Università degli Studi di Napoli “Federico II”

Ingegneria del Software

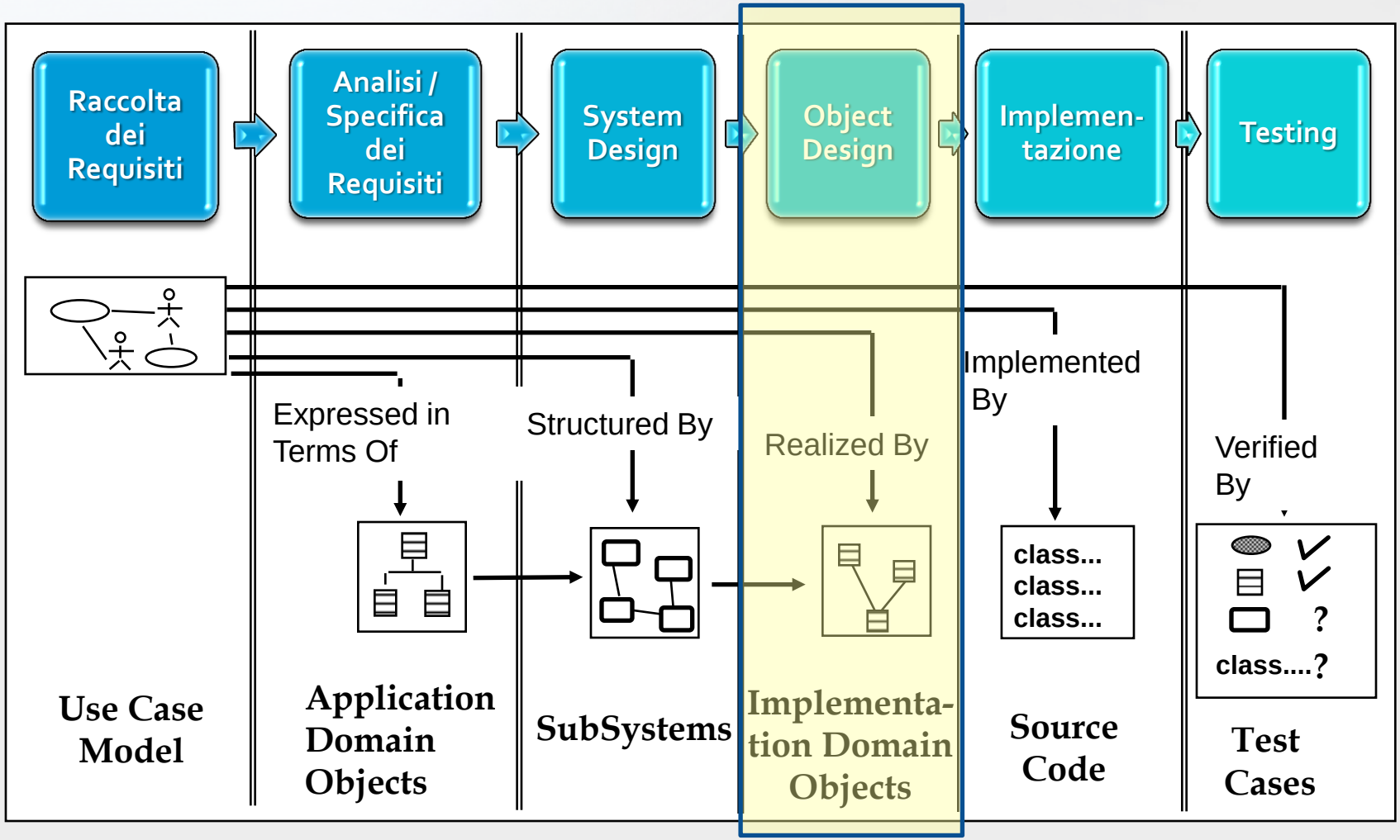
a.a. 2012/13

Object Design - I Design Patterns

Obiettivi della lezione

- Comprendere le attività dell'Object Design
  - I Design Patterns
    - Singleton
    - MVC

