Architectural Patterns

Multi-Tier, MVC, MVP, MVVM, IoC, DI, SOA

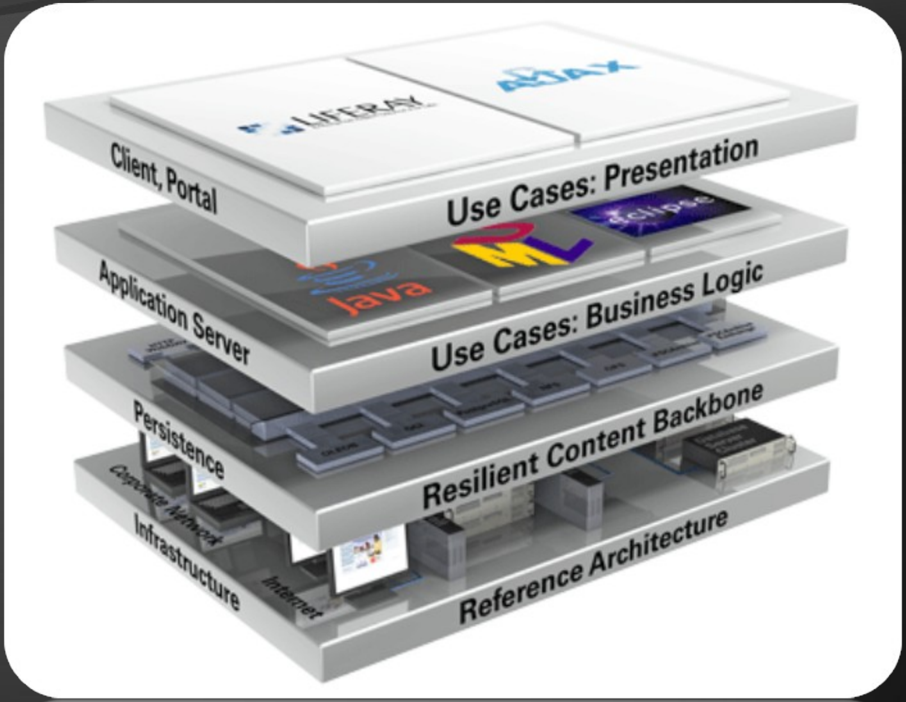
Svetlin Nakov

Telerik Corporation

www.telerik.com

Table of Contents

1. What is Software Architecture?
2. Client-Server Architecture
3. 3-Tier / Multi-Tier Architectures
4. MVC (Model-View-Controller)
5. MVP (Model-View-Presenter)
6. MVVM (Model-View-ViewModel)
7. IoC (Inversion of Control) and DI (Dependency Injection) Architectural Principals
8. SOA (Service-Oriented Architecture)



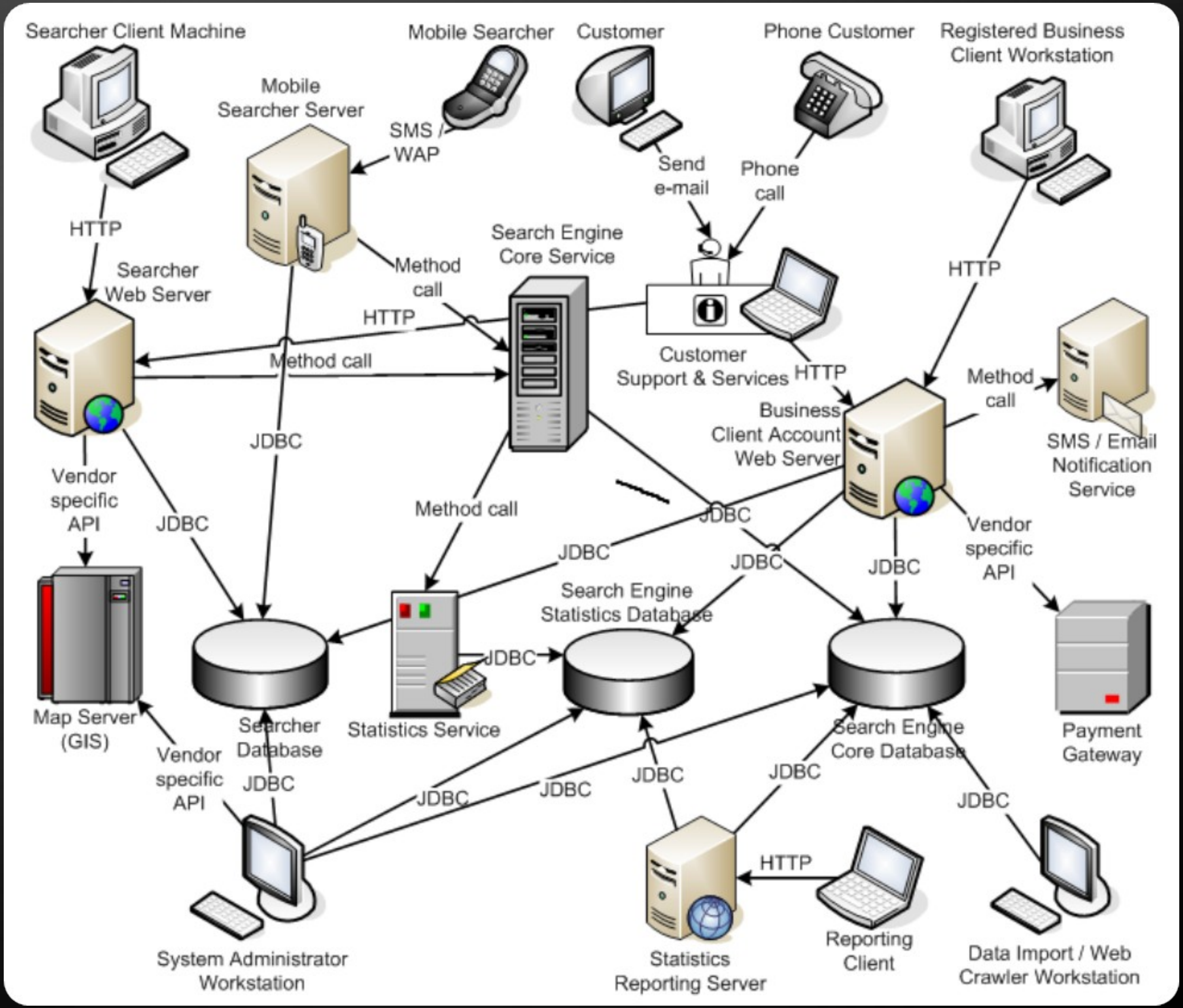
What is Software Architecture?

Software Architecture

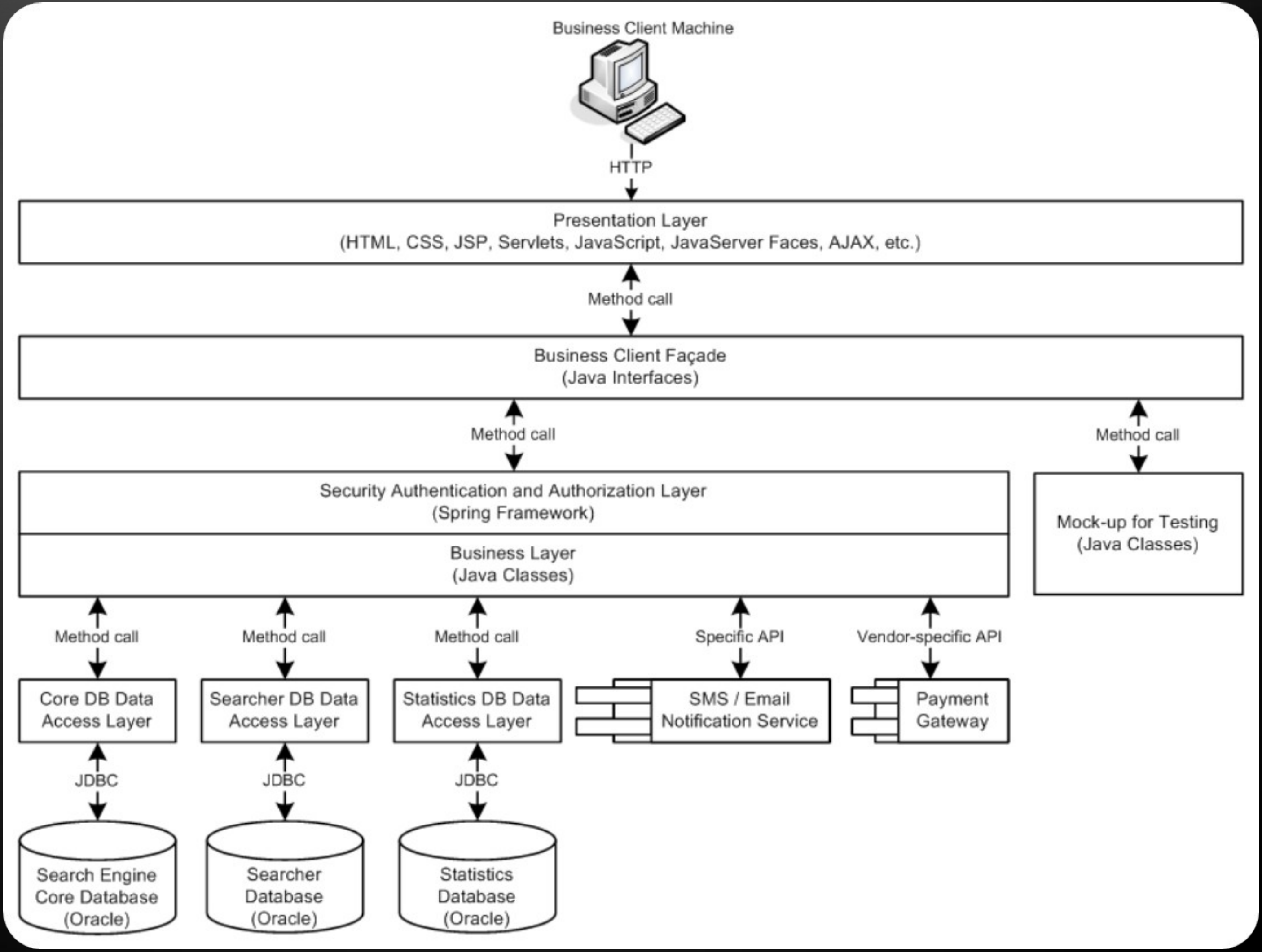
- Software architecture is a technical blueprint explaining how the system will be structured
- The system architecture describes:
  - How the system will be decomposed into subsystems (modules)
  - Responsibilities of each module
  - Interaction between the modules
  - Platforms and technologies

