

Laboratorio di Progettazione di Sistemi Software

Model-View-Controller architecture pattern

Valentina Presutti (A-L)
Riccardo Solmi (M-Z)

Index

- **Architectural pattern:**
 - Model View Controller
- **Design pattern catalogue:**
 - Observer
 - Command
 - Adapter
 - Strategy

Model View Controller

- **The MVC is an architectural pattern**
 - Participants are set of classes
- **Intent**
 - Building an interactive application partitioning into three components the responsibilities for managing the model, the representation and handling user inputs.
 - MVC ensures consistency between the user interface and the model.

© 2002-2004 Riccardo Solmi

3

Model View Controller/3

- **Applicability**
 - The same application data needs to be accessed when presented in *different views*: e.g. HTML, WML, JFC/Swing, XML, Java source, Java outline
 - The same application data needs to be updated through *different interactions*: e.g. link selections on an HTML page or WML card, button clicks on a JFC/Swing GUI, SOAP messages written in XML
 - Supporting multiple types of views and interactions should *not impact* the components that provide the *core functionality (model)* of the application

© 2002-2004 Riccardo Solmi

4

Model View Controller/4

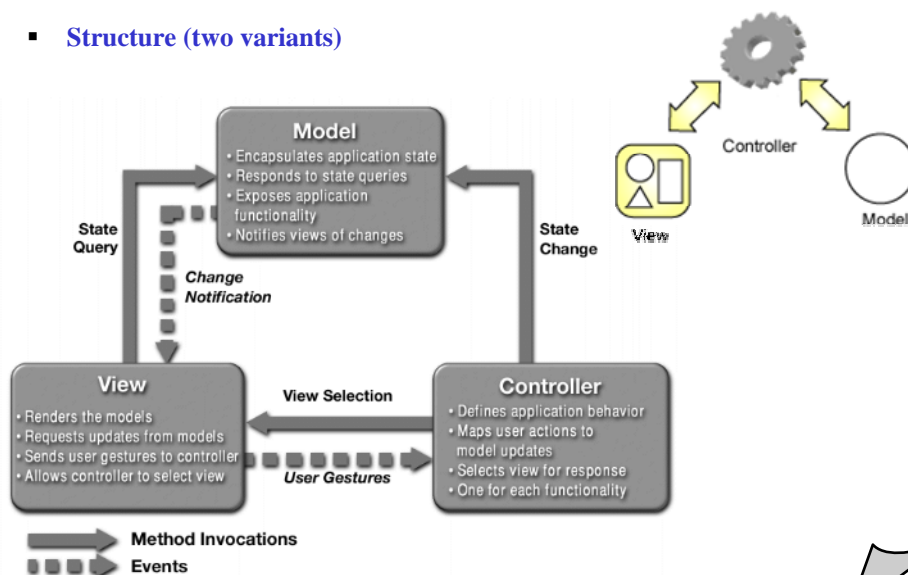
- **Solution**
- By applying the Model-View-Controller architecture you separate core business model functionality from the presentation and control logic that uses this functionality.
- This separation allows multiple views to share the same application data model, which makes supporting multiple clients easier to implement, test, and maintain.

© 2002-2004 Riccardo Solmi

5

Model View Controller/5

- **Structure (two variants)**



© 2002-2004 Riccardo Solmi

6

Model View Controller/6

▪ **Participants and Responsibilities**

- **Model:** the model represents the application data and the the rules that govern access to and updates of this data.
- **View:** the view renders the contents of a model.
- **Controller:** The controller translates interactions with the view into actions to be performed by the model. The actions performed by the model include changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.
- Getting the application data from the model and updating the view when the model changes can be a responsibility of the or (better) of the model.