Software Lifecycle Activities



Object Design

- L'Object Design è la fase nel ciclo di vita del software in cui si definiscono le scelte finali prima dell'implementazione
- In questa fase, l'analista deve scegliere tra i differenti modi di implementare i modelli di analisi, rispettando requisiti non funzionali e criteri di design.
  - Si specificano quali classi implementeranno le funzionalità descritte in analisi, come comincheranno tra loro, etc…
- E' il passo finale prima dell'implementazione

Goal dell'Object Design

- Identification of existing components
- Full definition of relations
- Full definition of classes (System Design => Service, Object Design => API)
- Specifying the contract for each component
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Detection of solution-domain classes
- Optimization
- Increase of inheritance
- Decision on control
- Packaging

Architetture e Design Patterns

- Le Architetture sono un modo per definire la struttura di un'applicazione ad alto livello
  - Definiamo aggregazioni di (molte) classi in ogni modulo che compone l'architettura
  - Grazie alle architetture già definite, abbiamo una vasta scelta di soluzioni largamente testate per una grande classe di problemi
- Esiste qualcosa simile a livello più basso?
- Ci sono modi "standard" di combinare classi tra loro per svolgere funzionalità tipiche?

  → Design Patterns!

Re-use

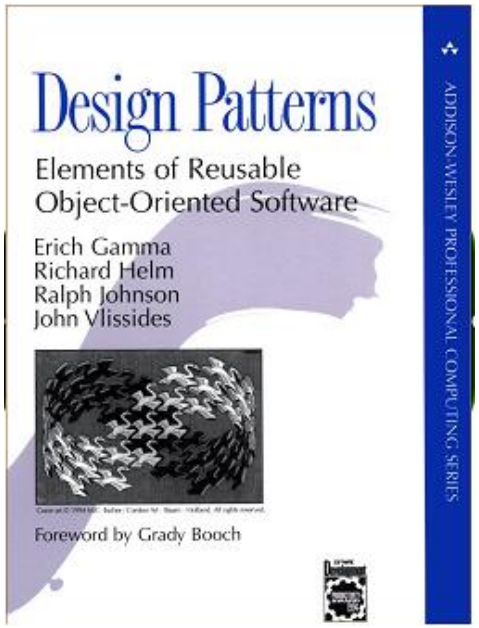
- Code re-use
  - Don’t reinvent the wheel
  - Requires clean, elegant, understandable, general, stable code
  - leverage previous work
- Design re-use
  - Don’t reinvent the wheel
  - Requires a precise understanding of common, recurring designs
  - leverage previous work

Motivation and Concept

- OO systems exploit recurring design structures that promote
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance
- Therein lies valuable design knowledge
- Problem: capturing, communicating, and applying this knowledge for re-use

What Is a Design Pattern?

- A design pattern
  - Is a common solution to a recurring problem in design
  - Abstracts a recurring design structure
  - Comprises class and/or object
    - Dependencies
    - Structures
    - Interactions
    - Conventions
  - Names & specifies the design structure explicitly
  - Distils design experience



History of Design Patterns

- Architect Christopher Alexander
  - A Pattern Language (1977)
  - A Timeless Way of Building (1979)
- "Gang of four"
    - Erich Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides
  - The Book: "*Design Patterns: Elements of Reusable Object-Oriented Software*" (1995)
- Many since
  - Conferences, symposia, books

What Is a Design Pattern?

- A design pattern has 4 basic parts:
  - 1. Name
  - 2. Problem
  - 3. Solution
  - 4. Consequences and trade-offs of application
- Language- and implementation-independent
- A “micro-architecture”
- No mechanical application
  - The solution needs to be translated into concrete terms in the application context by the developer

Goals

- Codify good design
  - Distil and disseminate experience
  - Aid to novices and experts alike
  - Abstract how to think about design
- Give design structures explicit names
  - Common vocabulary
  - Reduced complexity
  - Greater expressiveness
- Capture and preserve design information
  - Articulate design decisions succinctly
  - Improve documentation
- Facilitate restructuring/refactoring
  - Patterns are interrelated
  - Additional flexibility

Design Pattern Catalogues

- GoF ("the Gang of Four") catalogue
  - "Design Patterns: Elements of Reusable Object-Oriented Software," Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995
- POSA catalogue
  - Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996
- …