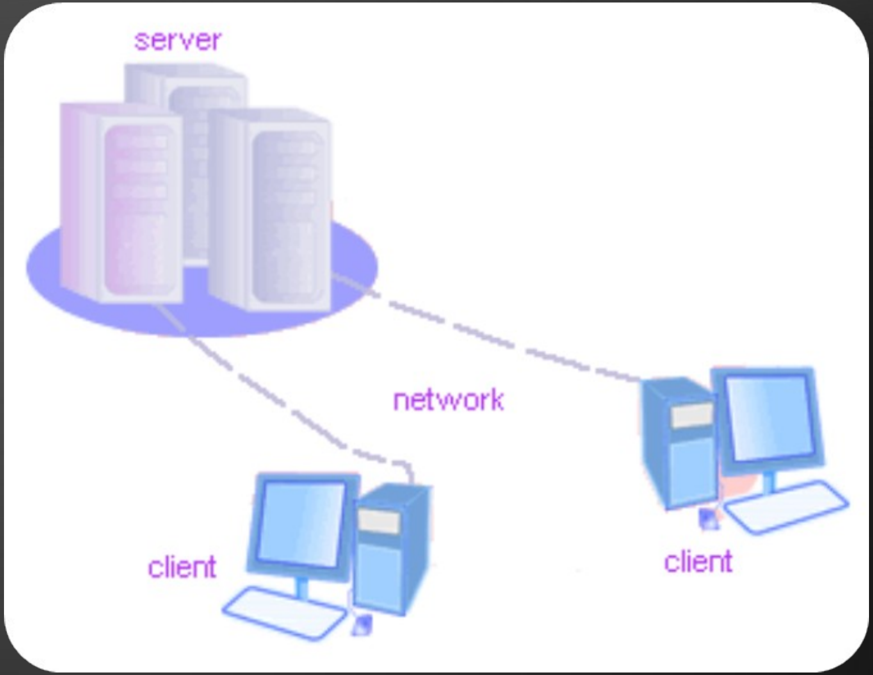
- Each module could also implement a certain architectural model / pattern

System Architecture Diagram – Example



Example of Multi-Tier Software Architecture


Business Client Machine
HTTP
Presentation Layer
(HTML, CSS, JSP, Servlets, JavaScript, JavaServer Faces, AJAX, etc.)
Method call
Business Client Façade
(Java Interfaces)
Method call
Method call
Security Authentication and Authorization Layer
(Spring Framework)
Business Layer
(Java Classes)
Mock-up for Testing
(Java Classes)
Method call
Method call
Method call
Specific API
Vendor-specific API
Core DB Data Access Layer
Searcher DB Data Access Layer
Statistics DB Data Access Layer
SMS / Email Notification Service
Payment Gateway
JDBC
JDBC
JDBC
Search Engine Core Database (Oracle)
Searcher Database (Oracle)
Statistics Database (Oracle)



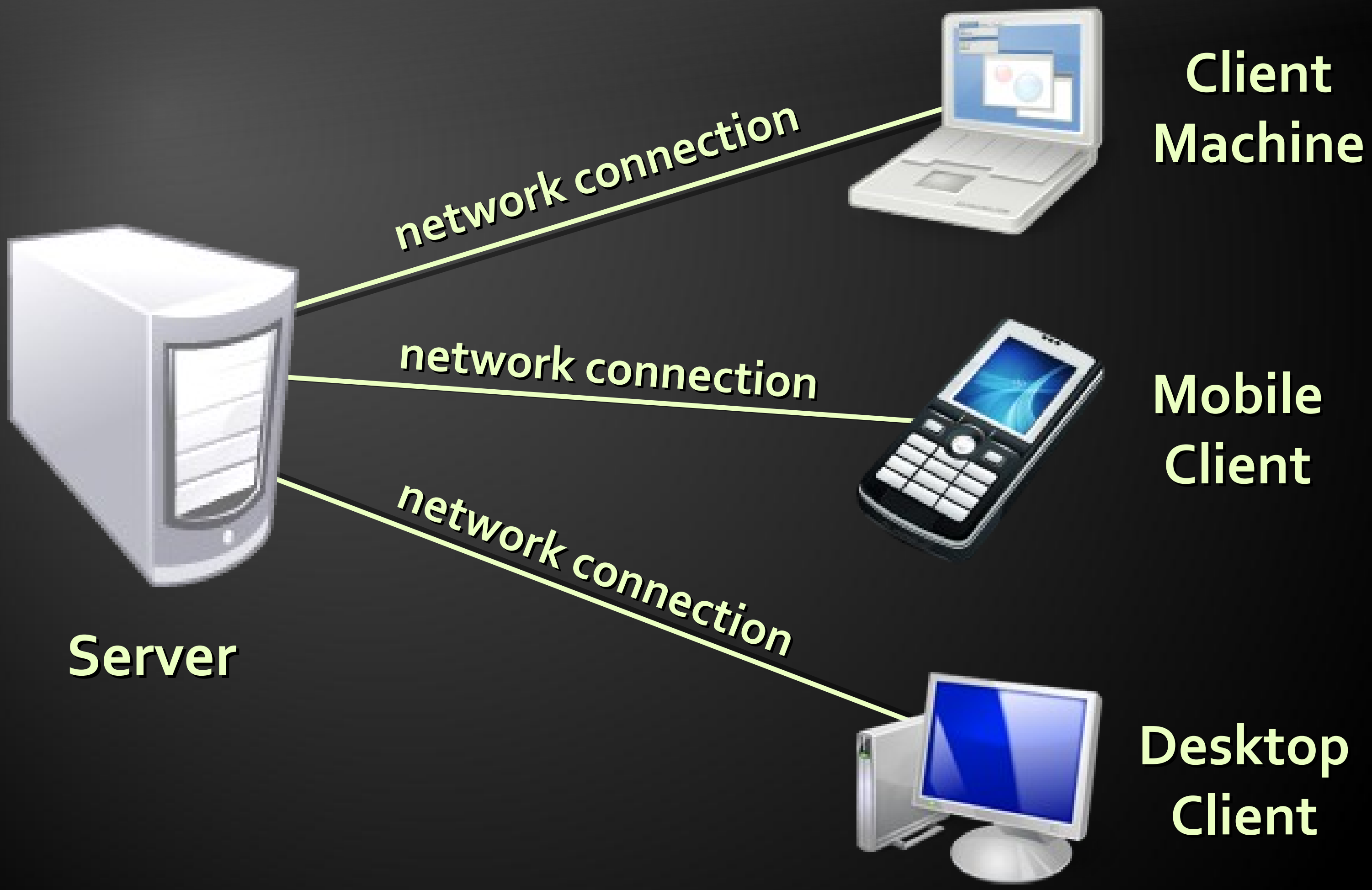
Client-Server Architecture

The Classical Client-Server Model

Client-Server Architecture

- The client-server model consists of:
  - Server – a single machine / application that provides services to multiple clients
    - Could be IIS based Web server
    - Could be WCF based service
    - Could be a services in the cloud
  - Clients –software applications that provide UI (front-end) to access the services at the server
    - Could be WPF, HTML5, Silverlight, ASP.NET, …