© 2002-2004 Riccardo Solmi

7

Model View Controller/7

Consequences

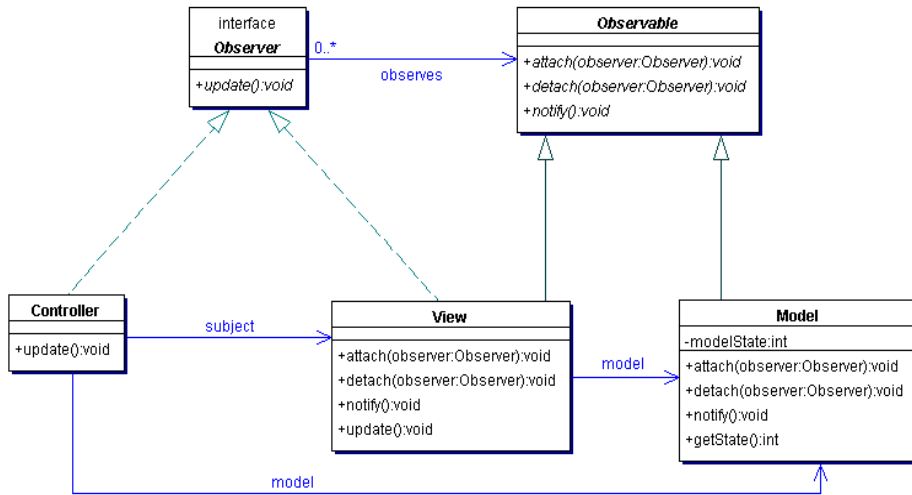
- Re-use of Model components.
- Easier support for new types of clients.
- Increased design complexity.

© 2002-2004 Riccardo Solmi

8

Model View Controller/8

- **Collaborations**

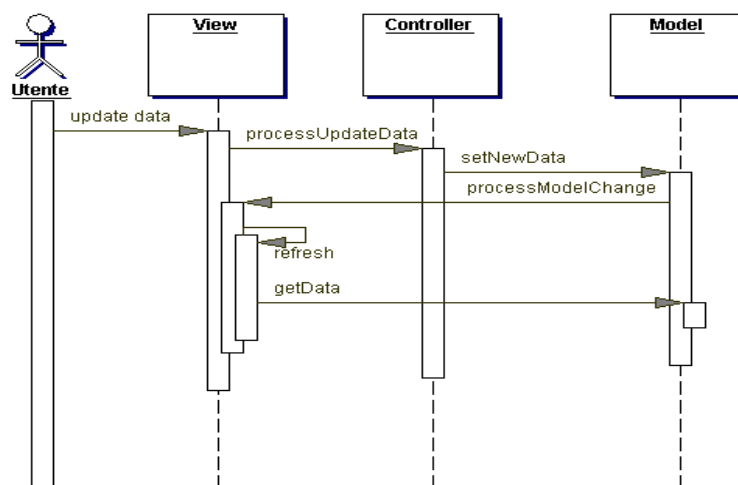


© 2002-2004 Riccardo Solmi

9

Model View Controller/9

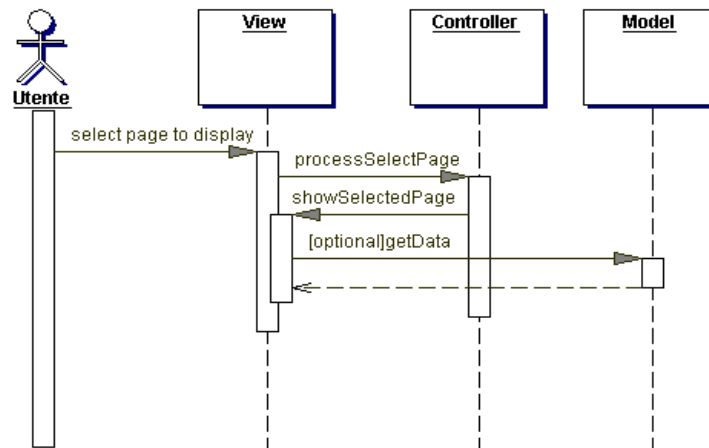
- **The following diagram shows the interactions between objects in an updating scenario**



10

Model View Controller/10

- The following diagram shows the interactions between objects in a scenario of page selection