Classification of GoF Design Pattern

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Singleton

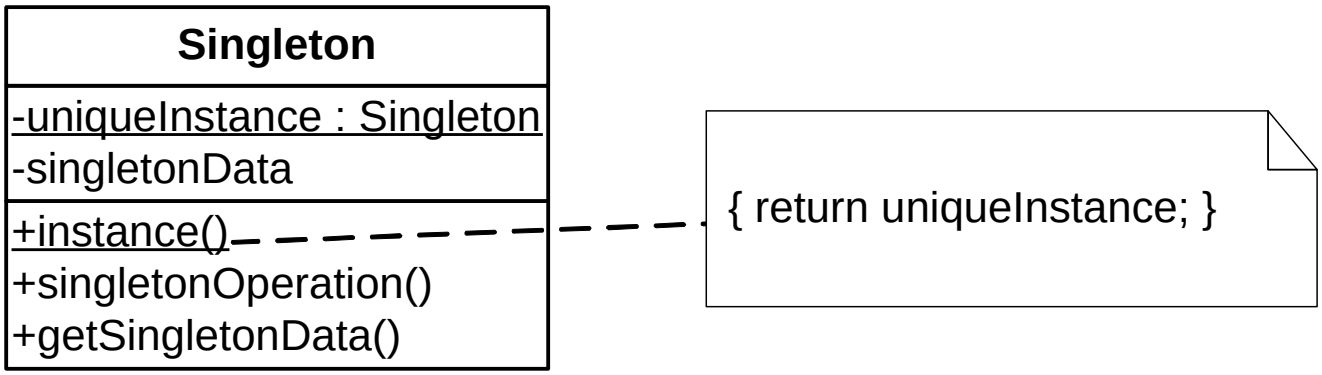
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Singleton (Creational)

- Intent
  - Ensure a class only ever has one instance, and provide a global point of access to it.
- Applicability
  - When there must be exactly one instance of a class, and it must be accessible from a well-known access point
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton (cont'd)

- Structure



Singleton (cont'd)

- Consequences
  - Reduces namespace pollution (no global variables)
  - Makes it easy to change your mind and allow more than one instance
  - Allow extension by subclassing
  - Same drawbacks of a global if misused
  - Implementation may be less efficient than a global
  - Concurrency pitfalls
- Implementation
  - Static instance operation
  - Registering the singleton instance

Singleton - Example

```
public class SingolaIstanza {
    private static SingolaIstanza istanza = null;

    private SingolaIstanza () {}

    public static SingolaIstanza getInstance()
    {
        if (istanza == null)
            istanza = new SingolaIstanza ();
        return istanza;
    }
}
```

- Although the above example uses a single instance, modifications to the function Instance() may permit a variable number of instances.
  - For example, you can design a class that allows up to three instances.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

The MVC pattern

- MVC stands for Model-View-Controller
  - The Model is the actual internal representation of the "knowledge"
  - The View (or a View) is a way of looking at or displaying the model
  - The Controller provides for user input and modification
- These three components are usually implemented as separate classes