The Client-Server Model

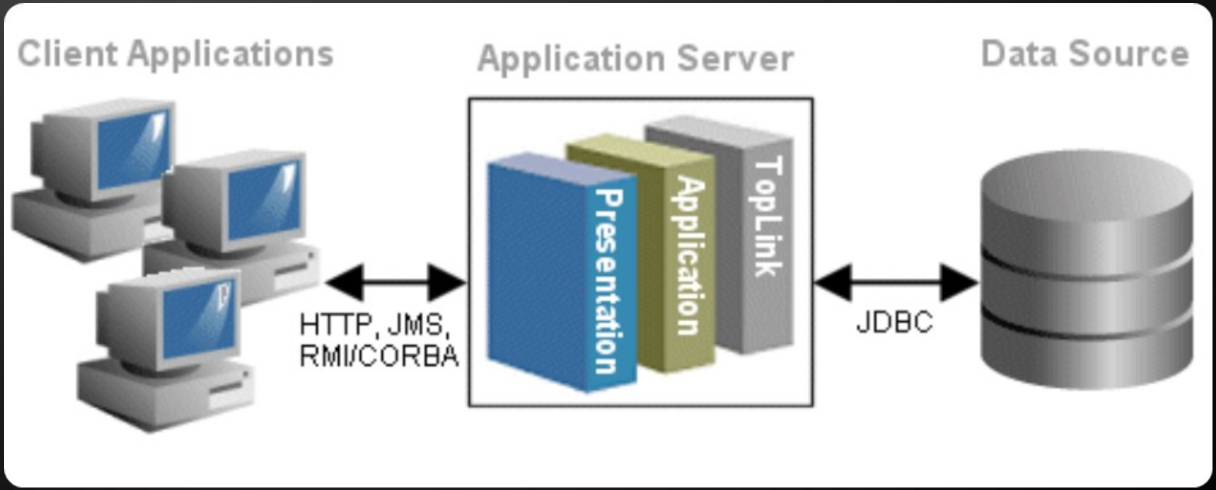


Client-Server Model – Examples

- Web server (IIS) – Web browser (Firefox)
- FTP server (ftpd) – FTP client (FileZilla)
- EMail server (qmail) – email client (Outlook)
- SQL Server – SQL Server Management Studio
- BitTorrent Tracker – Torrent client (μTorrent)
- DNS server (bind) – DNS client (resolver)
- DHCP server (wireless router firmware) – DHCP client (mobile phone /Android DHCP client/)
- SMB server (Windows) – SMB client (Windows)

3-Tier / Multi-Tier Architectures

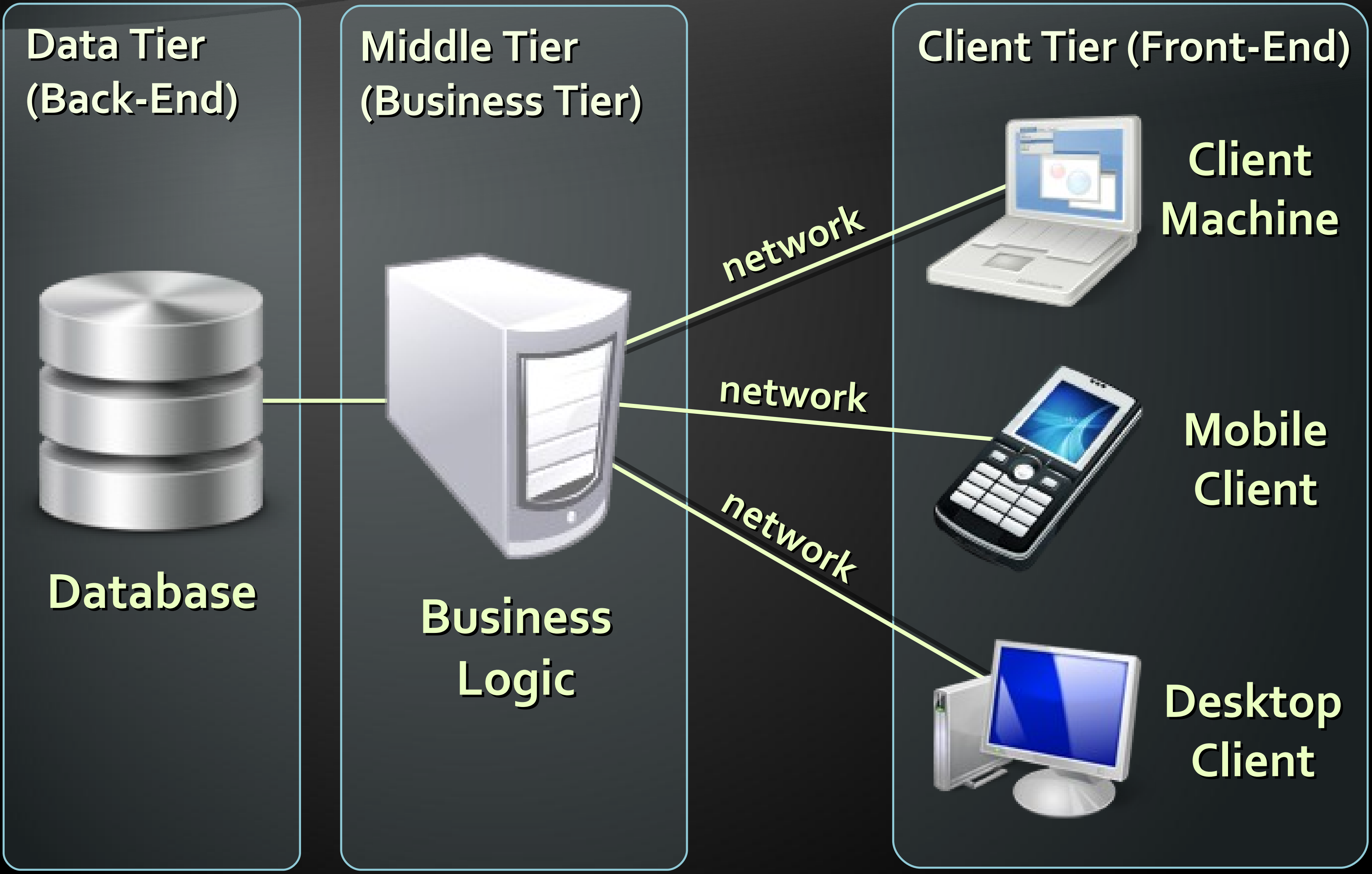
Classical Layered Structure of Software Systems



The 3-Tier Architecture

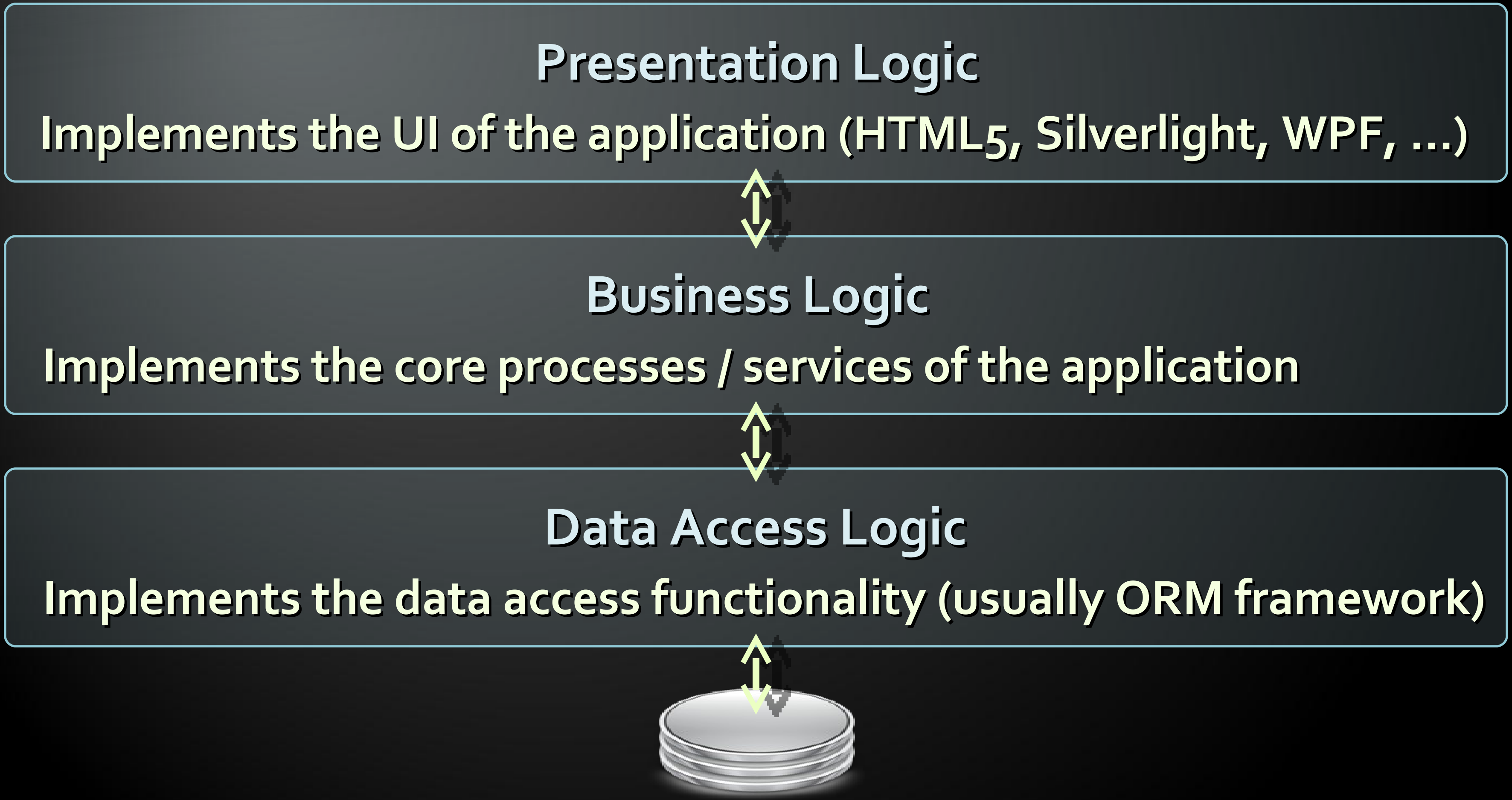
- The 3-tier architecture consists of the following tiers (layers):
  - Front-end (client layer)
    - Client software – provides the UI of the system
  - Middle tier (business layer)
    - Server software – provides the core system logic
    - Implements the business processes / services
  - Back-end (data layer)
    - Manages the data of the system (database / cloud)

