Principios SOLID
Arquitectura MVC

Semana 4

Agenda

- Principios de diseño aplicados al desarrollo: SOLID
  - Single Responsability Principle
  - Open-Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle
- Diseño de arquitectura Modelo Vista Controlador (MVC)

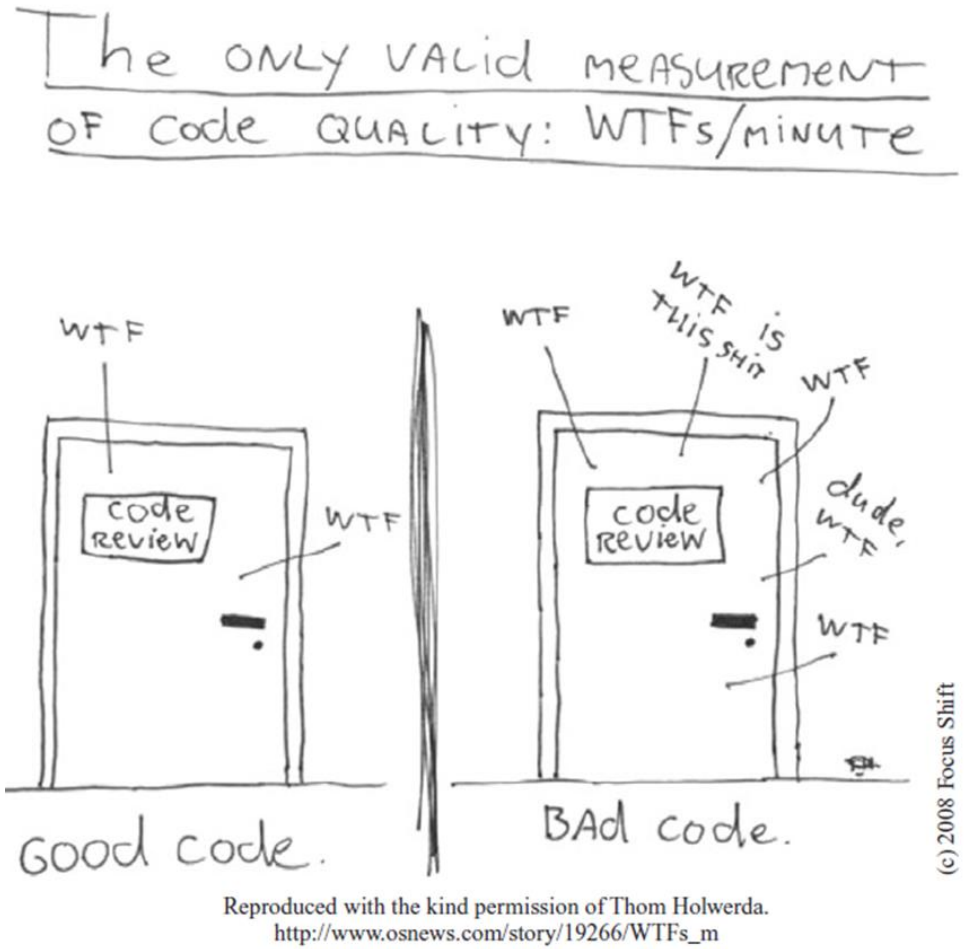
¿Qué significa código limpio?

Uncle Bob (El autor de nuestro texto guía)

- Robert C. Martin (conocido como Uncle Bob) empezó el debate de los principios SOLID a finales de los 1980s.
- El grupo de principios definitivos se consolidó en los 2000 y se denominó oficialmente como SOLID en el 2004.

Principios de diseño aplicados al desarrollo

- Buenos sistemas de software empiezan con un **código limpio**.
  - Pero, ¿qué es un **código limpio**?



Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Extractos de algunas definiciones de Código Limpio

- "Clean code can be read and **enhanced by a developer other than its original author**. It has unit and acceptance tests. It has meaningful names. It provides a clear and minimal API".
  - Dave Thomas, founder of OTI, godfather of the Eclipse strategy
- "Clean code is **simple and direct**. Clean code reads like well-written prose".
  - Grady Booch, author of Object-Oriented Analysis and Design with Applications
- "Clean code always looks like it was **written by someone who cares**. There is **nothing obvious that you can do to make it better**".
  - Michael Feathers, author of Working Effectively with Legacy Code

Extractos de algunas definiciones de Código Limpio

- "You know you are working on clean code when **each routine** you read turns out to be **pretty much what you expected**".
  - Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code.
- "The logic should be straightforward to make it **hard for bugs to hide**, the dependencies minimal to ease maintenance, **error handling complete** according to an **articulated strategy**, and **performance close to optimal**".
  - Bjarne Stroustrup, inventor of C++ and author of The C++ Programming Language

Principios de diseño aplicados al desarrollo: SOLID

Principios de diseño aplicados al desarrollo

- Los principios SOLID nos dicen cómo **ordenar** nuestras **funciones y estructuras de datos** en **clases** y cómo esas clases deben estar **interconectadas**.
- La palabra clase no implica que SOLID es exclusivo de OO, ya que las clases no son más que una forma de agrupar datos y funciones.
- El objetivo de estos principios es la creación de estructuras de software que:
  - Toleren cambios;
  - Sean entendibles; y
  - Sean la base de componentes que puedan ser reutilizables

Bad Coding...

In spaghetti code, the relations between the pieces of code are so tangled that it is nearly impossible to add or change something without unpredictably breaking something somewhere else.

Bad Coding

- RIGIDEZ
  - Un cambio afecta muchas partes.
- FRAGILIDAD
  - Fallos en lugares donde no debería.
- INMOBILIDAD
  - No se puede reusar el código fuera de su contexto original.

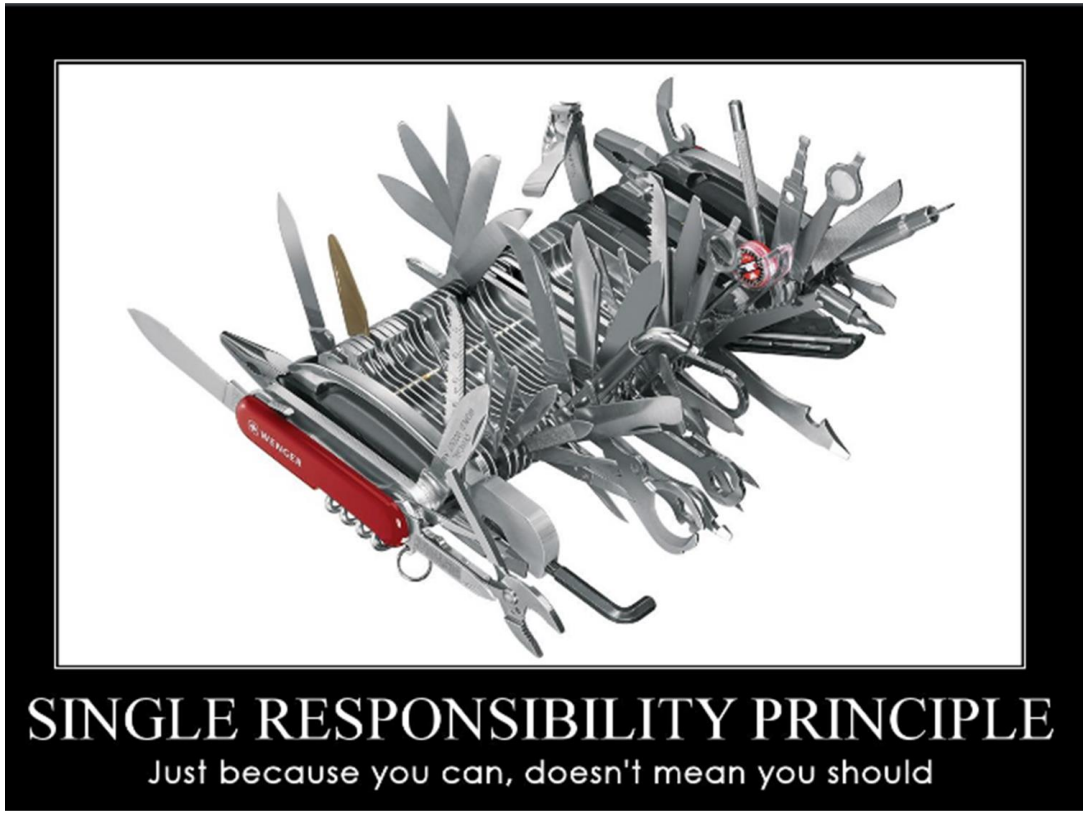
SOLID

- **S**: Single Responsability Principle
- **O**: Open-Closed Principle
- **L**: Liskov Substitution Principle
- **I:** Interface Segregation Principle
- **D:** Dependecy Inversion Principle

Single-Responsibility Principle (SRP)

Single-Responsibility Principle

- A class should have one and only one reason to change, meaning that a class should have only one job.