The Model

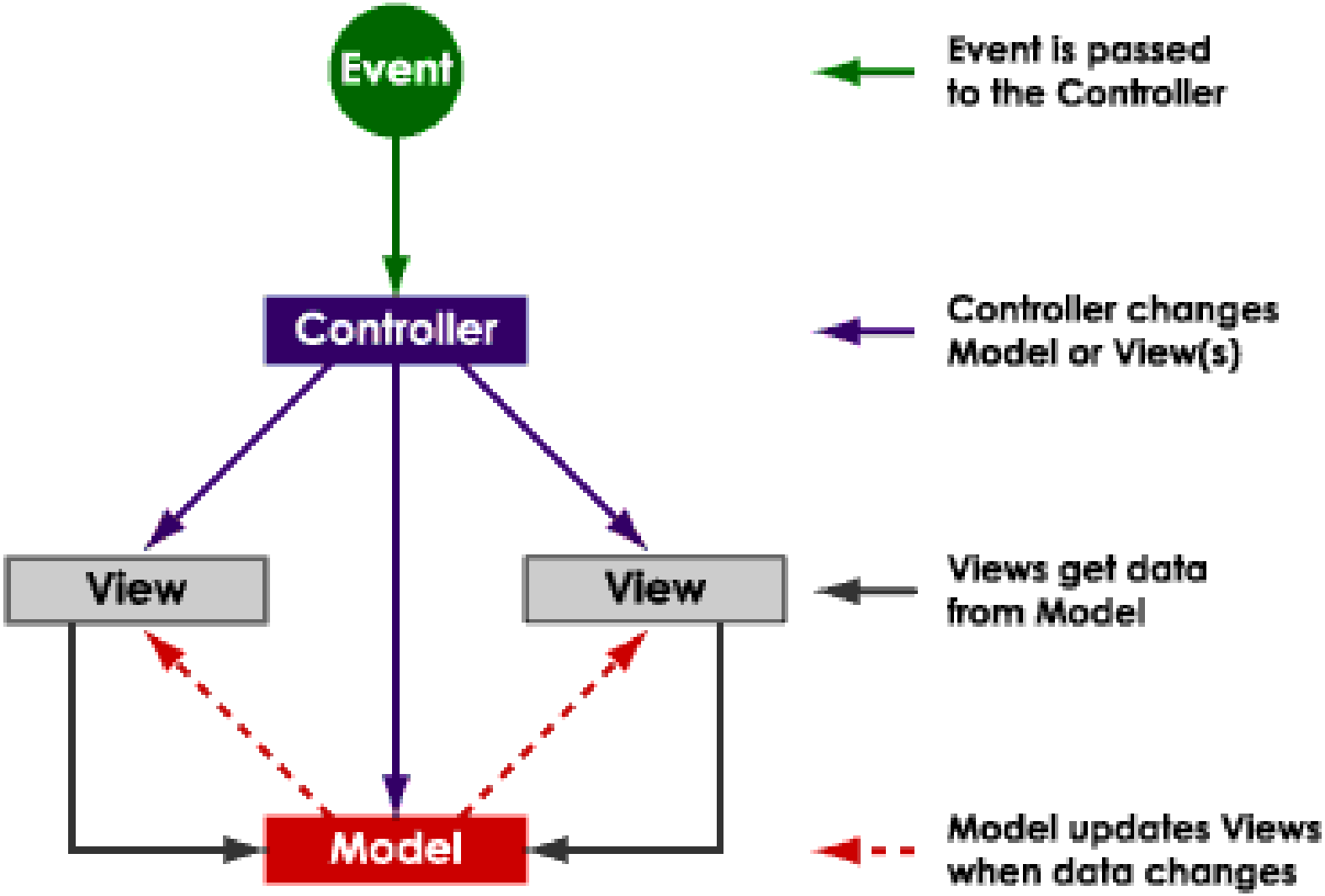
- Most programs are supposed to do work, not just be "another pretty face"
  - but there are some exceptions
  - useful programs existed long before GUIs
- The Model is the part that does the work--it *models* the actual problem being solved
- **The Model should be independent of both the Controller and the View**
  - **But it provides services (methods) for them to use**
- Independence gives flexibility, robustness

