

# DEVELOPMENT OF INFORMATION SYSTEMS

## Lecture 3

# Repetition of the last lecture

## Domain of Information System:

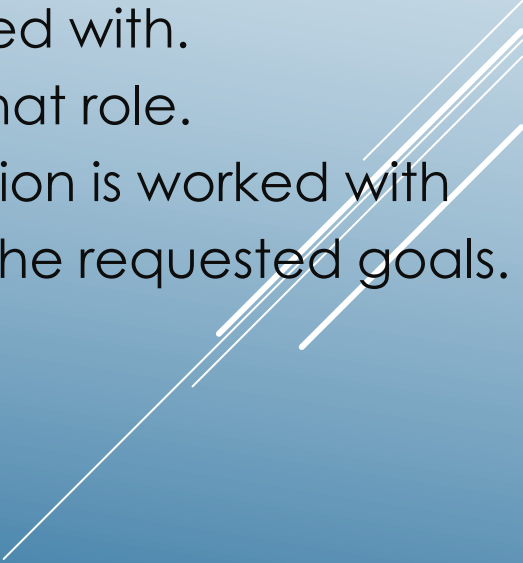
By the **domain** we understand a group of related "things, concepts, terms etc." in the sense of a customer or user point of view.

The domain is related to the purpose and area of Information system.

Persons working with IS understand domain terms as system fundamentals.

# Repetition of the last lecture

## Key questions related to the domain

- **WHAT?** It is information in the sense of data.
  - **HOW?** It is about the processes performed with the information.
  - **WHERE?** The places where information is worked with.
  - **WHO?** Who works with information and in what role.
  - **WHEN?** When and on what impulses information is worked with
  - **WHY?** It is about aims and rules to achieve the requested goals.
- 
- Three white lines of varying lengths and slopes are positioned in the bottom right corner of the slide, extending from the right edge towards the bottom left.

# Repetition of the last lecture

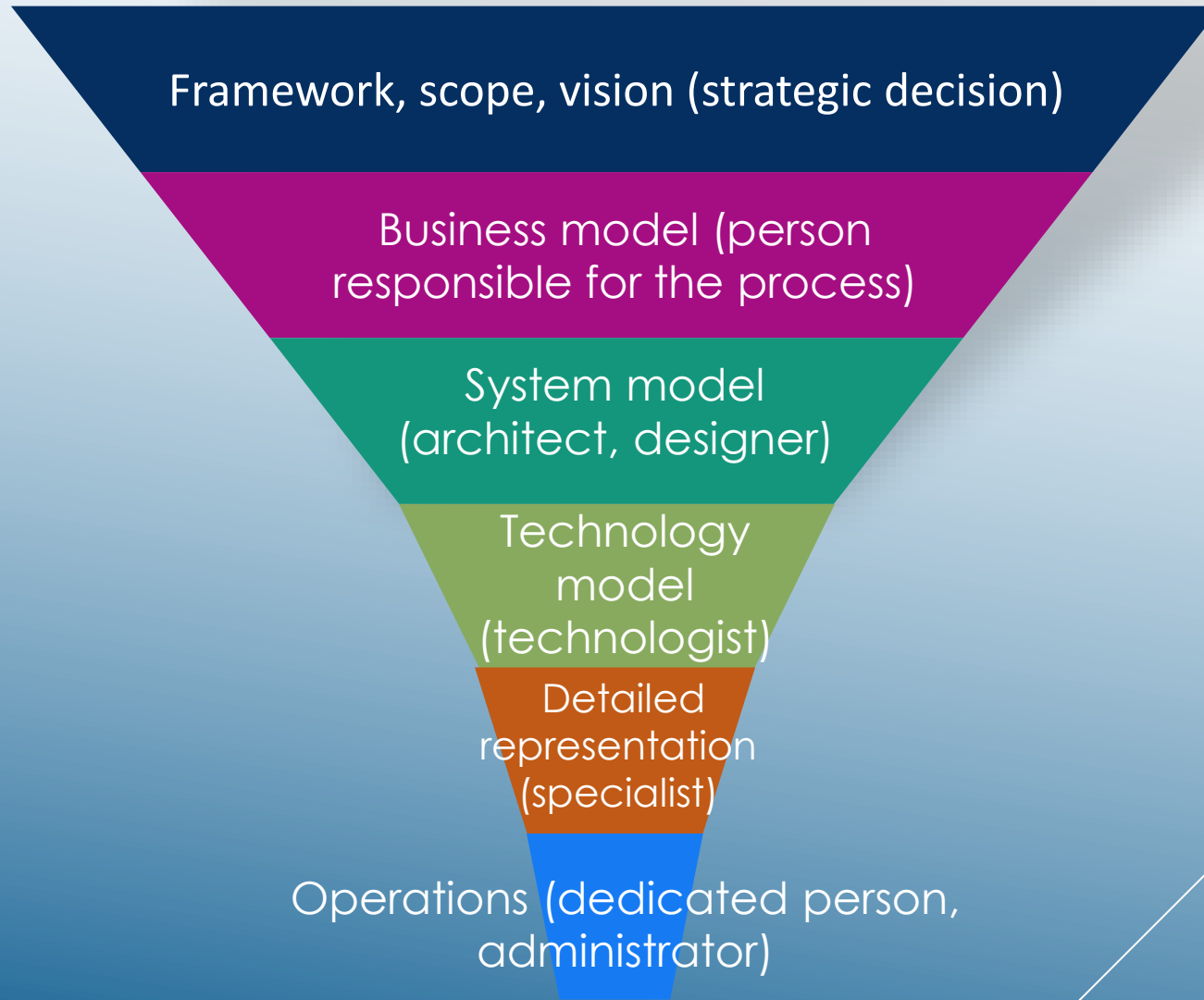
## Classification of IS

Different types of agendas:

- Economic
- Personal
- Warehouse
- Documentographic (library, records management)
- Geographics (GIS)
- School, students
- ERP, HRM, CRM, Project management, Supply chain management

# Repetition of the last lecture

## From the general to the specific



# Repetition of the last lecture

## Zachman's framework

	DATA <i>What</i>	FUNCTION <i>How</i>	NETWORK <i>Where</i>	PEOPLE <i>Who</i>	TIME <i>When</i>	MOTIVATION <i>Why</i>
Objective/Scope (contextual) <i>Role: Planner</i>	List of things important in the business	List of Business Processes	List of Business Locations	List of important Organizations	List of Events	List of Business Goal & Strategies
Enterprise Model (conceptual) <i>Role: Owner</i>	Conceptual Data/ Object Model	Business Process Model	Business Logistics System	Work Flow Model	Master Schedule	Business Plan
System Model (logical) <i>Role: Designer</i>	Logical Data Model	System Architecture Model	Distributed Systems Architecture	Human Interface Architecture	Processing Structure	Business Rule Model
Technology Model (physical) <i>Role: Builder</i>	Physical Data/Class Model	Technology Design Model	Technology Architecture	Presentation Architecture	Control Structure	Rule Design
Detailed Representation (out of context) <i>Role: Programmer</i>	Data Definition	Program	Network Architecture	Security Architecture	Timing Definition	Rule Speculation
Functioning Enterprise <i>Role: User</i>	Usable Data	Working Function	Usable Network	Functioning Organization	Implemented Schedule	Working Strategy

# Repetition of the last lecture

## IS development main steps

- **Vision**
  - **Analysis** (user requirements)
  - **Logical design** (functional req)
  - **Technological design** (non-functional req)
  - **Development**
  - **Deployment and operation**
- 
- Several white lines of varying lengths and orientations are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.



# Repetition of the last lecture

- **Software architecture** - „The basic organization of a software system, including its components, their interrelationships and relationships with the system environment, the principles of the design of such a system and its development.“
- **Software component** - A software component is a software package, service or generally a module that provides a particular functionality, and therefore encapsulating functionality and data.
- **ARCHITECTURE** deals primarily with technical (other than functional) and partly functional requirements, whereas **DESIGN** is based on purely functional requirements.



