Visitor Pattern: Collaborations



## Visitor Pattern: Consequences

- ▶ Makes adding new operations easier.
  - ▶ Collects related functionality.
  - ▶ Adding new Concrete Element classes is difficult.
  - ▶ Can “visit” across class types, unlike iterators.
  - ▶ Accumulates states as they visit elements.
  - ▶ May require breaking object encapsulation to support the implementation.
- The diagram ends with a blue arrow pointing to the right.

## Visitor: Related Patterns

---

- ▶ **Composites**
  - ▶ Visitors can be used to apply an operation over an object structure defined by the composite pattern.
- ▶ **Interpreter**
  - ▶ Visitors may be applied to do the interpretation.