The 3-Tier Architecture Model



Typical Layers of the Middle Tier

- The middle tier usually has parts related to the front-end, business logic and back-end:



Presentation Logic

Implements the UI of the application (HTML5, Silverlight, WPF, …)

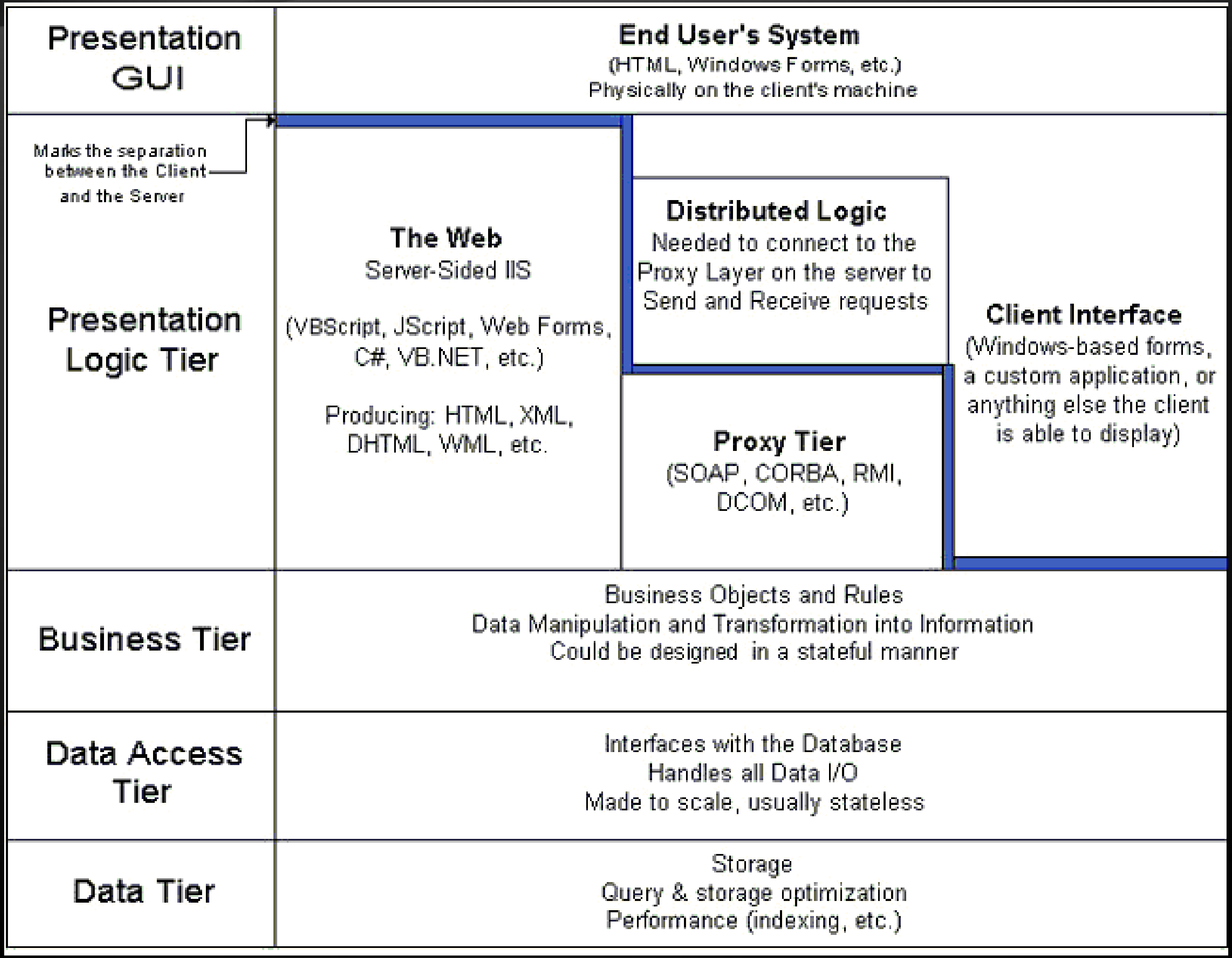
Business Logic

Implements the core processes / services of the application

Data Access Logic

Implements the data access functionality (usually ORM framework)

Multi-Tier Architecture



Presentation GUI

End User's System
(HTML, Windows Forms, etc.)
Physically on the client's machine

Marks the separation between the Client and the Server

Presentation Logic Tier

The Web
Server-Sided IIS

(VBScript, JScript, Web Forms, C#, VB.NET, etc.)

Producing: HTML, XML, DHTML, WML, etc.

Distributed Logic
Needed to connect to the Proxy Layer on the server to Send and Receive requests

Proxy Tier
(SOAP, CORBA, RMI, DCOM, etc.)

Client Interface
(Windows-based forms, a custom application, or anything else the client is able to display)

Business Tier

Business Objects and Rules
Data Manipulation and Transformation into Information
Could be designed in a stateful manner

Data Access Tier

Interfaces with the Database
Handles all Data I/O
Made to scale, usually stateless

Data Tier

Storage
Query & storage optimization
Performance (indexing, etc.)

HTML

ASP .NET

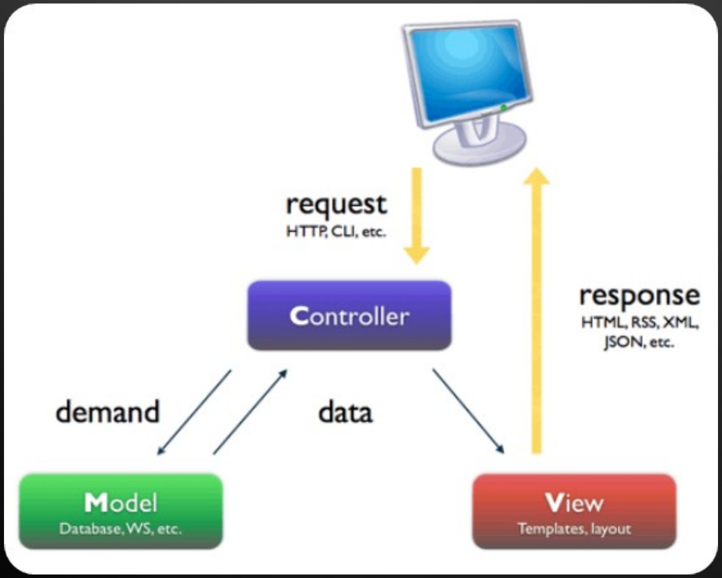
WCF

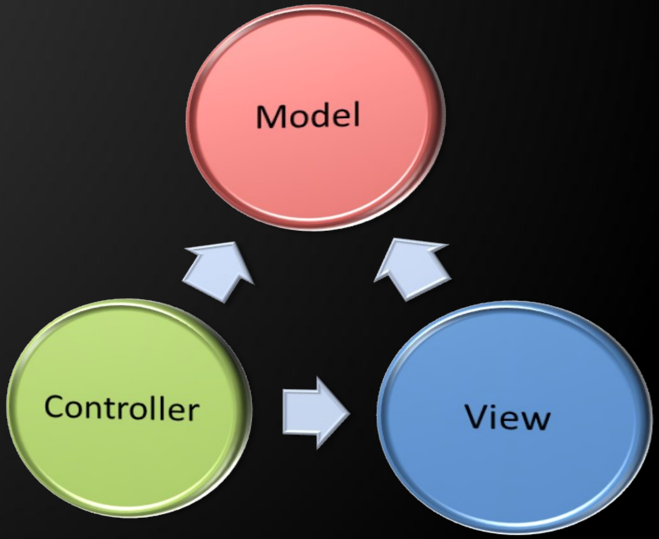
ORM

DB

MVC (Model-View-Controller)

What is MVC and How It Works?