# Recommended reading

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1995.
- **Designing programs using patterns - Building blocks of object-oriented 2003. [GoF]**
- **Martin Fowler. Patterns of Enterprise Application Architecture. 2003.**
- David Trowbridge, Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk, David Lavigne. Enterprise Solution Patterns Using Microsoft .NET. 2003.
- Adam Bien. Real World Java EE Patterns – Rethinking Best Practices. 2009.

# Gang of Four (1994, 1995)

General principles of software development

- 23 patterns (C++, Smalltalk)
- Creational patterns
- Structural patterns
- Behavioral patterns

# Patterns as ready-made solutions

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

- GoF is just the beginning.
- Patterns and pattern languages.
- A pattern is what works repeatedly.

# There are also anti-patterns

(they don't work repeatedly ☹ ).

- **Ambiguous viewpoint:** Presenting a model without specifying its viewpoint
- **Big ball of mud:** A system with no recognizable structure
- **Interface bloat:** Making an interface so powerful that it is extremely difficult to implement
- **Constant interface:** Using interfaces to define constants
- **God object:** Concentrating too many functions in a single part of the design (class)
- **Action at a distance:** Unexpected interaction between widely separated parts of a system
- **Boat anchor:** Retaining a part of a system that no longer has any use
- **Magic numbers:** Including unexplained numbers in algorithms

# Pattern structure

- Concise name (title)
- Problem (What)
- Context (Who, When, Where, Why)
- Solution including UML diagrams (How)
- Examples including source codes

[https://www.tutorialspoint.com/design\\_pattern/index.htm](https://www.tutorialspoint.com/design_pattern/index.htm)

# GOF Patterns overview

## Types Of Design Patterns

```
graph TD; Root[Types Of Design Patterns] --> Creational[Creational]; Root --> Sturctural[Sturctural]; Root --> Behavioural[Behavioural]; Creational --> C1[1.Singleton]; Creational --> C2[2.Factory]; Creational --> C3[3.Abstract Factory]; Creational --> C4[4.Builder]; Creational --> C5[5.Prototype]; Sturctural --> S6[6.Adapter]; Sturctural --> S7[7.Composite]; Sturctural --> S8[8.Proxy]; Sturctural --> S9[9.Fly Weight]; Sturctural --> S10[10.Facade]; Sturctural --> S11[11.Bridge]; Sturctural --> S12[12.Decorator]; Behavioural --> B13[13.Template Method]; Behavioural --> B14[14.Mediator]; Behavioural --> B15[15.Chain Of Responsibility]; Behavioural --> B16[16.Observer]; Behavioural --> B17[17.Strategy]; Behavioural --> B18[18.Command]; Behavioural --> B19[19.State]; Behavioural --> B20[20.Visitor]; Behavioural --> B21[21.Iterator]; Behavioural --> B22[22.Interpreter]; Behavioural --> B23[23.memento];
```

### Creational

- 1.Singleton
- 2.Factory
- 3.Abstract Factory
- 4.Builder
- 5.Prototype


### Sturctural

- 6.Adapter
- 7.Composite
- 8.Proxy
- 9.Fly Weight
- 10.Facade
- 11.Bridge
- 12.Decorator

### Behavioural

- 13.Template Method
- 14.Mediator
- 15.Chain Of Responsibility
- 16.Observer
- 17.Strategy
- 18.Command
- 19.State
- 20.Visitor
- 21.Iterator
- 22.Interpreter
- 23.memento

# Singleton

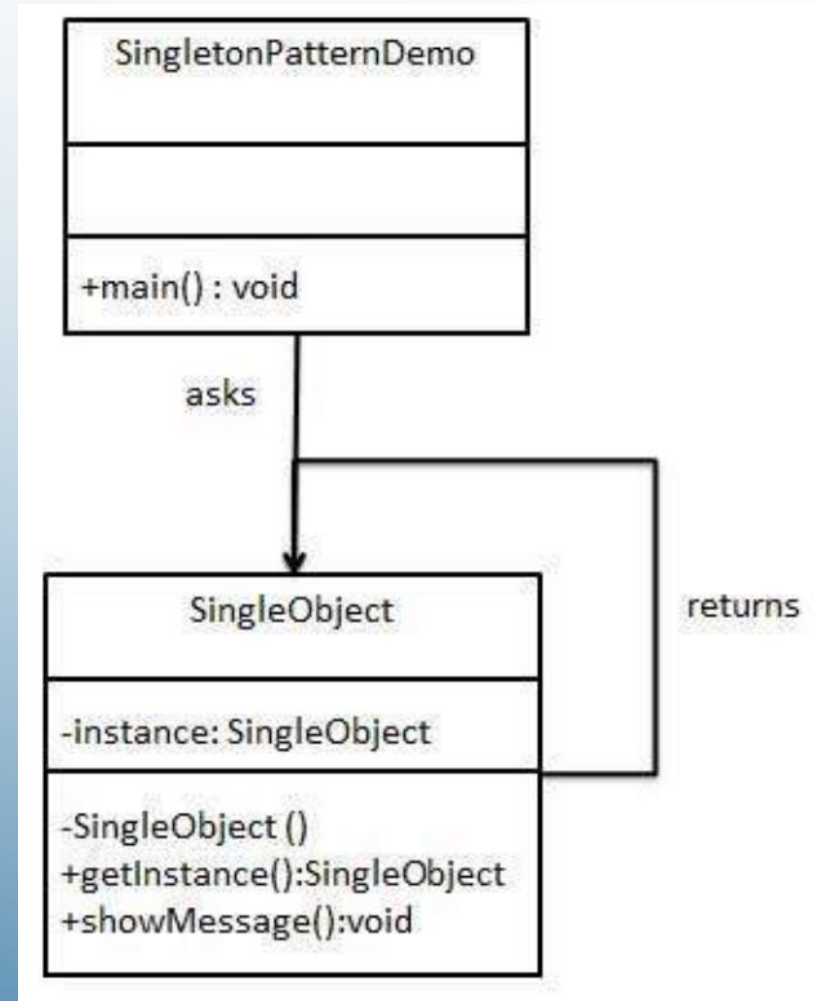
- Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
  - This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.
  - What is antipattern?
- 
- A series of three parallel white diagonal lines in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.



# Singleton implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

*SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



# Source code in JAVA

*SingleObject.java*

```
public class SingleObject {

    //create an object of SingleObject
    private static SingleObject instance = new SingleObject();

    //make the constructor private so that this class cannot be
    //instantiated
    private SingleObject(){}

    //Get the only object available
    public static SingleObject getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }
}
```

*SingletonPatternDemo.java*

```
public class SingletonPatternDemo {
    public static void main(String[] args) {

        //illegal construct
        //Compile Time Error: The constructor SingleObject() is not visible
        //SingleObject object = new SingleObject();


        //Get the only object available
        SingleObject object = SingleObject.getInstance();

        //show the message
        object.showMessage();
    }
}
```