Single-Responsibility Principle

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  public boolean isPromotionDueThisYear(){
    //promotion logic implementation
  }
  public Double calcIncomeTaxForCurrentYear(){
    //income tax logic implementation
  }
  //Getters & Setters for all the private attributes
}
```

¿Se viola el principio SRT? ¿Cuántas responsabilidades tiene esta clase?

Single-Responsibility Principle

HRPromotions.java

```
public class HRPromotions{
  public boolean isPromotionDueThisYear(Employee emp){
    //promotion logic implementation using the employee information passed
  }
}
```

FinITCalculations.java

```
public class FinITCalculations{
  public Double calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation using the employee information passed
  }
}
```

Implementación con SRP

Employee.java adhering to Single Responsibility Principle

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  //Getters & Setters for all the private attributes
}
```

La clase Employee mantiene su responsabilidad de manejar la info del Empleado y usa los servicios de las demás clases.

Open-Closed Principle (OCP)

Open-Closed Principle

- Objects or entities should be open for extension, but closed for modification

Open-Closed Principle

- Supongamos que deseamos calcular el área de varias figuras geométricas-

Rectangle.java

```
public class Rectangle{
 public double length;
 public double width;
}
```

AreaCalculator.java

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle rectangle){
    return rectangle.length *rectangle.width;
  }
}
```

Open-Closed Principle

Circle.java

```
public class Circle{
 public double radius;
}
```

AreaCalculator.java

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle rectangle){
    return rectangle.length *rectangle.width;
  }
  public double calculateCircleArea(Circle circle){
    return (22/7)*circle.radius*circle.radius;
  }
}
```

¨¿Se cumple el OCP?

Cumpliendo OCP

Shape.java

```
public interface Shape{
  public double calculateArea();
}

public class Rectangle implements Shape{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

public class Circle implements Shape{
  public double radius;
  public double calculateArea(){
    return (22/7)*radius*radius;
  }
}
```

Cumpliendo OCP y SRP

AreaCalculator.java

```
public class AreaCalculator{
  public double calculateShapeArea(Shape shape){
    return shape.calculateArea();
  }
}
```