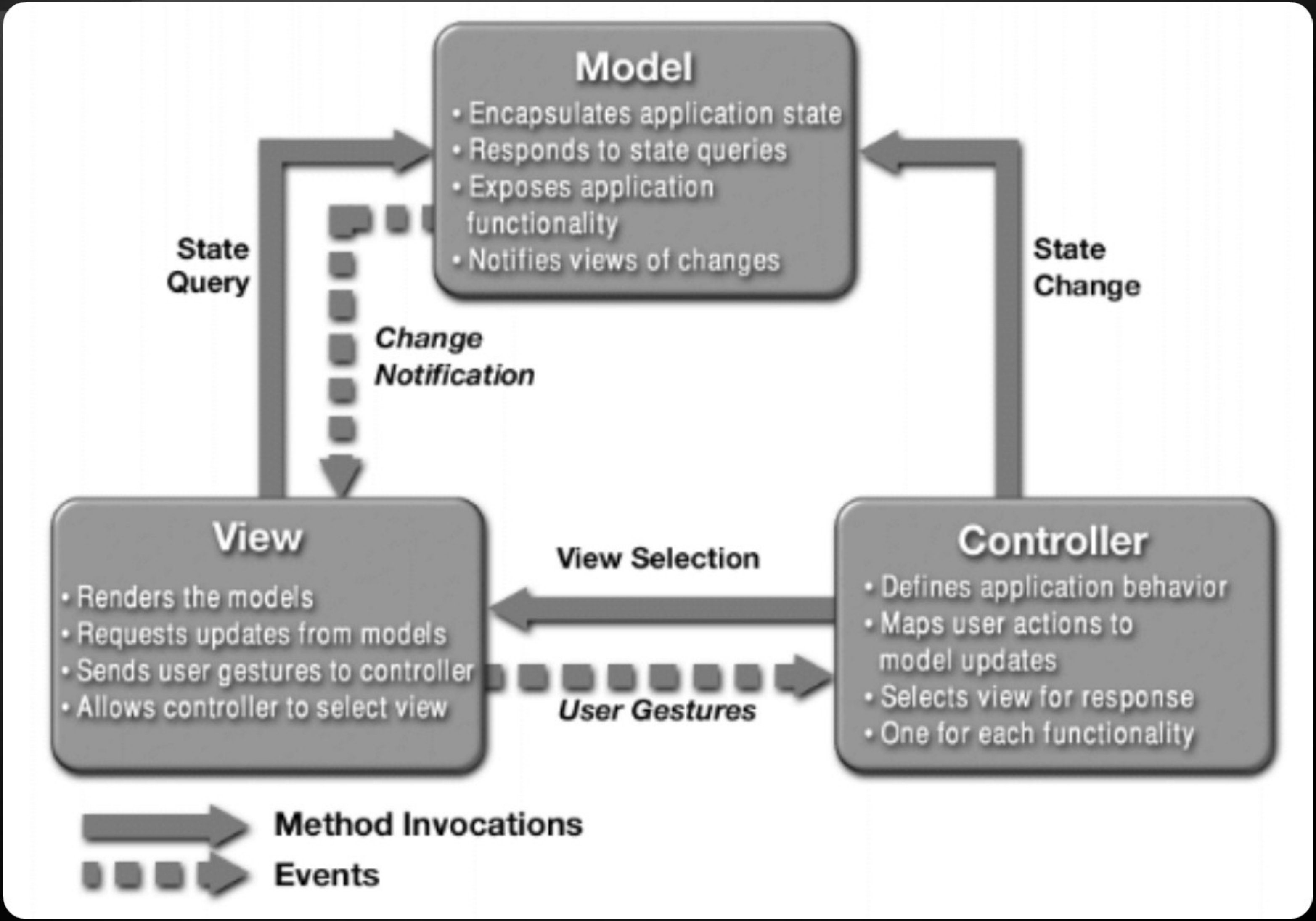




Model-View-Controller (MVC)

- Model-View-Controller (MVC) architecture
  - Separates the business logic from application data and presentation
- Model
  - Keeps the application state (data)
- View
  - Displays the data to the user (shows UI)
- Controller
  - Handles the interaction with the user

MVC Architecture Blueprint



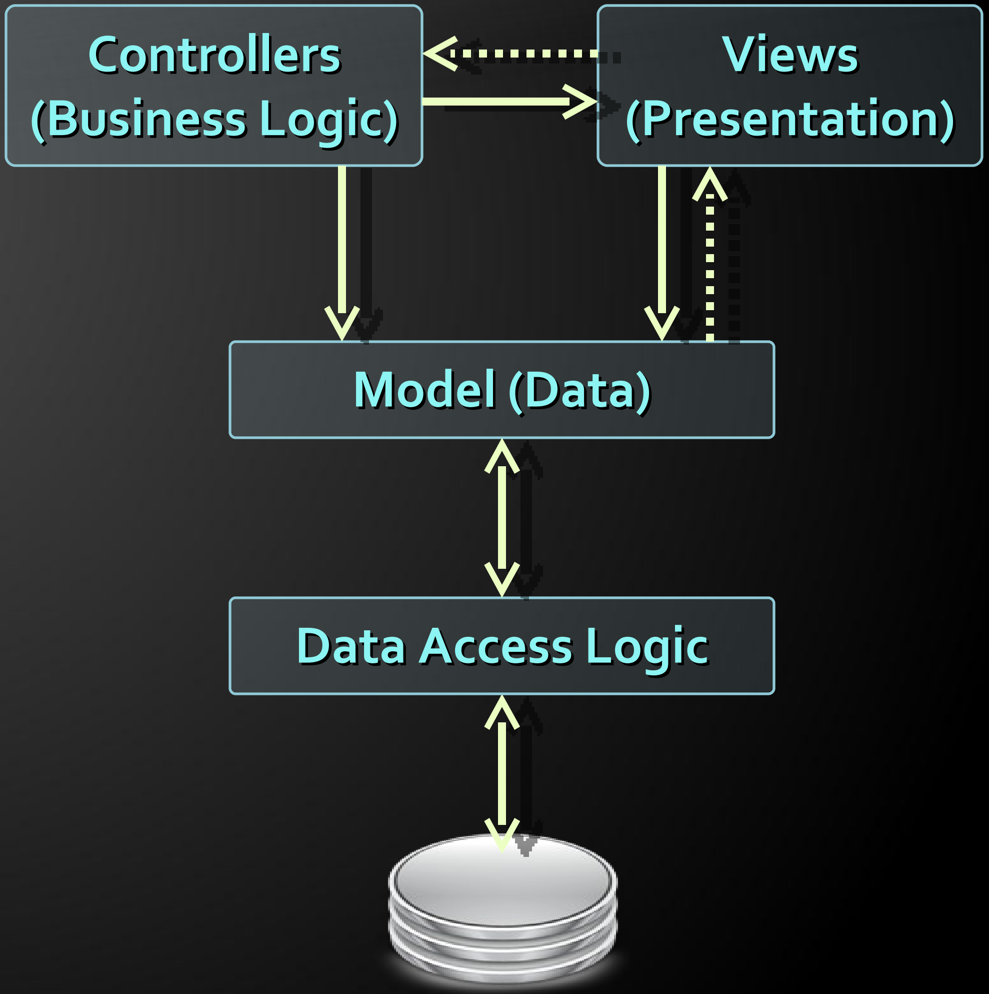
Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

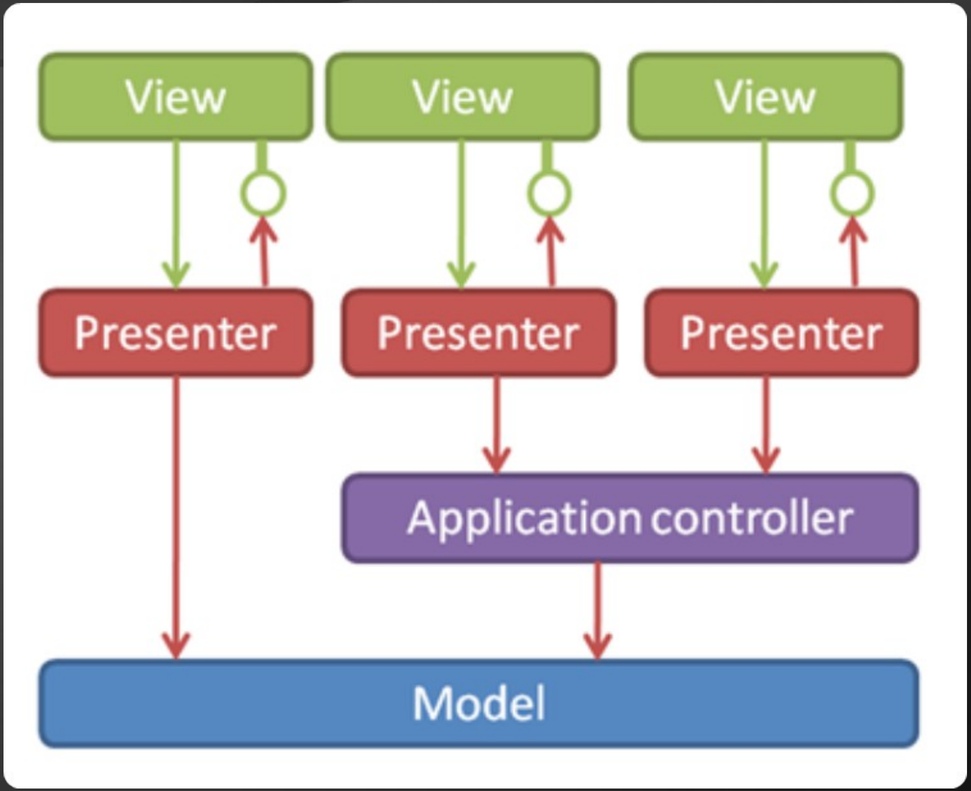
MVC-Based Frameworks

- .NET
  - ASP.NET MVC, MonoRail
- Java
  - JavaServer Faces (JSF), Struts, Spring Web MVC, Tapestry, JBoss Seam, Swing
- PHP
  - CakePHP, Symfony, Zend, Joomla, Yii, Mojavi
- Python
  - Django, Zope Application Server, TurboGears
- Ruby on Rails