# Facade pattern

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

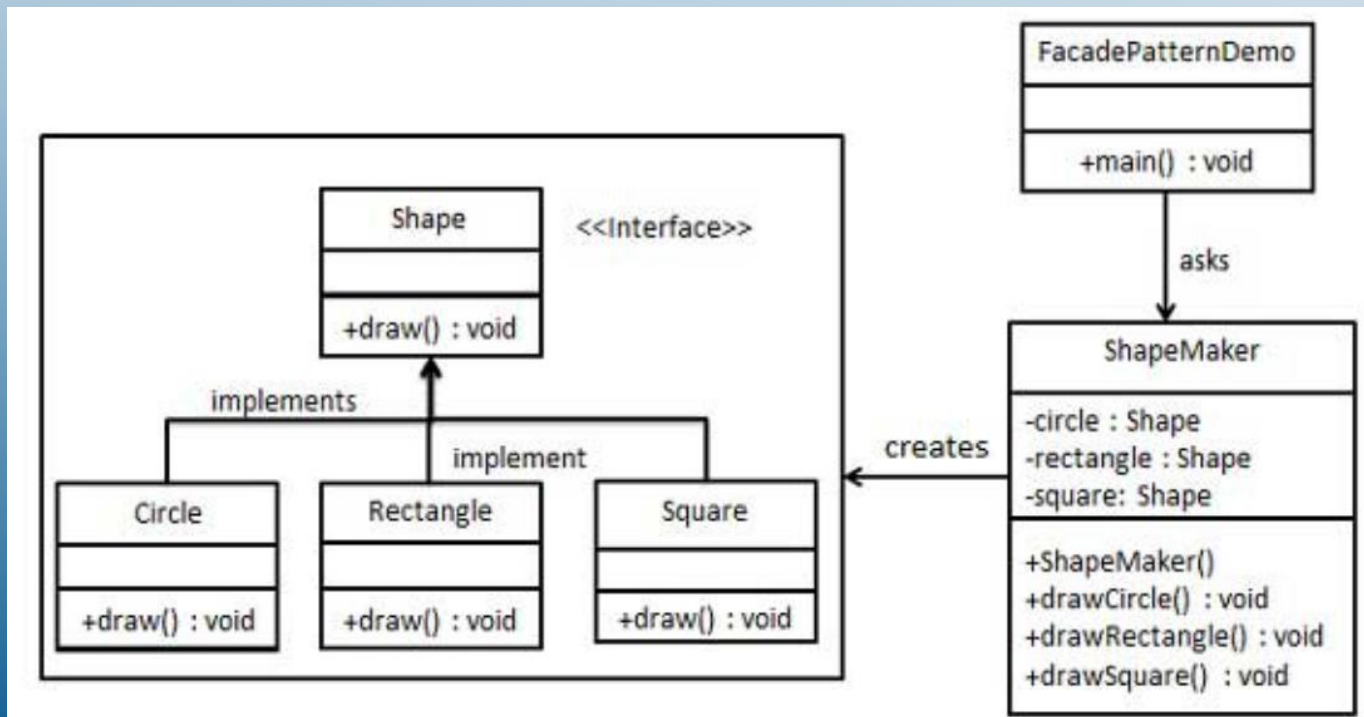
This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

A decorative graphic consisting of several parallel white lines of varying lengths, slanted diagonally upwards from left to right, located in the bottom right corner of the slide.

# Facade implementation

We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

*ShapeMaker* class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



# Facade source code

```
public interface Shape {  
    void draw();  
}
```

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}
```

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}
```

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Circle::draw()");  
    }  
}
```

```
public class ShapeMaker {  
    private Shape circle;  
    private Shape rectangle;  
    private Shape square;  
  
    public ShapeMaker() {  
        circle = new Circle();  
        rectangle = new Rectangle();  
        square = new Square();  
    }  
  
    public void drawCircle(){  
        circle.draw();  
    }  
    public void drawRectangle(){  
        rectangle.draw();  
    }  
    public void drawSquare(){  
        square.draw();  
    }  
}
```

# Proxy pattern

In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.

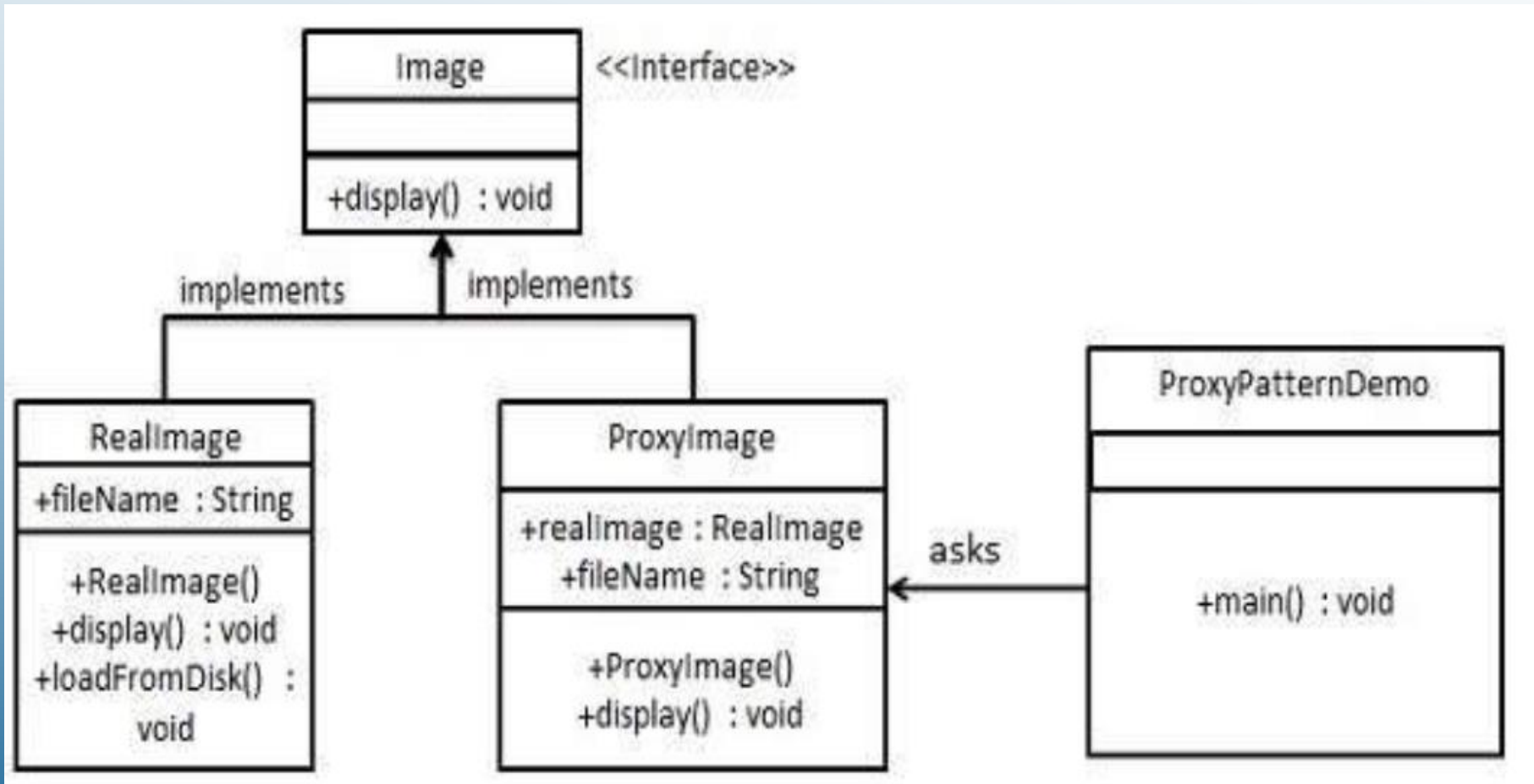
In proxy pattern, we create object having original object to interface its functionality to outer world.

## Implementation

We are going to create an *Image* interface and concrete classes implementing the *Image* interface. *ProxyImage* is a proxy class to reduce memory footprint of *RealImage* object loading.

*ProxyPatternDemo*, our demo class, will use *ProxyImage* to get an *Image* object to load and display as it needs.

# Proxy pattern diagram





# Proxy pattern source code

```
public interface Image {  
    void display();  
}
```

```
public class RealImage implements Image {  
  
    private String fileName;  
  
    public RealImage(String fileName){  
        this.fileName = fileName;  
        loadFromDisk(fileName);  
    }  
  
    @Override  
    public void display() {  
        System.out.println("Displaying " + fileName);  
    }  
  
    private void loadFromDisk(String fileName){  
        System.out.println("Loading " + fileName);  
    }  
}
```