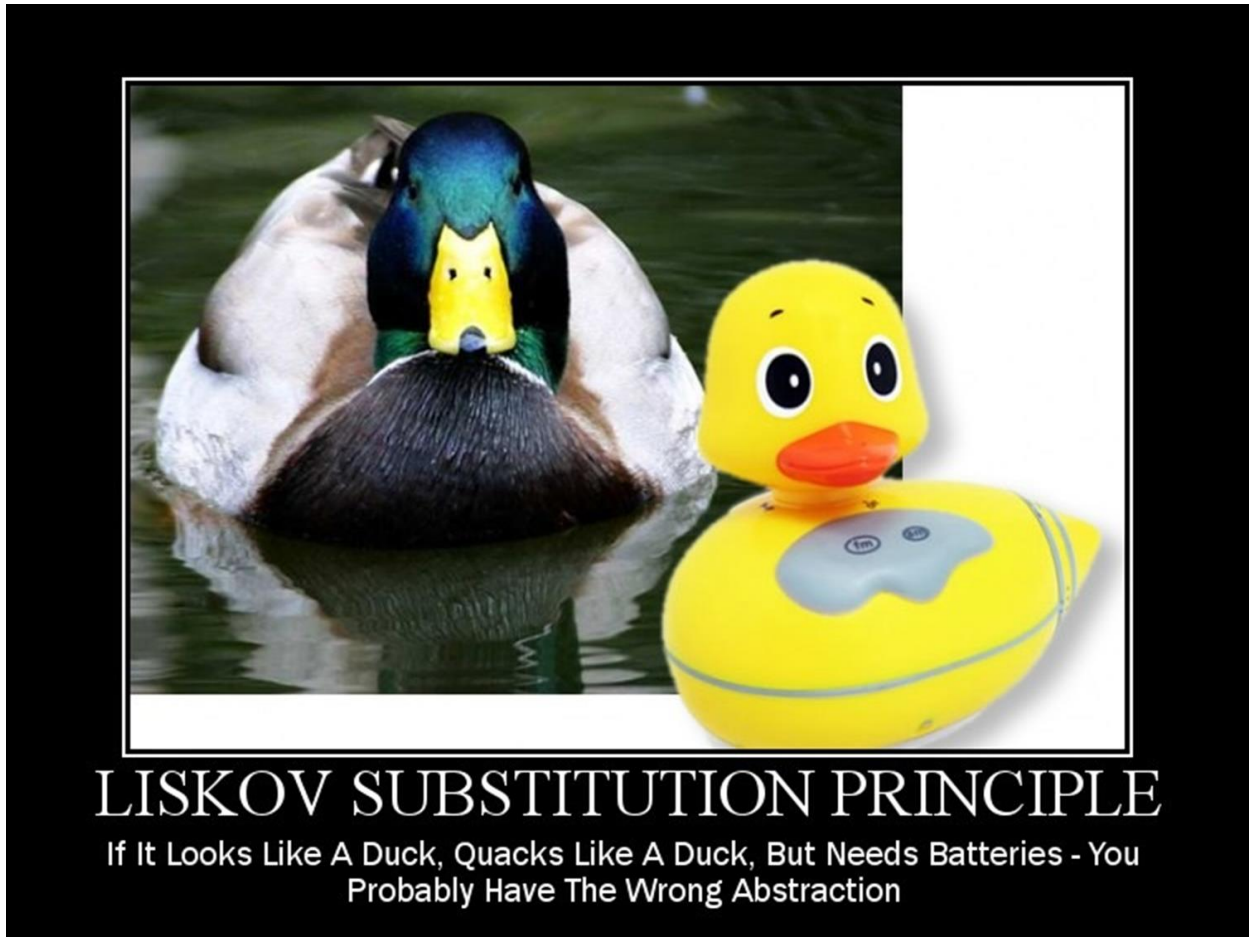
- Con este diseño, el código **queda abierto para extensión** pues muchas más formas pueden agregarse sin modificar el código existente y cada forma debe implementar el método calculateArea().

- El diseño queda **cerrado para modificación, AreaCalculator** ejecuta una función independientemente de las formas que se agreguen.

Liskov Substitution Principle (LSP)

Liskov Substitution Principle

- Derived classes must be completely substitutable for their base classes.



Liskov Substitution Principle

```
public interface IDuck{
       void Swim();
    }
public class Duck implements Iduck {
       public void Swim(){
          //do something to swim
       }
    }
public class ElectricDuck implements IDuck{
       public void Swim() {
          if (!IsTurnedOn)
            return;
          //swim logic  }
     }
```

Liskov Substitution Principle

En el main se llama a este método.

```
void MakeDuckSwim(IDuck duck) {
        duck.Swim();
    }
```

¿Problema?

Se viola LSP, porque una clase derivada puede lazar una excepción que la clase base no.

LSP

- Intento de solución 1:

```
void MakeDuckSwim(IDuck duck)
    {
        if (duck is ElectricDuck)
            ((ElectricDuck)duck).TurnOn();
        duck.Swim();
    }
```

¿Problema?

Liskov Substitution Principle

- Intento de solución 1:

```
void MakeDuckSwim(IDuck duck)
    {
        if (duck is ElectricDuck)
            ((ElectricDuck)duck).TurnOn();
        duck.Swim();
    }
```

Problema: OCP

Liskov Substitution Principle

- Intento de solución 2:

```
public class ElectricDuck : IDuck
   {
      public void Swim()
      {
         if (!IsTurnedOn)
           TurnOnDuck();

         //swim logic
      }
   }
```

Interface Segregation Principle (ISP)

Interface Segregation Principle

- Make fine-grained interfaces with specific methods. Clients should not be forced to depend upon interfaces that they don't use.