The Controller

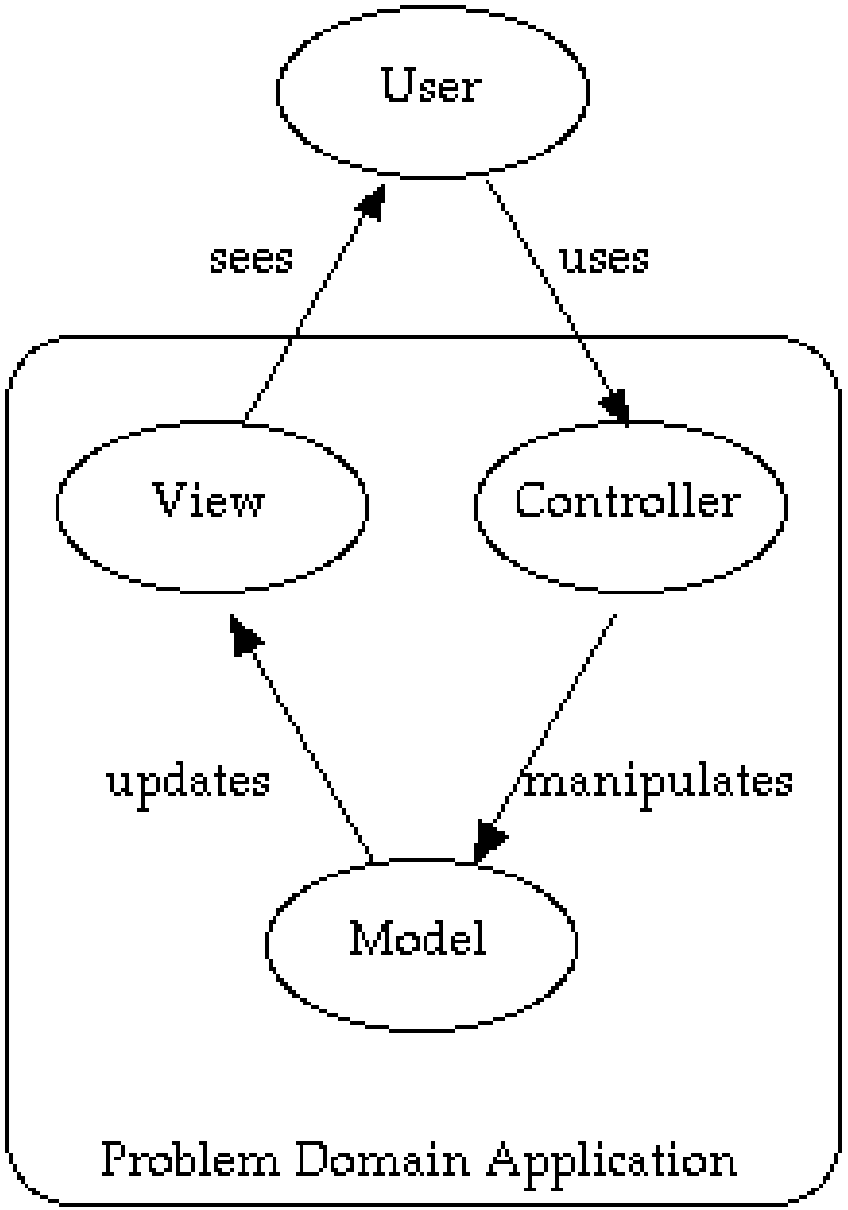
- The Controller decides what the model is to do
- Often, the user is put in control by means of a GUI
  - in this case, the GUI and the Controller are often the same
- The Controller and the Model can almost always be separated (what to do versus how to do it)
- The design of the Controller depends on the Model

The View

- Typically, the user has to be able to see, or view, what the program is doing
- The View shows what the Model is doing
  - The View is a passive observer; it should not affect the model
- The Model should be independent of the View, but (but it can provide access methods)
- The View should not display what the Controller thinks is happening



Events typically cause a controller to change a model, or view, or both. Whenever a controller changes a model's data or properties, all dependent views are automatically updated. Similarly, whenever a controller changes a view, the view gets data from the underlying model to refresh itself.


User
sees
uses
View
Controller
updates
manipulates
Model
Problem Domain Application

Combining Controller and View

- Sometimes the Controller and View are combined, especially in small programs
- Combining the Controller and View is appropriate if they are very interdependent
- The Model should still be independent
- *Never* mix Model code with GUI code!

Separation of concerns

- As always, you want code independence
- The Model should not be contaminated with control code or display code
- The View should represent the Model as it really is, not some remembered status
- The Controller should *talk to* the Model and View, not *manipulate* them
  - The Controller can set variables that the Model and View can read

The “Reverser” program

- In this program we combine the Controller and the View (indeed, it’s hard to separate them)
- The Model, which does the computation (reversing the string), we put in a separate class

ReverserGUI.java

- A bunch of import statements, then...

```
public class ReverserGUI extends JFrame {
    ReverserModel model = new ReverserModel();
    JTextField text = new JTextField(30);
    JButton button = new JButton("Reverse");

    public static void main(String[] args) {
        ReverserGUI gui = new ReverserGUI();
        gui.create();
        gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    ... create()...
}
```