```
public class ProxyImage implements Image{  
  
    private RealImage realImage;  
    private String fileName;  
  
    public ProxyImage(String fileName){  
        this.fileName = fileName;  
    }  
  
    @Override  
    public void display() {  
        if(realImage == null){  
            realImage = new RealImage(fileName);  
        }  
        realImage.display();  
    }  
}
```

# Proxy pattern using

```
public class ProxyPatternDemo {  
  
    public static void main(String[] args) {  
        Image image = new ProxyImage("test_10mb.jpg");  
  
        //image will be loaded from disk  
        image.display();  
        System.out.println("");  
  
        //image will not be loaded from disk  
        image.display();  
    }  
}
```

# Fowler (EAA, 2003)

- Various extensive solutions in the form of a catalogue
- More than 50 patterns (Java, C#).

Domain Logic Patterns, Data Source Architectural Patterns, Object-Relational Behavioral Patterns, Object-Relational Structural Patterns, Object-Relational Metadata Mapping Patterns, Web Presentation Patterns, Distribution Patterns, Offline Concurrency Patterns, Session State Patterns, Base Patterns

More <http://martinfowler.com/eaDev/>

# Layers vs Tiers

**Layer is often used interchangeably – and mistakenly – for tier, as in 'presentation layer' or 'business logic layer.'**

- They aren't the same. A '**layer**' refers to a functional division of the software, but a '**tier**' refers to a functional division of the software that runs on infrastructure separate from the other divisions. The Contacts app on your phone, for example, is a three-layer application, but a single-tier application, because all three layers run on your phone.
- The difference is important, because layers can't offer the same benefits as tiers.
- Monolithic architecture

# Layered architecture

## Benefits of three-layer/tier architecture

### **Logical and physical separation of functionality.**

Each tier can run on a separate operating system and server platform - e.g., web server, application server, database server - that best fits its functional requirements. And each tier runs on at least one dedicated server hardware or virtual server, so the services of each tier can be customized and optimized without impact the other tiers.

- **Faster development:** Because each tier can be developed simultaneously by different teams
- **Improved scalability:** Any tier can be scaled independently of the others as needed.
- **Improved reliability:** An outage in one tier is less likely to impact the availability or performance of the other tiers.
- **Improved security:** Because the presentation tier and data tier can't communicate directly

# Layered architecture

**Two-tier architecture** is the original client-server architecture, consisting of a presentation tier and a data tier; the business logic lives in the presentation tier, the data tier or both. In two-tier architecture the presentation tier - and consequently the end user - has direct access to the data tier, and the business logic is often limited. A simple contact management application, where users can enter and retrieve contact data, is an example of a two-tier application.

**N-tier architecture** - also called or multi-tier architecture - refers to *any* application architecture with more than one tier. But applications with more than three layers are rare, because additional layers offer few benefits and can make the application slower, harder to manage and more expensive to run. As a result, n-tier architecture and multi-tier architecture are usually synonyms for three-tier architecture.

# Three layer/tier architecture

Three-tier architecture is a well-established software application architecture that organizes applications into three logical and physical computing tiers: the presentation tier, or user interface; the application tier, where data is processed; and the data tier, where the data associated with the application is stored and managed.