Interface Segregation Principle

- Supongamos que disponemos de una interfaz llamada Messenger, la cual permite mostrar una variedad de mensajes cuando un usuario interactúa con un cajero automático.

- Se desea crear una App que mostrará mensajes cuando se efectúe un **Depósito**.

```
public interface Messenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
  tellCardWasSiezed();
  askForAccount();
  tellNotEnoughMoneyInAccount();
  tellAmountDeposited();
  tellBalance();
}
```

¿Problema?

Interface Segregation Principle

- Dividamos la interfaz, así una clase no es forzada a implementar lo que no necesita.

```
public interface LoginMessenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
}

public interface WithdrawalMessenger {
  tellNotEnoughMoneyInAccount();
  askForFeeConfirmation();
}

publc class EnglishMessenger implements LoginMessenger, WithdrawalMesse
  ...
}
```

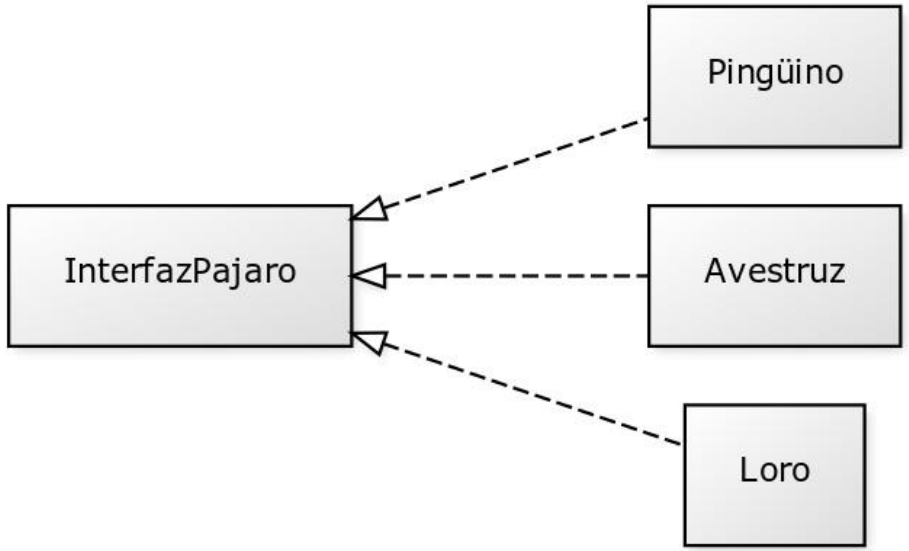
Interface Segregation Principle

- Para nuestra App, podríamos utilizar una interface con los métodos que sí va a utilizar.

```
public interface Deposit  {
  tellAmountDeposited();
  tellBalance();
}
```