The create method

```
private void create() {
    setLayout(new GridLayout(2, 1));
    add(text);
    add(button);

    button.addActionListener(new ActionListener()  {
        public void actionPerformed(ActionEvent arg0) {
            String s = text.getText();
            s = model.reverse(s);
            text.setText(s);
        }
    });
    pack();
    setVisible(true);
}
```

The model

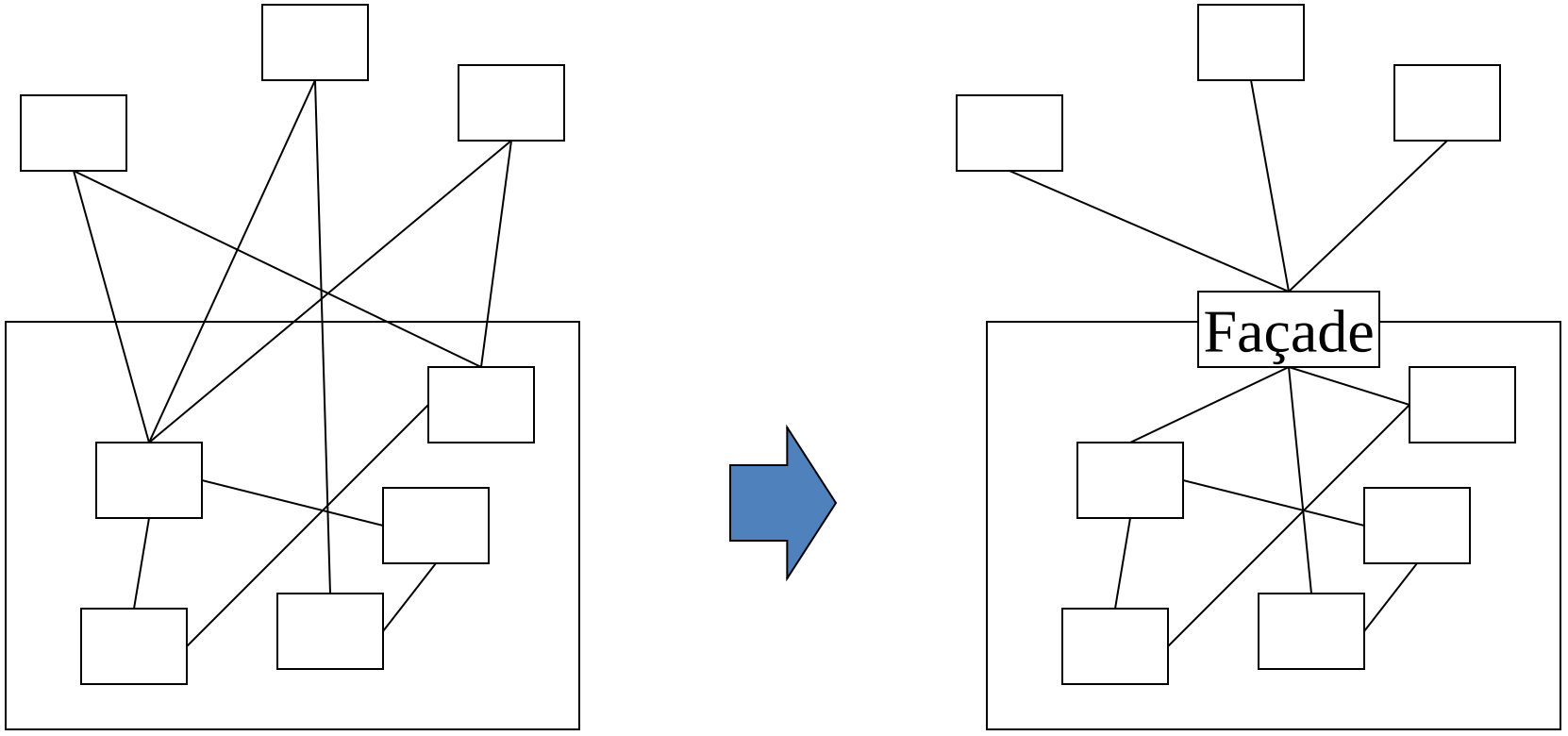
```
public class ReverserModel {

    public String reverse(String s) {
        StringBuilder builder = new StringBuilder(s);
        builder.reverse();
        return builder.toString();
    }
}
```