MVC and Multi-Tier Architecture

- MVC does not replace the multi-tier architecture
  - Both are usually used together
- Typical multi-tier architecture can use MVC
  - To separate logic, data and presentation



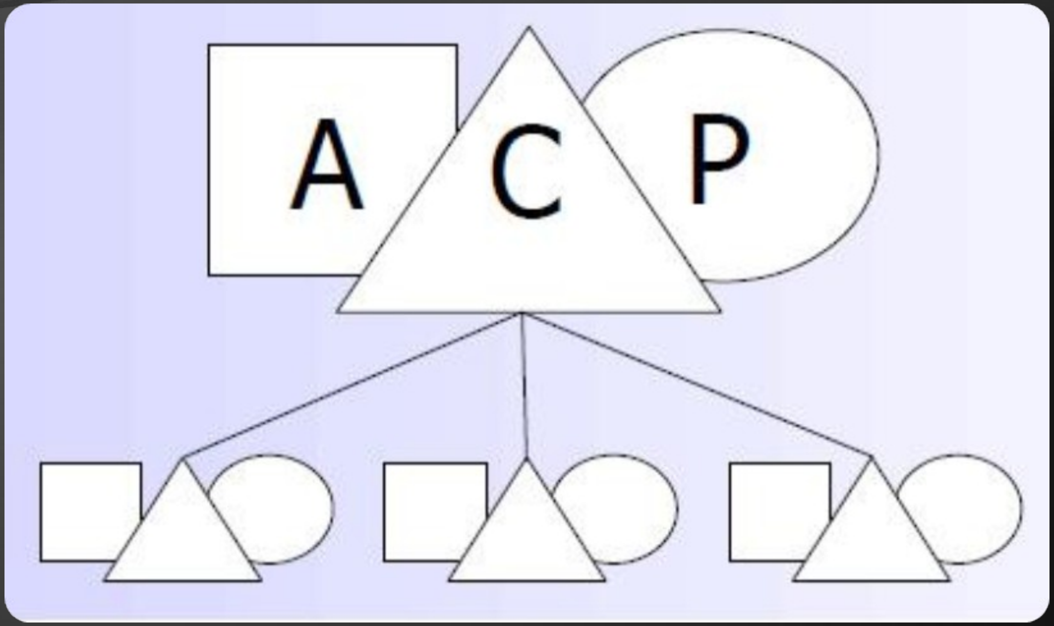


MVP (Model-View-Presenter)

What is MVP Architecture and How it Works?

Model-View-Presenter (MVP)

- Model-View-Presenter (MVP) is UI design pattern similar to MVC
  - Model
    - Keeps application data (state)
  - View
    - Presentation – displays the UI and handles UI events (keyboard, mouse, etc.)
  - Presenter
    - Presentation logic (prepares data taken from the model to be displayed in certain format)



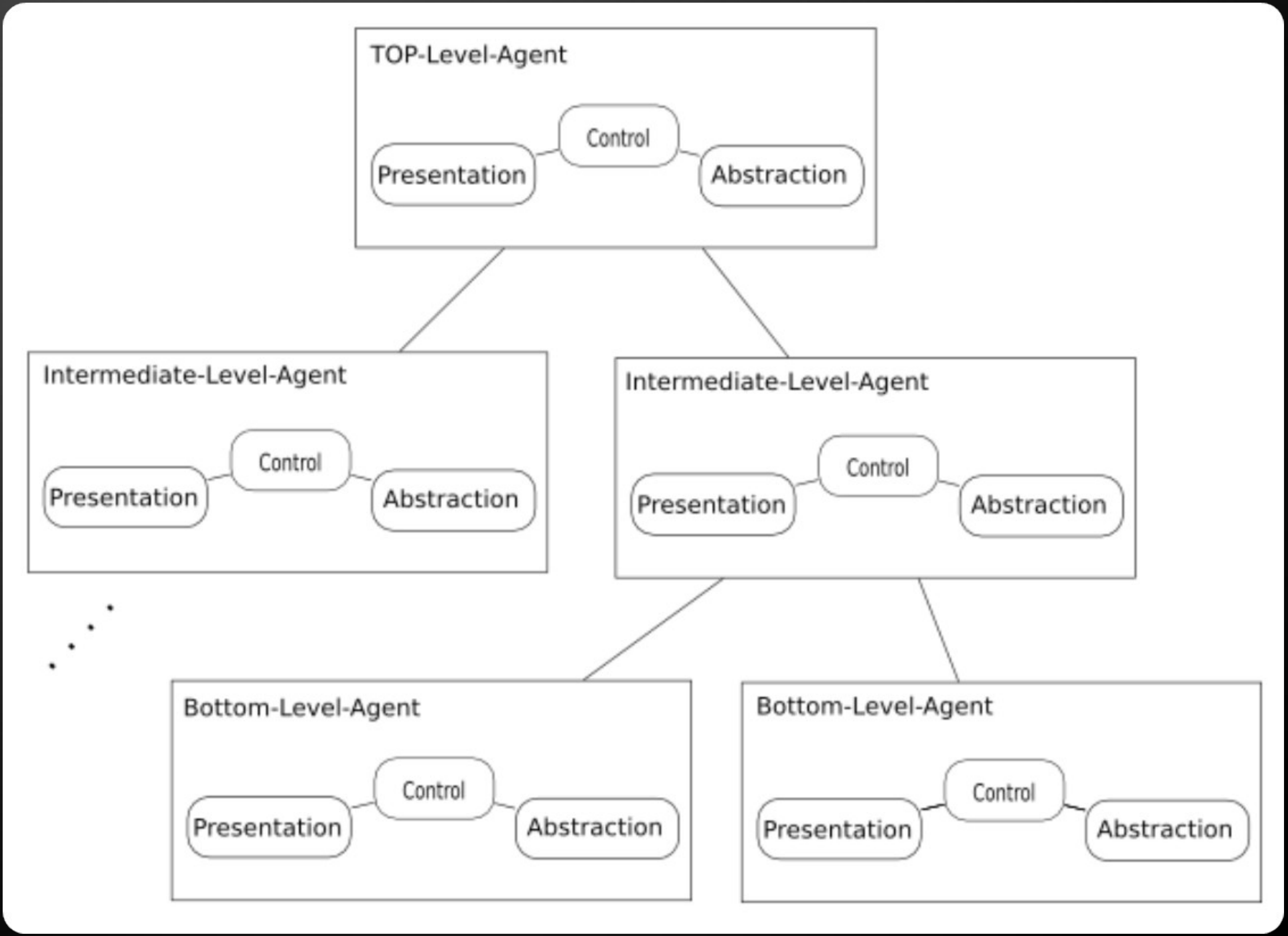
Presentation-Abstraction-Control (PAC)

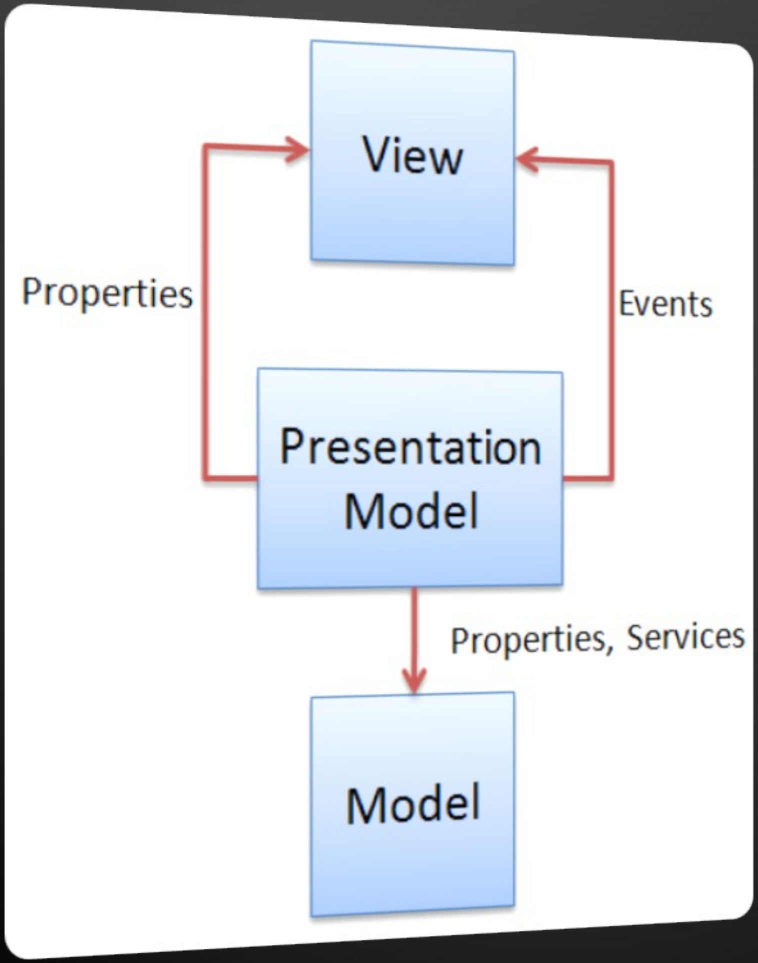
What is PAC and How It Works?