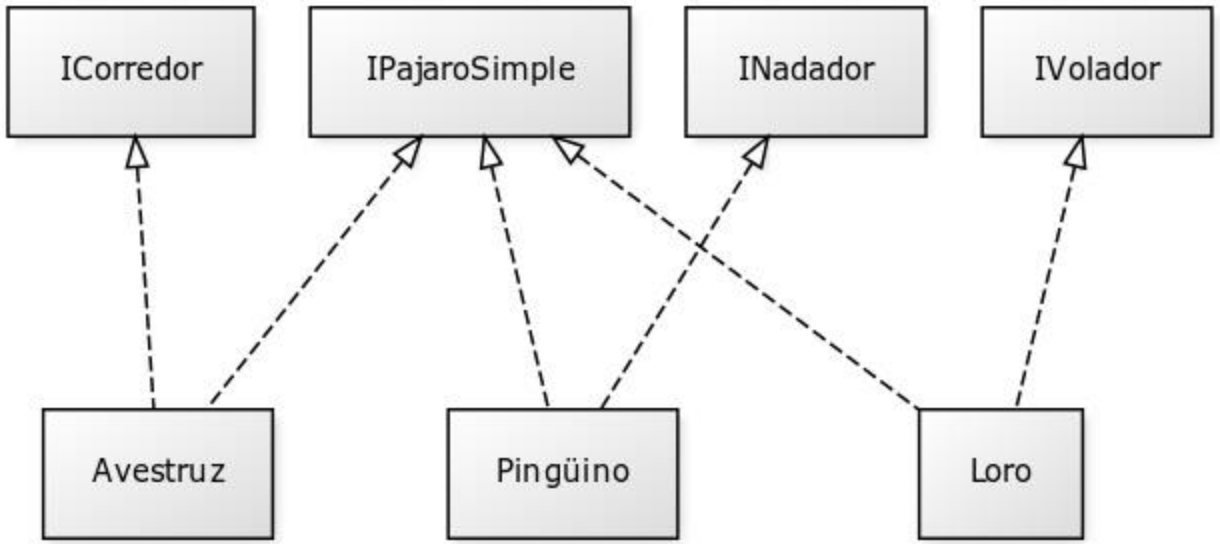
Interface Segregation Principle

- Volar
- Correr
- Nadar
- Comer
- Cantar
- Graznar
- Planear
- Caminar
- Hablar
- Poner huevo



¿Problema?

ISP - Solución



Dependency Inversion Principle (DIP)

Dependency Inversion Principle

- Depend on abstractions, not on concretions. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Take advantage of interfaces

Incumpliendo DIP

```
public class Door {…}
public class Window {…}

public class House {
    private Door _door;
    private Window _window;

    public House()
    {
        _door = new Door();
        _window = new Window();
    }
}
```

Cumpliendo DIP

```
public class House{
    private IDoor _door;
    private IWindow _window;
    public House(IDoor door, IWindow window) {
        _door = door;
        _window = window;
    }
    public House(){}
    public IDoor Door;
    public IWindow Window;
}
```

RESUMEN: S.O.L.I.D.

SRP	Single Responsability Principle	A class should have one, and only one, reason to change.
OCP	Open-Closed Principle	You should be able to extend a classes behavior, without modifying it.
LSP	Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
ISP	Interface Segregation Principle	Make fine grained interfaces that are client specific.
DIP	Dependency Inversion Principle	Depend on abstractions, not on concretions.


SOLID
Software Development is not a Jenga game

Arquitectura Modelo-Vista-Controlador (MVC)

Diseño de arquitectura modelo vista controlador

- En 1988, se empezó a argumentar que las aplicaciones deben ser construidas usando capas, de manera que los diferentes *concerns* sean tomados en consideración.
  - Específicamente se propuso un diseño de **tres capas** que desacople **interfaz, navegación** y **comportamiento** de la aplicación.
- MVC es un **patrón de diseño arquitectural** que **desacopla** la **interfaz de usuario** de la **funcionalidad de la aplicación** y del **contenido de información**.
  - También es un modelo de infraestructura de aplicaciones Web.