The key benefit of three-tier architecture is that because each tier runs on its own infrastructure, each tier can be developed simultaneously by a separate development team, and can be updated or scaled as needed without impacting the other tiers.

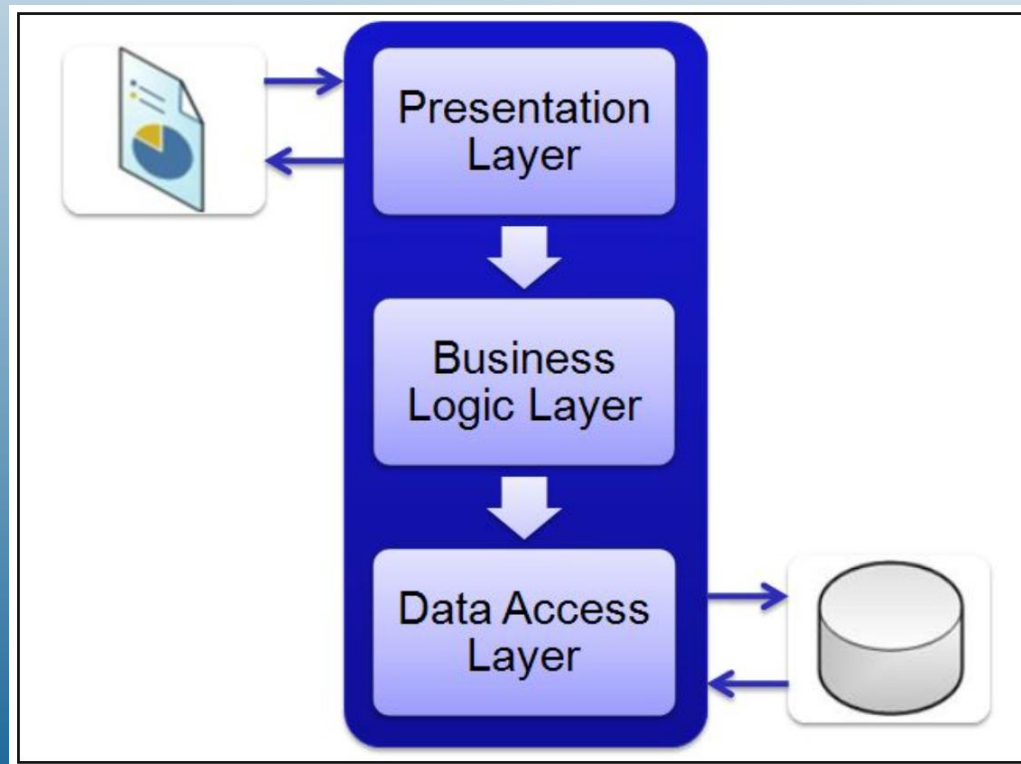
For decades three-tier architecture was the prevailing architecture for client-server applications



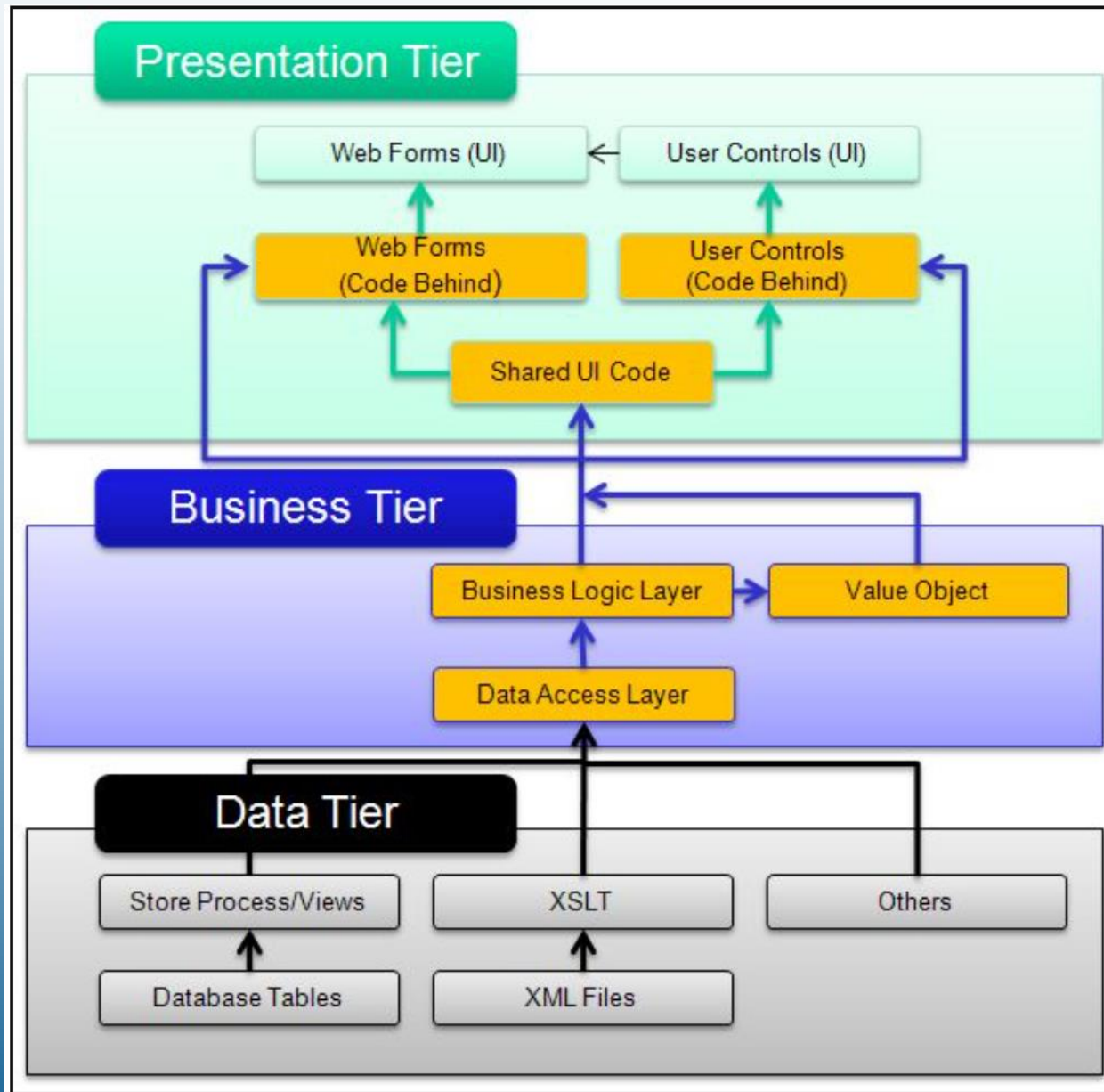
# Three layer architecture

Layer indicates logical separation of components, such as having distinct namespaces and classes for the

- Database Access Layer
- Business Logic Layer (Application layer)
- User Interface Layer (Presentation Layer).



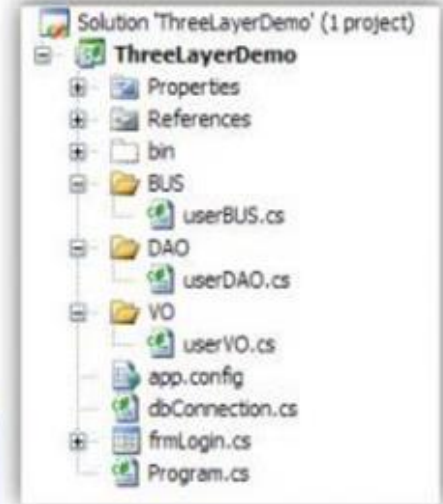
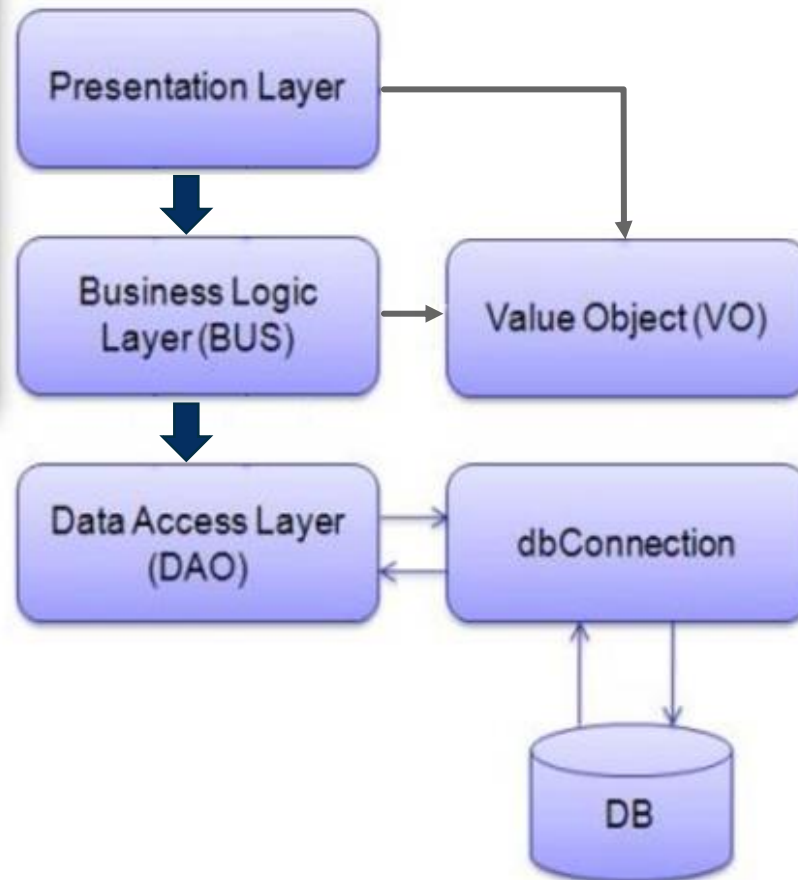
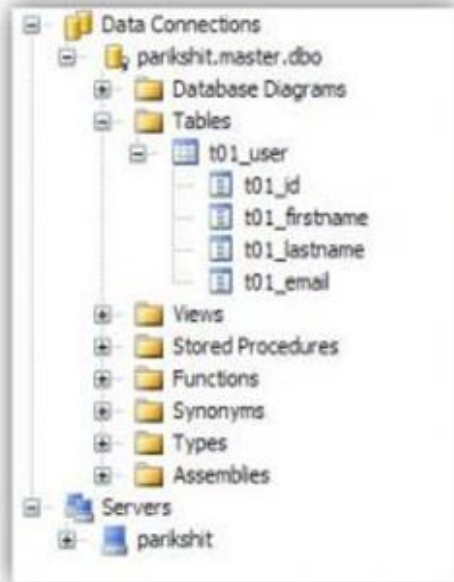
# Three tier architecture



# Three tier architecture

- Data Tier is basically the server which stores all the application's data. Data tier contents Database Tables, XML Files and other means of storing Application Data.
- Business Tier is mainly working as the bridge between Data Tier and Presentation Tier. All the Data passes through the Business Tier before passing to the presentation Tier. Business Tier is the sum of Business Logic Layer, Data Access Layer and Value Object and other components used to add business logic.
- Presentation Tier is the tier in which the users interact with an application. Presentation Tier contents Shared UI code, Code Behind and Designers used to represent information to user.

# Three layer architecture in Visual Studio



# Three tier architecture for Web development

- The **web server** is the **presentation tier** and provides the user interface. This is usually a web page or web site, such as an ecommerce site where the user adds products to the shopping cart, adds payment details or creates an account. The content can be static or dynamic, and is usually developed using HTML, CSS and Javascript .
- The **application server** corresponds to the **middle tier**, housing the business logic used to process user inputs. To continue the ecommerce example, this is the tier that queries the inventory database to return product availability, or adds details to a customer's profile. This layer often developed using Python, Ruby or PHP and runs a framework such as Django, Rails, Symphony or ASP.NET, for example.
- The **database server** is the **data or backend tier** of a web application. It runs on database management software, such as MySQL, Oracle, DB2 or PostgreSQL, for example.

# MVC Model-View-Controller

Trygve Reenskaug invented MVC. The first reports on MVC were written when he was visiting a scientist at Xerox Palo Alto Research Laboratory (PARC) in 1978/79. The goal of Tygrve was to solve the problem of users controlling a large and complex data set.

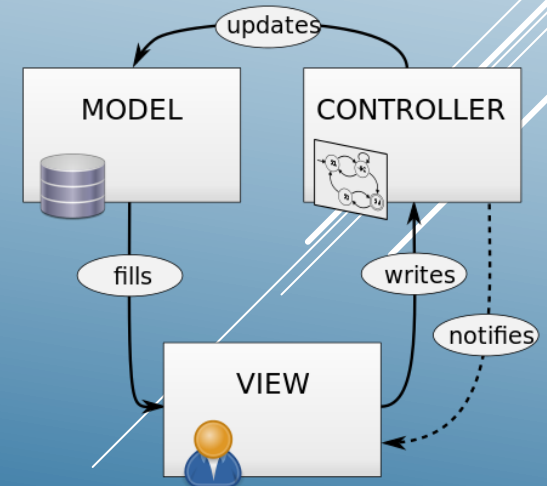
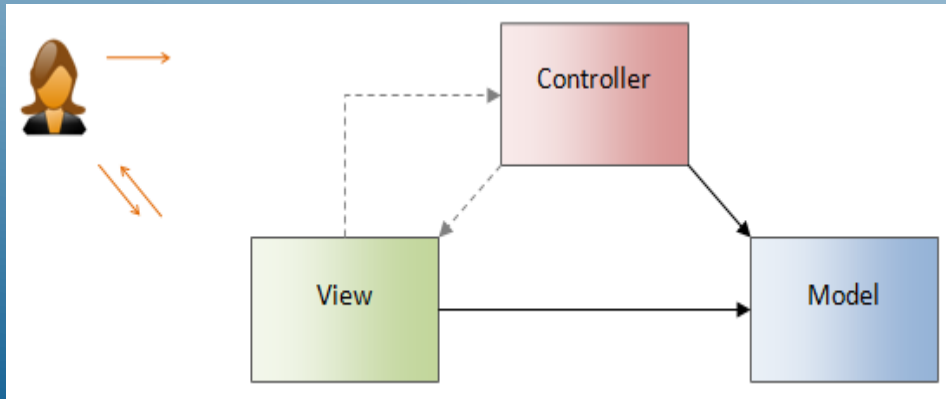
MVC is known as an architectural pattern, which embodies three parts Model, View and Controller, or to be more exact it divides the application into three logical parts: the model part, the view and the controller. It was used for desktop graphical user interfaces but nowadays is used in designing mobile apps and web apps.