Presentation-Abstraction-Control (PAC)

- Presentation-Abstraction-Control (PAC) interaction-oriented architectural pattern
  - Similar to MVC but is hierarchical (like HMVC)
  - Presentation
    - Prepares data for the UI (similar to View)
  - Abstraction
    - Retrieves and processes data (similar to Model)
  - Control
    - Flow-control and communication (similar to Controller)

Presentation-Abstraction-Control (PAC) – Hierarchy





MVVM (Model-View-ViewModel)

What is MVVM and How It Works?

Model-View-ViewModel (MVVM)

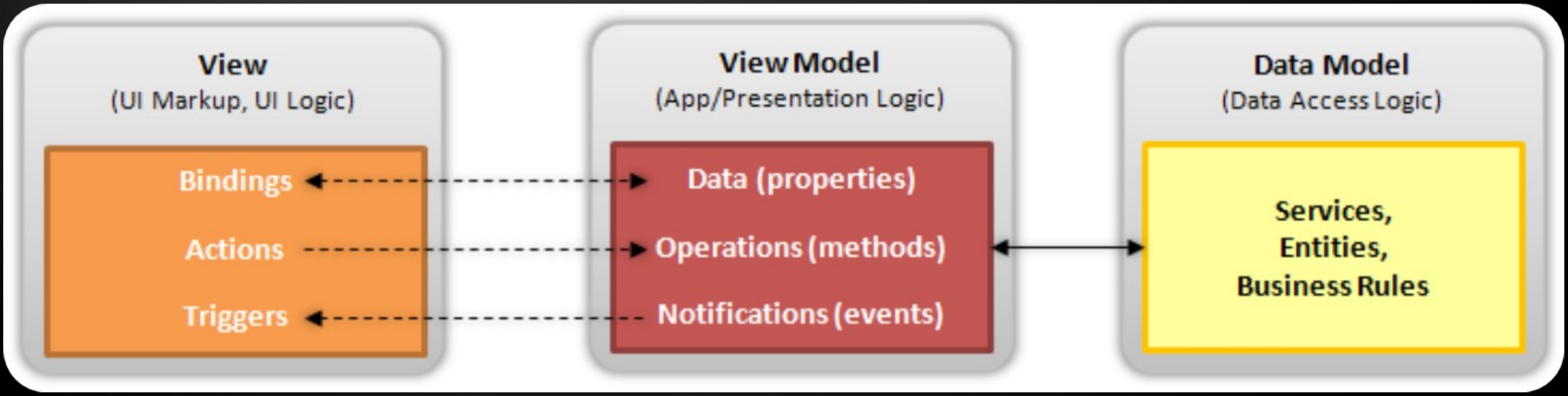
- Model-View-ViewModel (MVVM) is architectural pattern for modern UI development
  - Invented by Microsoft for use in WPF and Silverlight
  - Based on MVC, MVP and Martin Fowler's Presentation Model pattern
  - Officially published in the Prism project (Composite Application Guidance for WPF and Silverlight)
  - Separates the "view layer" (state and behavior) from the rest of the application

MVVM Structure

- Model
  - Keeps the application data / state representation
  - E.g. data access layer or ORM framework
- View
  - UI elements of the application
  - Windows, forms, controls, fields, buttons, etc.
- ViewModel
  - Data binder and converter that changes the Model information into View information
  - Exposes commands for binding in the Views

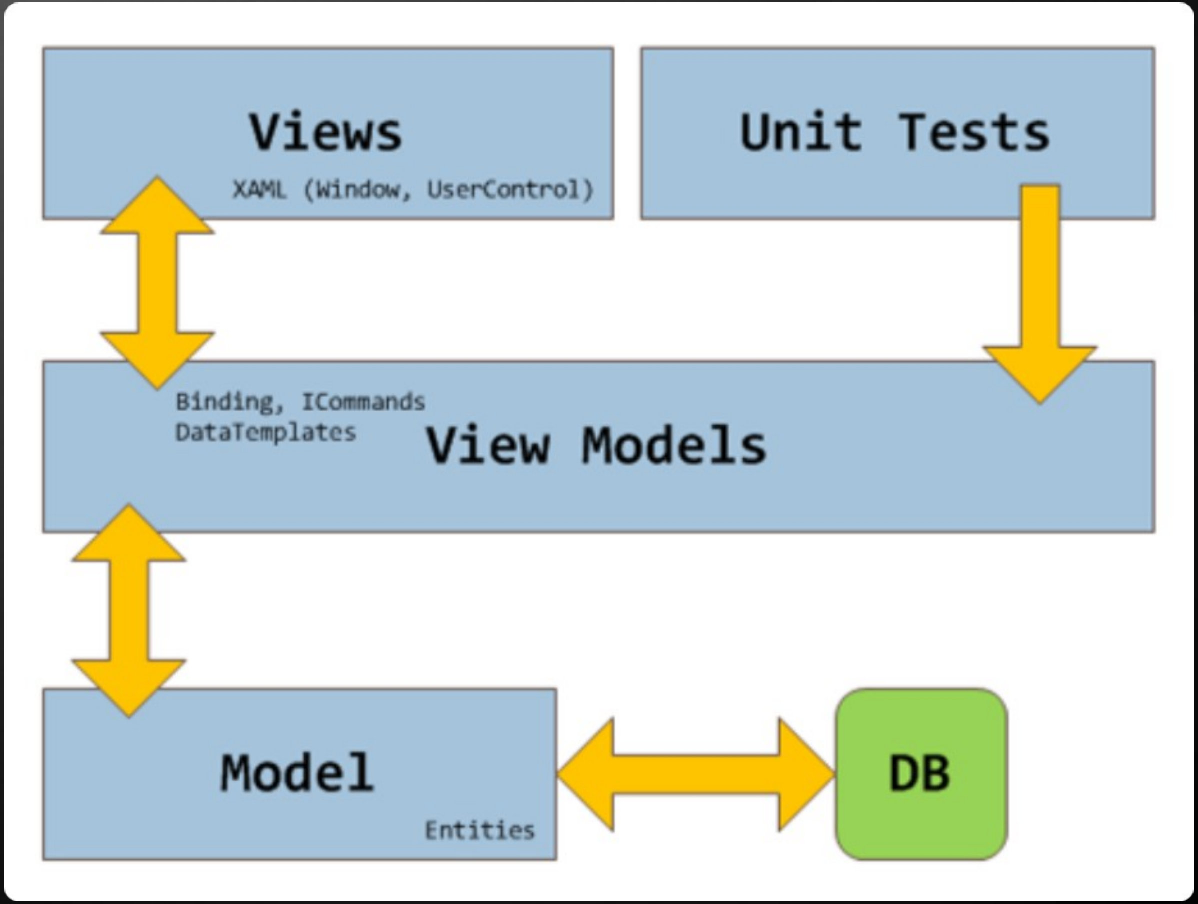
MVVM in WPF / Silverlight

- View – implemented by XAML code + code behind C# class
- Model – implemented by WCF services / ORM framework / data access classes
- ViewModel – implemented by C# class and keeps data (properties), commands (code), notifications