Arquitectura MVC

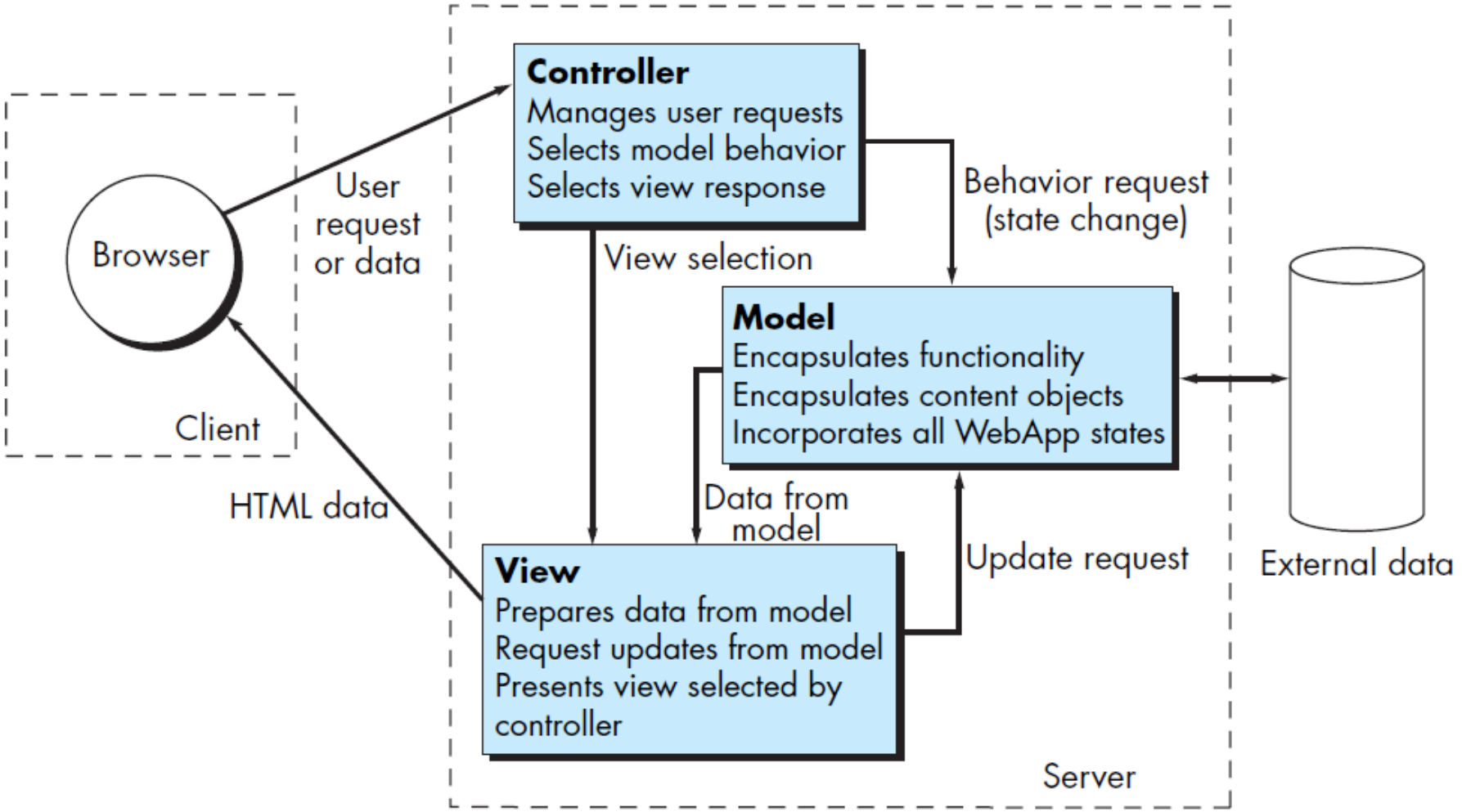
- **El modelo** contiene objetos de contenido, el acceso a fuentes de información/datos y el procesamiento de la funcionalidad específica.
- **La vista** contiene todas las funciones específicas de las interfaces y habilita la presentación de contenido requerida por el usuario final.
- **El controlador** maneja acceso al modelo y la vista; y coordina el flujo de dato entre ellas.
- En una aplicación Web la **vista es actualizada por el controlador con datos desde el modelo basados en la petición del usuario**.