Currently MVC it's used for designing web applications. Some web frameworks that use MVC concept: Ruby on Rails, Laravel, Zend framework, CherryPy, Symphony, etc



# MVC Model-View-Controller

- Separation of layers at the logical level
- Dependency minimization, isolated modification of individual layers
- **Model** - it is known as the lowest level which means it is responsible for maintaining data.
- **View** - data representation is done by the view component.
- **Controller** - it's known as the main man because the controller is the component that enables the interconnection between the views and the model so it acts as an intermediary.






# MVC vs 3-layer/tier architecture


At first glance, the three tiers may seem similar to the model-view-controller (MVC) concept; however, topologically they are different.

- A fundamental rule in a three tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middle tier.
- Conceptually the **three-tier architecture is linear**. However, the [model-view-controller]
- **MVC architecture is triangular**: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model

# MVC - misunderstanding

- Usually does not address data access (in the sense of database access) at all.
  - There are variations for different platforms and situations.
  - It should be understood as a very general (and very correct) architectural concept.
  - Not to be interchanged with a three-layer architecture (which is linear).
- 
- A series of three parallel white diagonal lines in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

# Patterns close to MVC

- **Observer and Data Binding** (event-based solutions)
  - **Presentation Model** and its variant **Model View ViewModel** aka MVVM (Microsoft)
  - **Model View Presenter** (Supervising Controller and Passive View)
  - **Page Controller and Front Controller** ( for medium and complex web solutions)
- 

# Patterns of "enterprise" architecture


- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1995.

## **Martin Fowler. Patterns of Enterprise Application Architecture. 2003.**

- David Trowbridge, Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk, David Lavigne. Enterprise Solution Patterns Using Microsoft .NET. 2003.
- Adam Bien. Real World Java EE Patterns – Rethinking Best Practices. 2009.

## ► Domain Logic Patterns

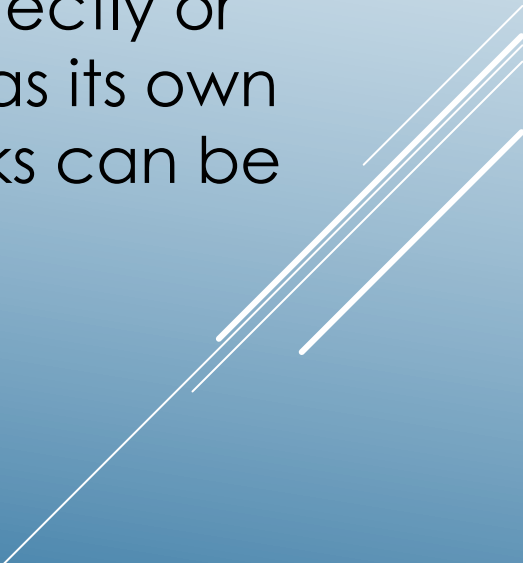
Domain logic or Business logic is the part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed.

- Transaction Script
  - Domain Model
  - Table Module
  - Service Layer
- 
- A series of four parallel white diagonal lines in the bottom right corner of the slide, slanting upwards from left to right.

# Transaction script pattern

Most business applications can be represented as a set of transactions. Some of them choose the data, some - change. Each user and system interaction contains a specific set of actions. In some cases, this may simply be a data output from the database. In other cases, these actions can contain many calculations and checks.

The Transaction Script pattern organizes all this logic into one procedure, working in the database directly or through a thin wrapper. Each transaction has its own Transaction Script, although general subtasks can be broken down into procedures.

Several white lines of varying lengths and slopes are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.

# Transaction script pattern

Organizes business logic by procedures where each procedure handles a single request from the presentation.