© 2002-2004 Riccardo Solmi

11

Observer

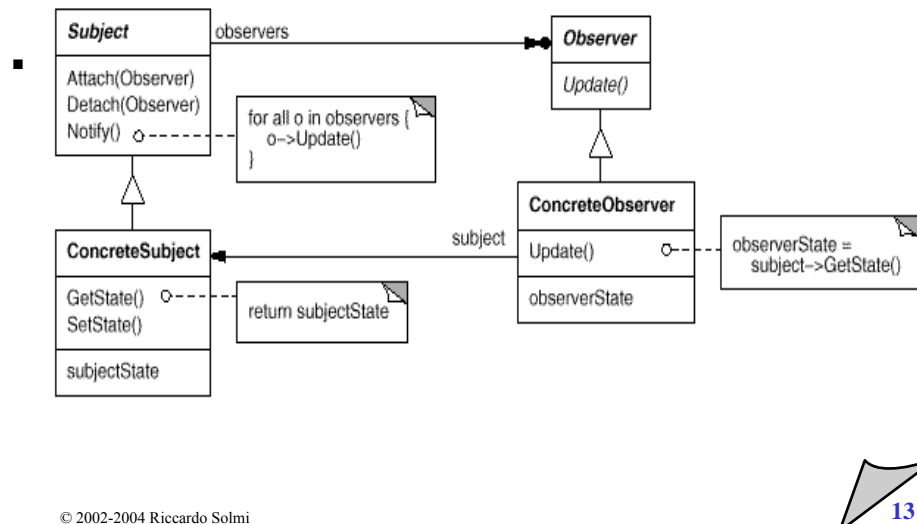
- **Intent**
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Applicability**
 - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - When a change to one object requires changing others, and you don't know how many objects need to be changed.
 - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

© 2002-2004 Riccardo Solmi

12

Observer/2

Structure



13

Observer/3

Participants

- **Subject**: knows its observers. Any number of Observer objects may observe a subject. provides an interface for attaching and detaching Observer objects.
- **Observer**: defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**: stores state of interest to ConcreteObserver objects and sends a notification to its observers when its state changes.
- **ConcreteObserver**: maintains a reference to a ConcreteSubject object, stores state that should stay consistent with the subject's and implements the Observer updating interface to keep its state consistent with the subject's.

© 2002-2004 Riccardo Solmi

14

Observer/4

▪ Collaborations

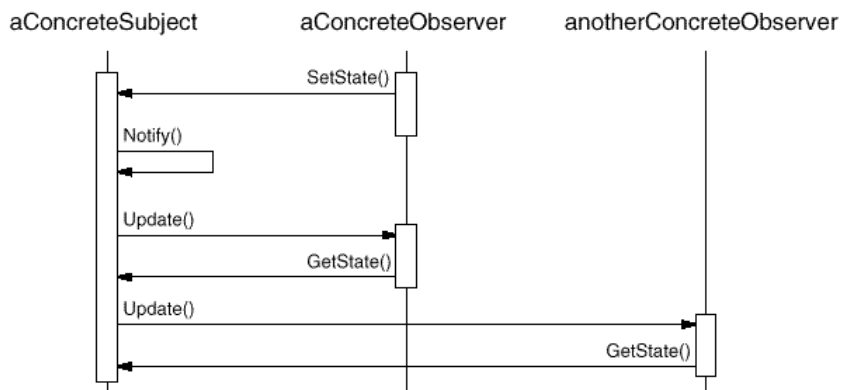
- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

© 2002-2004 Riccardo Solmi

15

Observer/5

- **The following interaction diagram illustrates the collaborations between a subject and two observers:**



© 2002-2004 Riccardo Solmi

16

Observer/6

- **Consequences**

- The Observer pattern lets you vary subjects and observers independently.
- You can reuse subjects without reusing their observers, and vice versa.
- It lets you add observers without modifying the subject or other observers.
- Abstract coupling between Subject and Observer.
- Support for broadcast communication.
- Unexpected updates.

© 2002-2004 Riccardo Solmi

17

Command

- **Intent**

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

- **Applicability**

- parameterize objects by an action to perform. Commands are an object-oriented replacement for callbacks.
- specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
- support undo. The Command's Execute operation can store state for reversing its effects in the command itself.

© 2002-2004 Riccardo Solmi

18

Command/2

Structure

Participants

- **Command**
 - declares an interface for executing an operation
- **ConcreteCommand:**
 - defines a binding between a Receiver object and an action
 - Implements Execute by invoking the corresponding operation on Receiver
- **Client:**
 - Creates a ConcreteCommand object and sets its receiver
- **Invoker:**
 - Asks the command to carry out the request
- **Receiver:**
 - Knows how to perform the operations associated with carrying out a request. Any class may serve as a receiver