Arquitectura modelo vista controlador

FIGURE 17.8

The MVC architecture

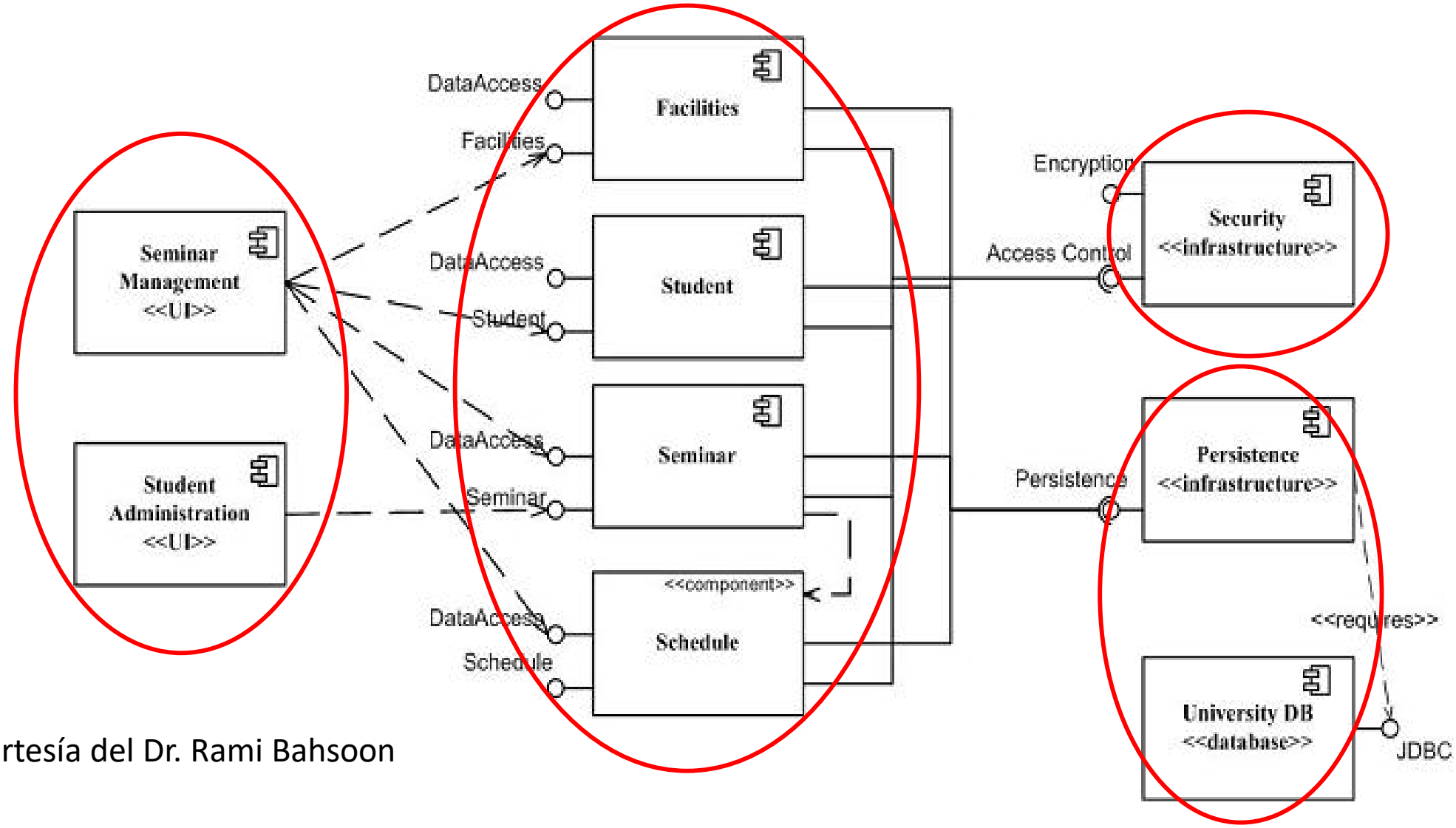
Source: Adapted from [Jac02b].



Descripción de la figura de la Arquitectura MVC

- Peticiones del usuario o datos son manejados por el controlador.
- El controlador selecciona el objeto vista que es aplicable basado en la petición del usuario.
- Una vez que el tipo de petición es determinado, una petición de comportamiento es transmitida al modelo, el cual implementa la funcionalidad o recupera el contenido requerido para acomodar la petición.
- El objeto modelo puede acceder a los datos almacenados en un base de datos o archivos.
- Los datos desarrollados por el modelo deben ser formateados y organizados por el objeto vista apropiado y luego transmitidos desde el servidor de aplicaciones de retorno al browser para el despliegue en la máquina del usuario.

Ejemplo de una arquitectura con 4 capas



Cortesía del Dr. Rami Bahsoon

Antes de finalizar

Puntos para recordar

- ¿Cuáles son características de un código limpio?
- ¿Qué es SOLID?
- ¿Puede resumir el objetivo de cada principio SOLID?
- ¿Puede proveer un ejemplo para cada principio SOLID?
- ¿Cuántas capas tiene una arquitectura MVC?
- ¿Cuál es la responsabilidad de cada capa?

Lectura adicional

- Robert C. Martin, “Clean Architecture”
  - Chapter 7: The Single Responsibility Principle
  - Chapter 8: The Open-Close Principle
  - Chapter 9: The Liskov Substitution Principle
  - Chapter 10: The Interface Segregation Principle
  - Chapter 11: The Dependency Inversion Principle
- Pressman and Maxim, “Software Engineering”
  - Chapter 17: WebApp Design
- Robert C.Martin, “Clean Code”
  - Chapter 1: Clean Code

Próxima sesión

- Lenguaje unificado de modelado: casos de uso, clases y secuencias.