```
recognizedRevenue(contractNumber: long, asOf: Date) : Money  
calculateRevenueRecognitions(contractNumber long) : void
```



# Domain model pattern

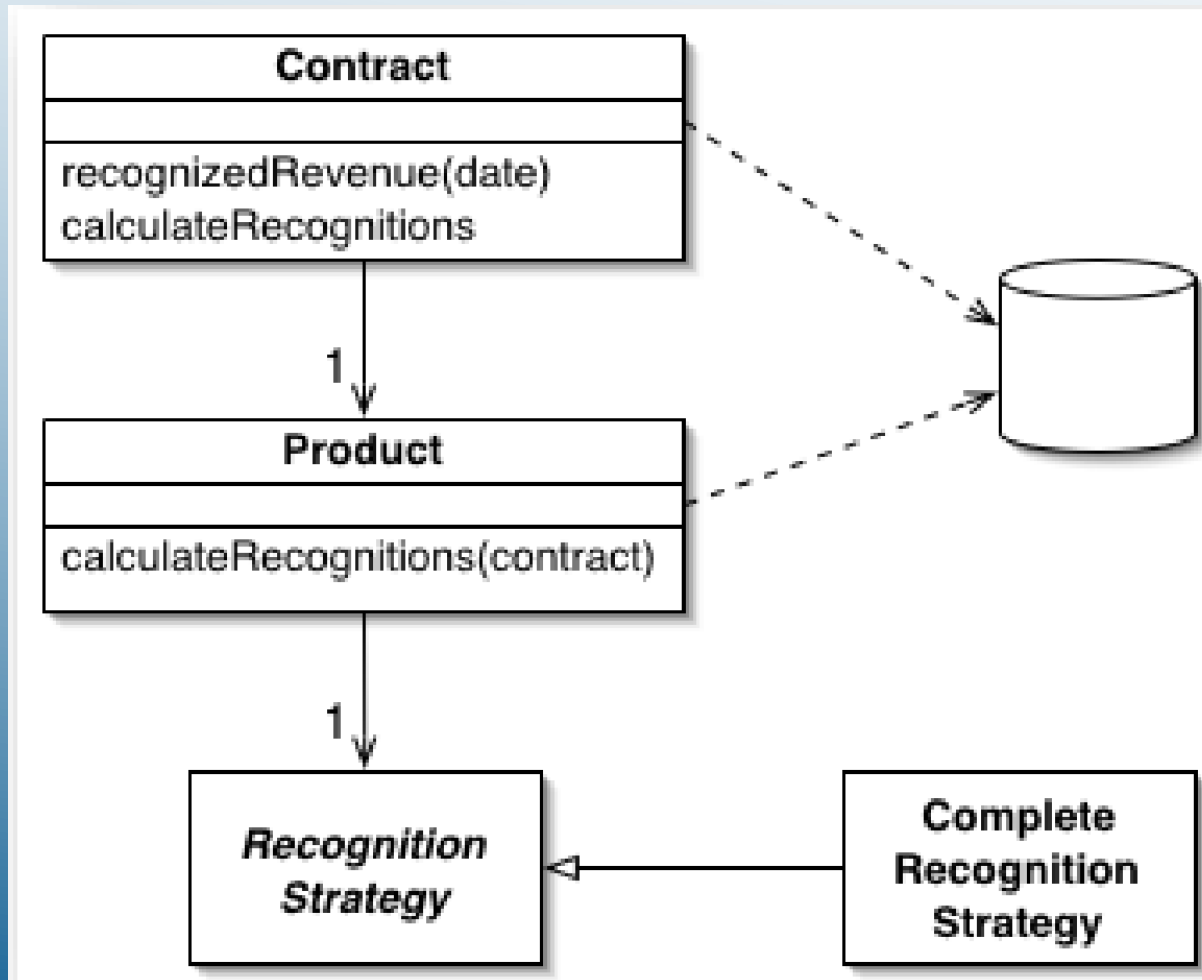
Domain object model that combines data and behavior.

The business logic of an application can be very complex. Rules and logic describe a variety of cases and behavior modifications. To handle all this complex logic and designed objects. The Domain Model pattern (domain definition model) forms a network of interconnected objects, in which each object is a separate significant entity: it can be as big as a corporation or as small as the line from the order form.



# Domain model pattern

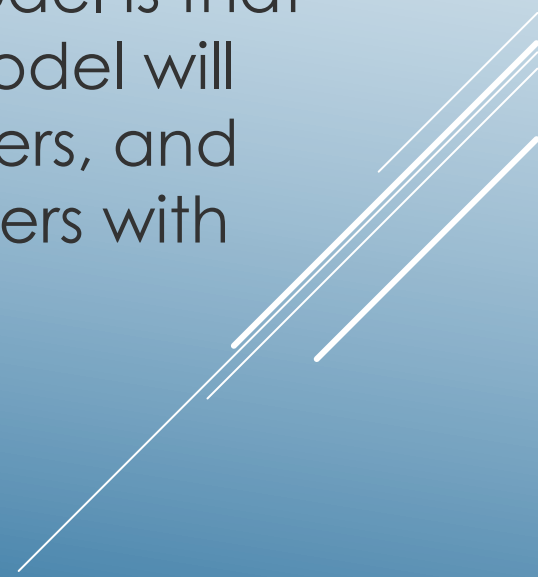
An object model of the domain that incorporates both behavior and data.



# Table module pattern

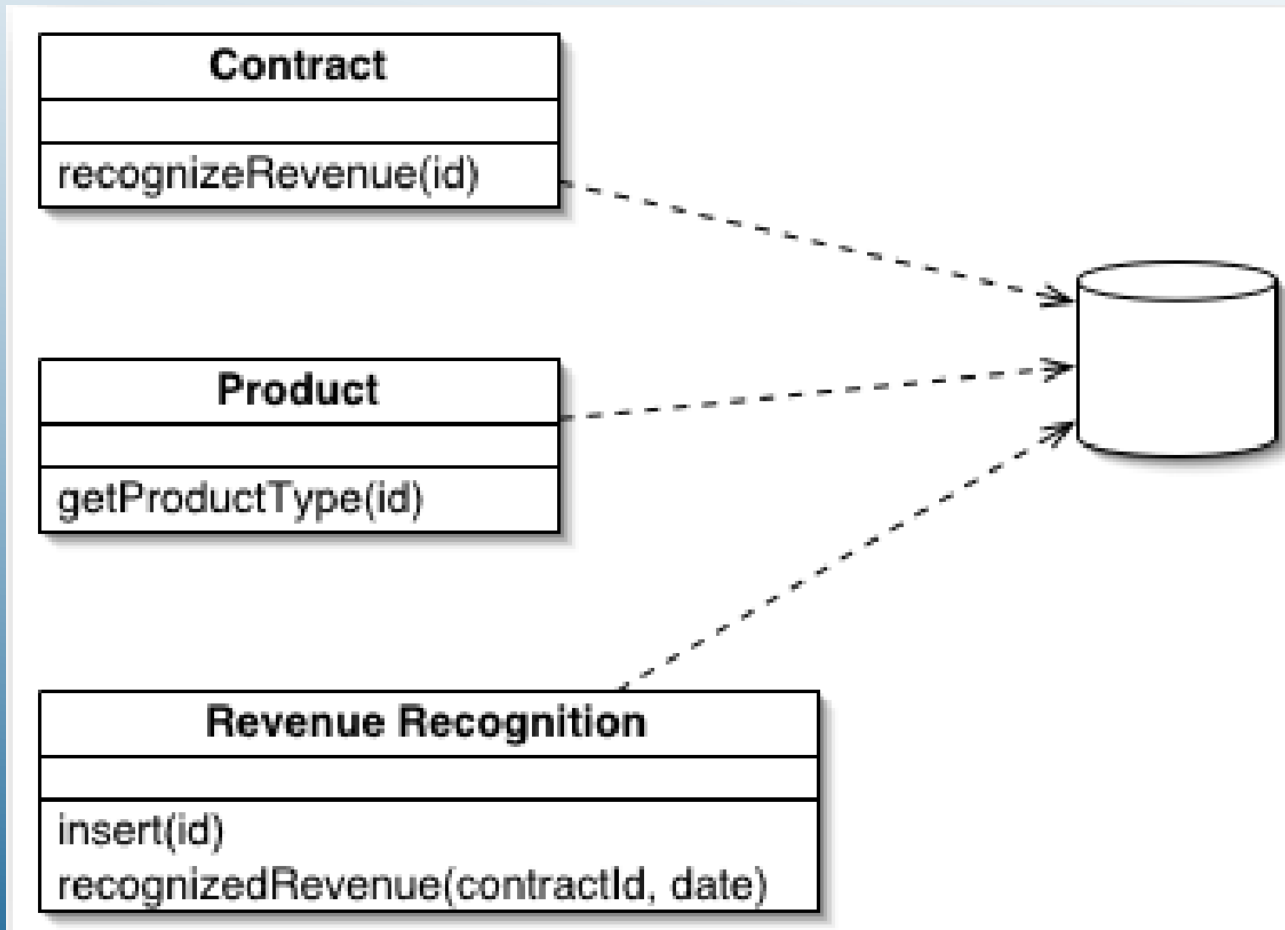
The Table Module pattern divides the logic of the definition domain (domain) into separate classes for each table in the database and one instance of the class contains various procedures that work with data.

The main difference from the Domain Model is that if there are several orders, the Domain Model will create its own object for each of the orders, and the Table Module will manage all the orders with one object.

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the bottom left.

# Table module pattern


A single instance that handles the business logic for all rows in a database table or view.



# **Service Layer (*Service Level*) pattern**

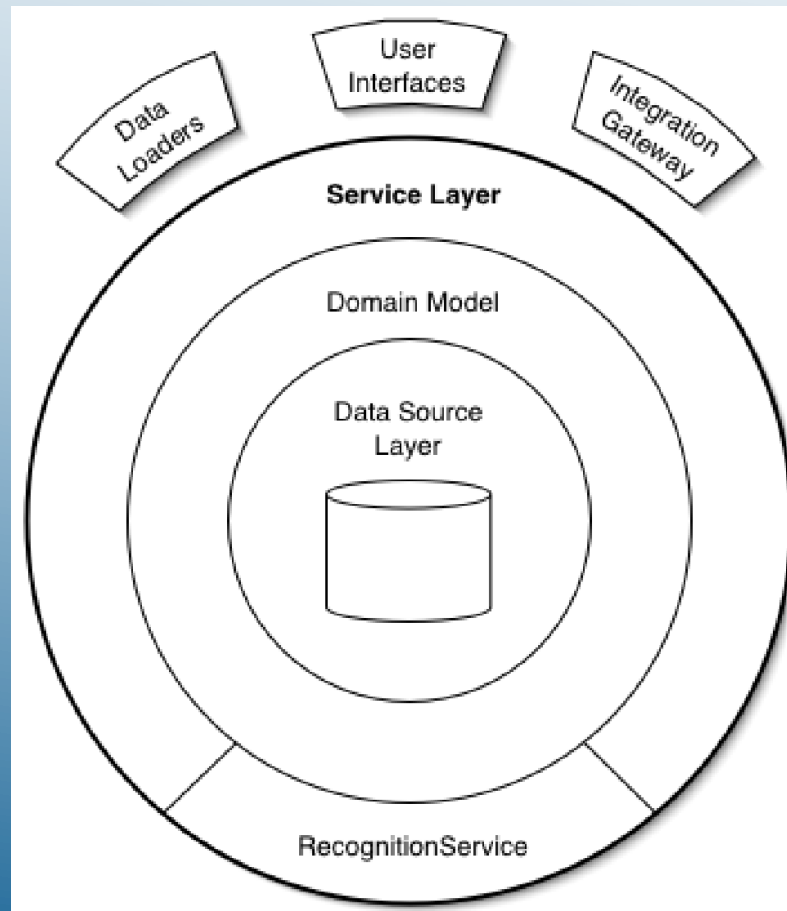
The Service Layer pattern defines for the application a boundary and a set of allowed operations from the point of view of client interactions with it.