© 2002-2004 Riccardo Solmi

19

Command/3

Collaborations

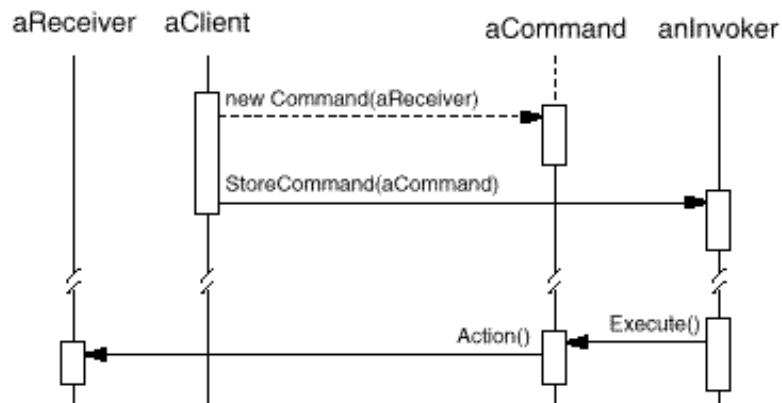
- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.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

© 2002-2004 Riccardo Solmi

20

Command/4

- **Collaborations (continue)**
- The following diagram shows the interactions between these objects



© 2002-2004 Riccardo Solmi

21

Command/5

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

© 2002-2004 Riccardo Solmi

22

Adapter

- **Intent**

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- **Applicability**

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (*object adapter only*) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

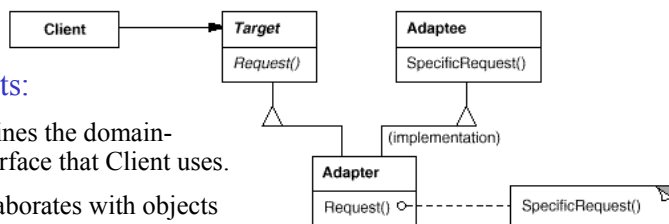
© 2002-2004 Riccardo Solmi

23

Adapter/2

- **Structure**

- **A class adapter uses multiple inheritance to adapt one interface to another:**



- **Participants:**

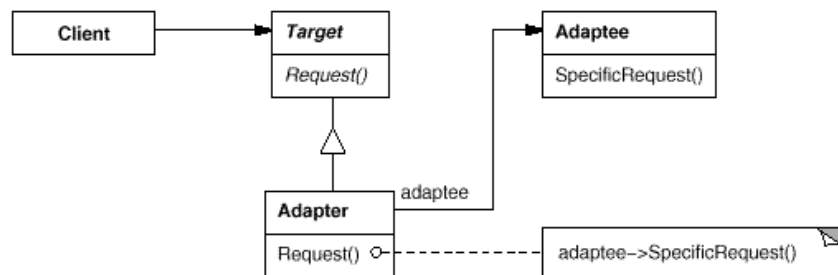
- **Target:** defines the domain-specific interface that Client uses.
- **Client:** collaborates with objects conforming to the Target interface.
- **Adaptee:** defines interface that needs adapting.
- **Adapter:** adapts the interface of Adaptee to the Target interface.

© 2002-2004 Riccardo Solmi

24

Adapter/2

- **Structure**
- **An object adapter relies on object composition:**



© 2002-2004 Riccardo Solmi

25

Adapter/4

- **Consequences**
- **Class and object adapters have different trade-offs.**
- **A class adapter**
 - adapts Adaptee to Target by committing to a concrete Adapter class. A class adapter won't work when we want to adapt a class *and* all its subclasses.
 - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
 - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- **An object adapter**
 - lets a single Adapter work with many Adaptees, that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
 - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

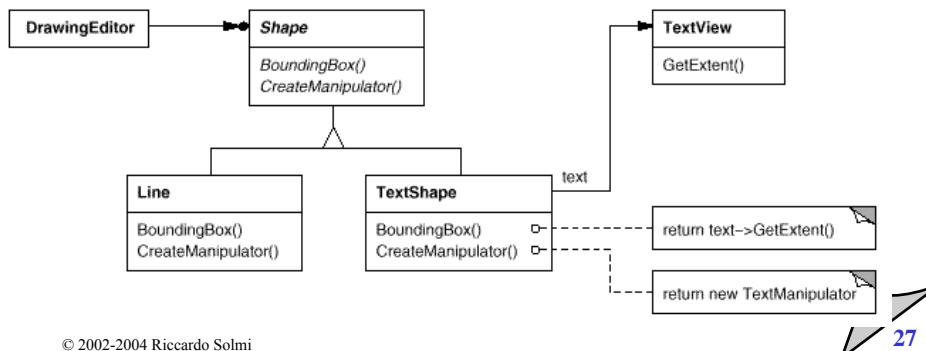
© 2002-2004 Riccardo Solmi

26

Adapter/5

▪ Example

- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires. Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. An user interface toolkit might already provide a sophisticated `TextView` class for displaying and editing text.