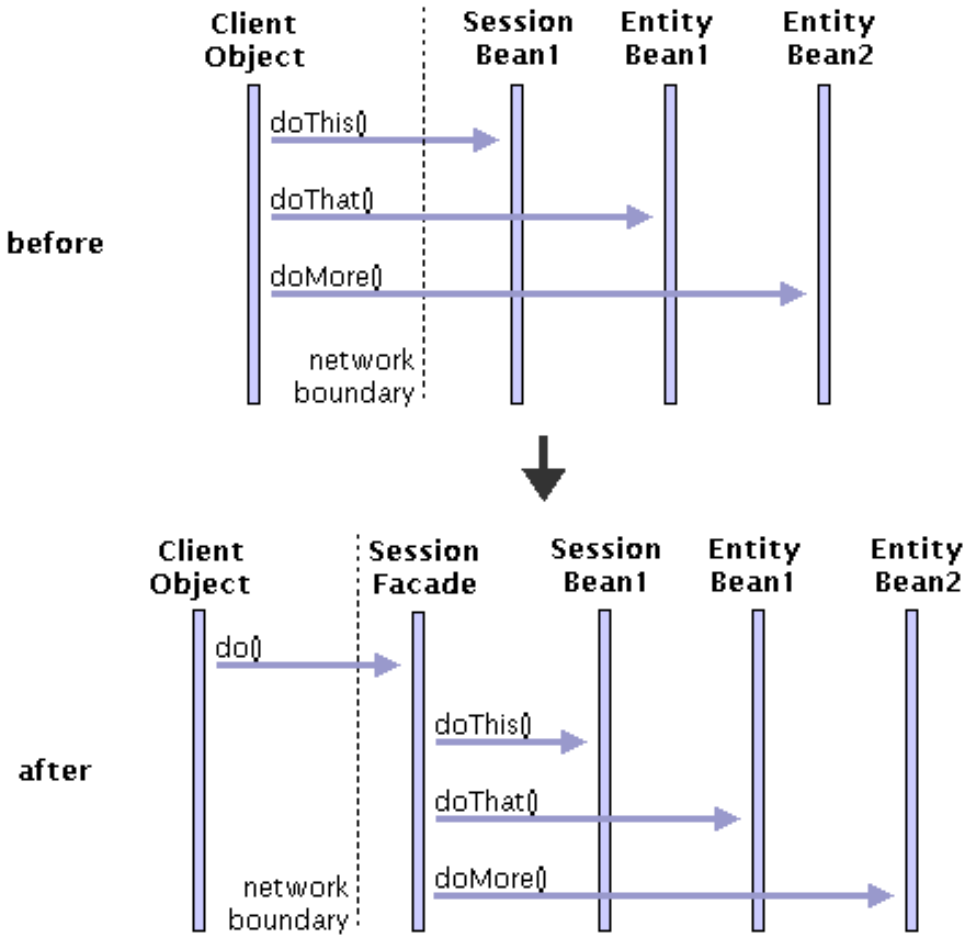
Facade

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Façade

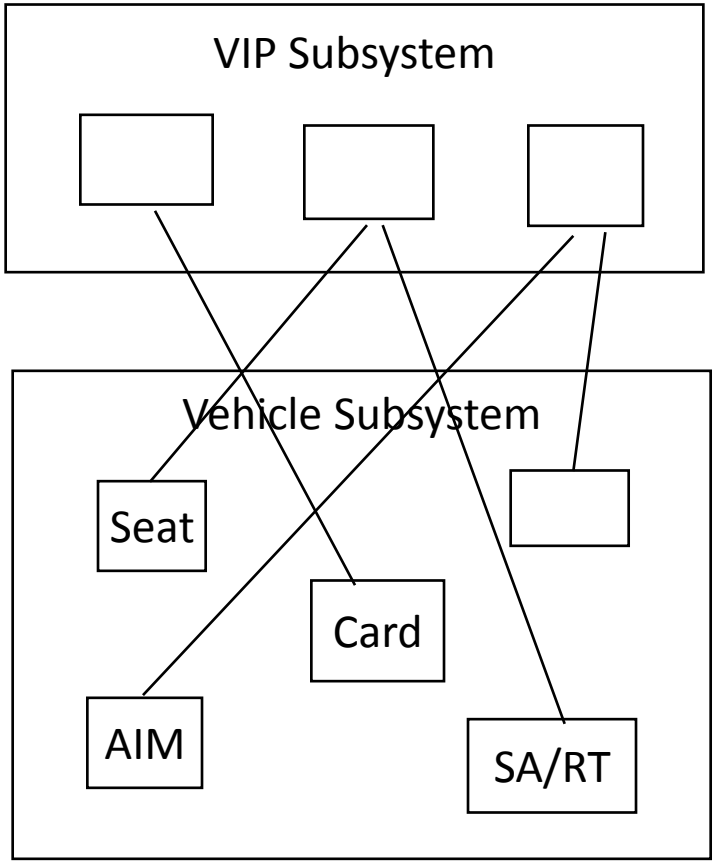


Façade


Client Object
Session Bean1
Entity Bean1
Entity Bean2
before
doThis()
doThat()
doMore()
network boundary
Client Object
Session Facade
Session Bean1
Entity Bean1
Entity Bean2
after
do()
doThis()
doThat()
doMore()
network boundary

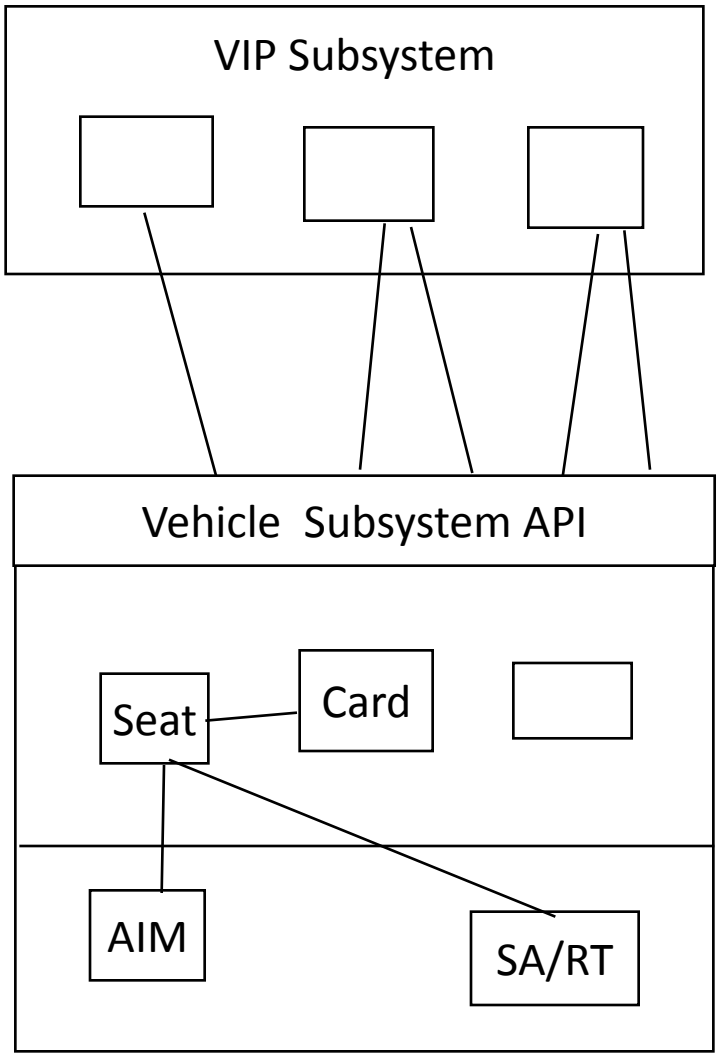
Open vs Closed Architecture

- Open architecture:
  - Any client can see into the vehicle subsystem and call on any component or class operation at will.
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can't expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code



Realizing a Closed Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a façade is used, the subsystem can be used in an early integration test
  - We need to write only a



Façade - Example