It encapsulates the business logic of the application, managing transactions and managing responses in the implementation of these operations.

A series of three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

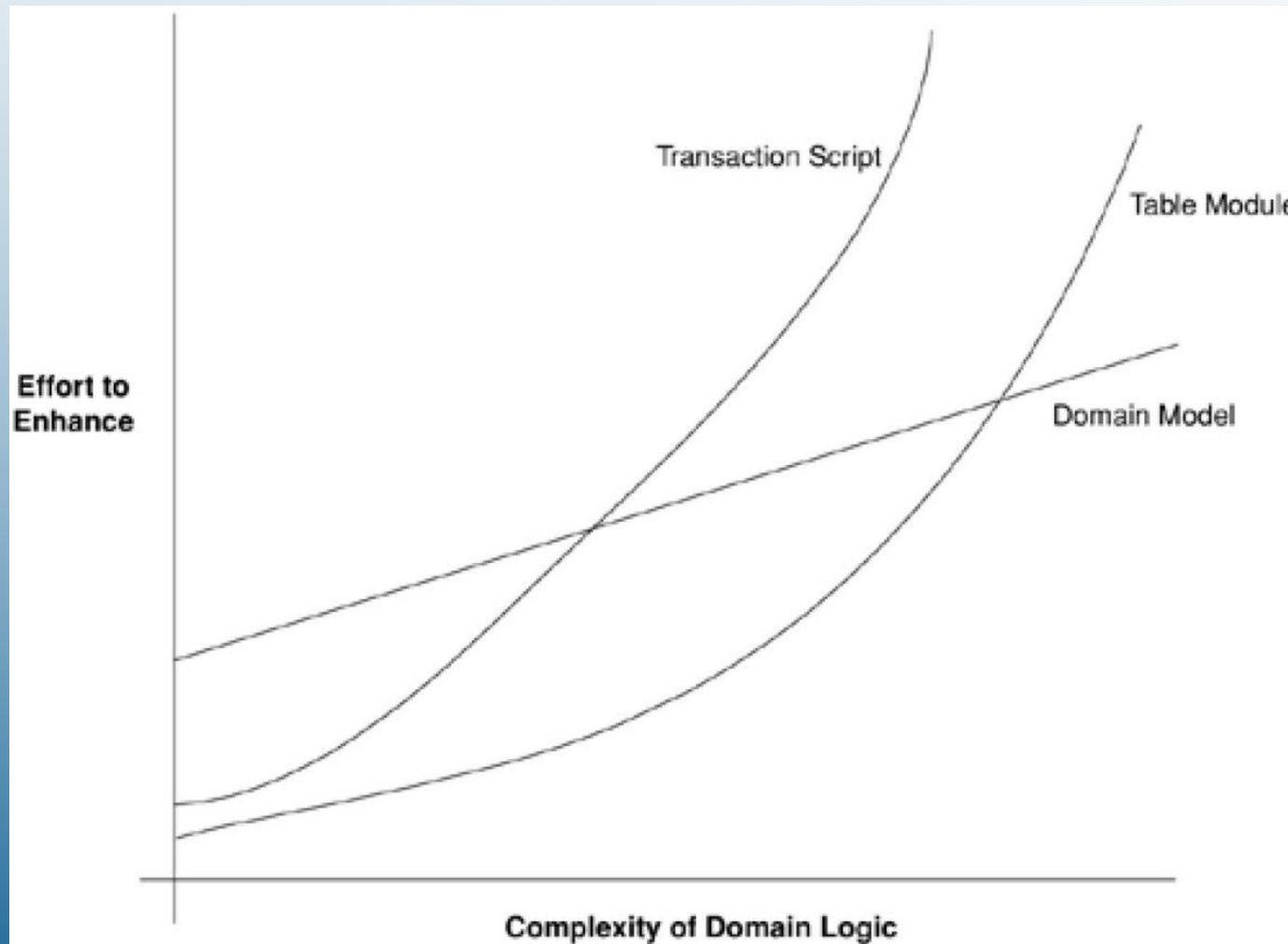
# Service Layer (*Service Level*) pattern

Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.



# When to use which pattern?

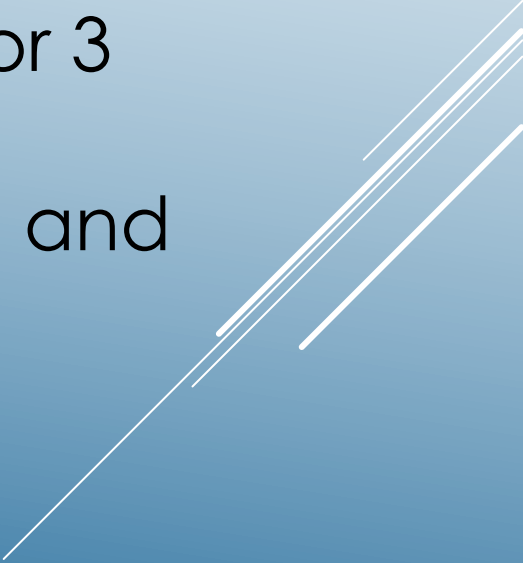
It depends on equilibrium between complexity and effort to enhance





# Semestral project -Artifact 2

## Specification of requirements:

- Functional requirements in the form of a use-case model
  - Use-case diagram (whole system)
  - Use-case (structured description for 3 scenarios)
  - Activity diagram (one to use-case and one through use-case)
- 

# Exercise tasks

Re-implement the task from the previous exercise using the patterns from the lecture

- Transaction script
- Domain model
- Module for table

# Lecture checking questions

- What is meant by a design pattern? What does each pattern contain? Give examples.
- Describe the essence of three-layer architecture. What is the difference between a physical and a logical three-layer architecture?
- What is the essence of the MVC pattern? How does it differ from the three-layer architecture?
- What does the domain logic design pattern group address?
- Describe the essence of each domain logic pattern (Transaction script, Domain model, Table module, Service layer).
- How do the different domain logic patterns differ from each other? When, where and why to use/not to use them? Give examples.