© 2002-2004 Riccardo Solmi

27

Strategy

▪ Intent

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

▪ Applicability

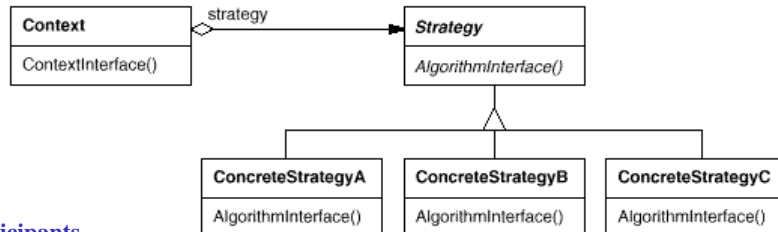
- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about.
- Instead of many conditionals.

© 2002-2004 Riccardo Solmi

28

Strategy /2

Structure



Participants

- Strategy
 - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy
 - implements the algorithm using the Strategy interface
- Context
 - is configured with a ConcreteStrategy object.
 - maintains a reference to a Strategy object.
 - may define an interface that lets Strategy access its data

© 2002-2004 Riccardo Solmi

29

Strategy /3

Collaborations

- Strategy and Context interact to implement the chosen algorithm. A context may pass data or itself to the Strategy.
- A context forwards requests from its clients to its strategy.
- Clients usually create and pass a ConcreteStrategy to the context; thereafter, they interact with the context exclusively.

Consequences

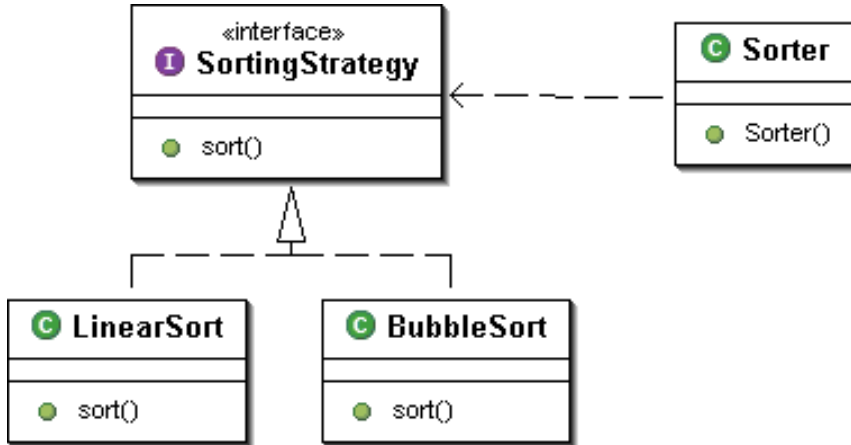
- Families of related algorithms.
- An alternative to subclassing. Vary the algorithm independently of its context even *dynamically*.
- Strategies eliminate conditional statements.
- A choice of implementations (space/time trade-offs).
- Clients must be aware of different Strategies.
- Communication overhead between Strategy and Context.
- Increased number of objects (*stateless* option).

© 2002-2004 Riccardo Solmi

30

Strategy example 1

- **Sorting Strategy**

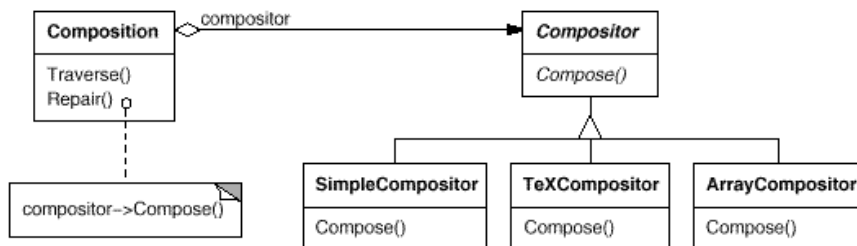


© 2002-2004 Riccardo Solmi

31

Strategy example 2

- **To define different algorithms**



© 2002-2004 Riccardo Solmi

32

Strategy questions

- **What happens when a system has an explosion of Strategy objects? Is there some way to better manage these strategies?**
- **Is it possible that the data required by the strategy will not be available from the context's interface? How could you remedy this potential problem?**