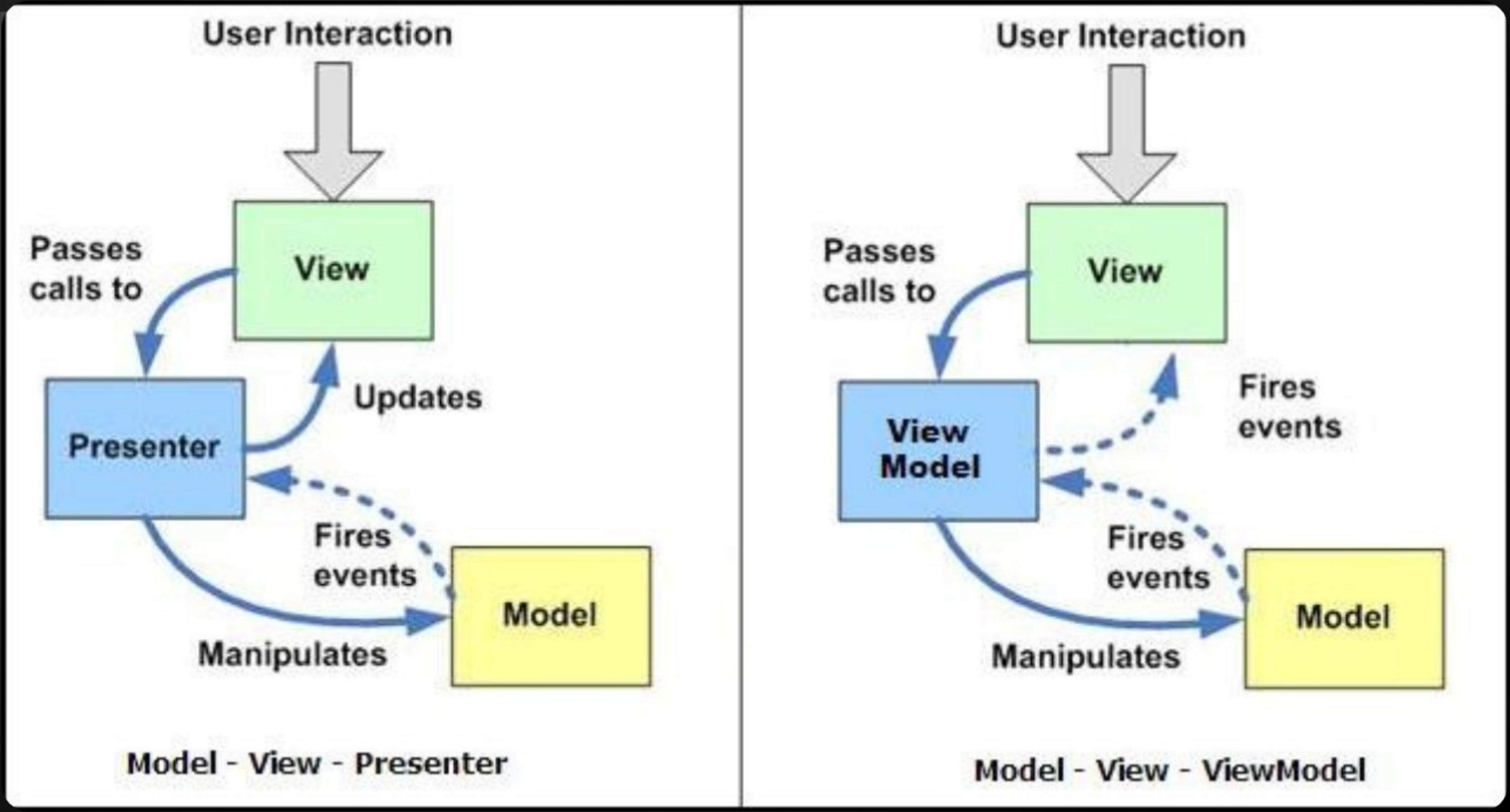


MVVM Architecture

- MVVM is typically used in XAML applications (WPF, Silverlight, WP7) and supports unit testing



MVP vs. MVVM Patterns



- MVVM is like MVP but leverages the platform's build-in bi-directional data binding mechanisms

IoC (Inversion of Control) and DI (Dependency Injection)

Architectural Principals or Design Patterns?

Inversion of Control (IoC)

- Inversion of Control (IoC) is an abstract principle in software design in which
  - The flow of control of a system is inverted compared to procedural programming
  - The main control of the program is inverted, moved away from you to the framework
- Basic IoC principle:

Don't call us, we'll call you!

- Implementations typically rely on callbacks

Procedural Flow Control – Example

```
private void DoSomeTransactionalWork(IDbSesion)
{
  …
}

IDbSession session = new DbSession();
session.BeginTransaction();
try
{
  DoSomeTransactionalWork(session);
  session.CommitTransaction();
}
catch (Exception)
{
  session.RollbackTransaction();
  throw;
}
```

Step by step execution

Inverted Flow Control – Example

```
private static void ExecuteInTransaction(
  Action<IDbSession> doSomeTransactionalWork)
{
  IDbSession session = new DbSession();
  session.BeginTransaction();
  try
  {
    doSomeTransactionalWork(session);
    session.CommitTransaction();
  }
  catch (Exception)
  {
    session.RollbackTransaction();
    throw;
  }
}

ExecuteInTransaction(DoSomeTransactionalWork);
```

Inverted flow control

Dependency Inversion Principle

- Dependency inversion principle
  - Decouples high-level components from low-level components
  - To allow reuse with different low-level component implementations
- Design patterns implementing the dependency inversion principle:
  - Dependency Injection
  - Service Locator

Highly Dependent Components

- Example of highly dependent components:

```
public class LogsDAO
{
  private void AppendToLogs(string message)
  {
    DbSession session = new DbSession();
    session.ExecuteSqlWithParams("INSERT INTO " +
      "Logs(MsgDate, MsgText) VALUES({0},{1})",
      DateTime.Now, message);
  }
}
```

- The `LogsDAO` class is highly-coupled (dependent) to `DbSession` class

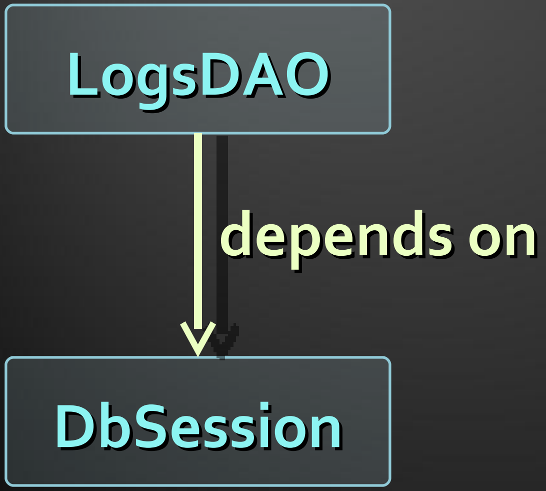
Decoupled Components

```
public class LogsDAO
{
  private IDbSession session;

  public LogsDAO(IDbSession session)
  {
    this.session = session;
  }

  private void AppendToLogs(string message)
  {
    session.ExecuteSqlWithParams("INSERT INTO " +
      "Logs(MsgDate, MsgText) VALUES({0},{1})",
      DateTime.Now, message);
  }
}
```

- The LogsDAO and DbSession are now decoupled

Decoupling Components

- Highly-coupled components:

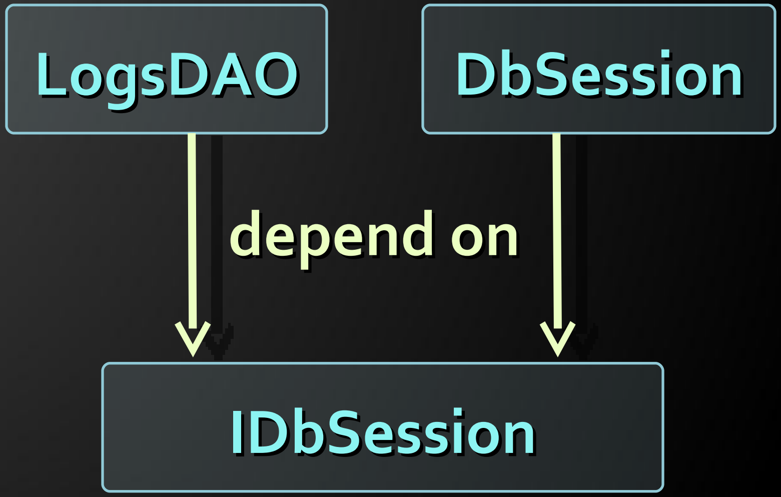


LogsDAO

depends on

DbSession

- Decoupled components:



LogsDAO

DbSession

depend on

IDbSession

Dependency Injection (DI)

- Dependency Injection (DI) is the main method to implement Inversion of Control (IoC) pattern
  - DI and IoC are considered the same concept
  - DI separates behavior from dependency resolution and thus decouples highly dependent components
  - Dependency injection means passing or setting of dependencies into a software component
  - Instead of components having to request dependencies, they are passed (injected) into

Types of Injection

- Dependency Injection (DI) usually runs with IoC Container (also called DI Container)
- Types of dependency injection:
    - Constructor injection – a dependency is passed to the constructor as a parameter
    - Setter injection – a dependency is injected into the dependent object through a property setter
    - Interface injection – an interface is used to inject a dependency into the dependent object
- IoC containers can inject dependencies automatically at run-time

IoC Container – Example

- IoC containers have two main functions
  - Register injectable classes
    - Can be done declaratively (with XML or attributes) or programmatically (in C# code)
  - Resolve already registered classes
    - Done in C# code at runtime
  - Dependency injection could be done automatically with no code
    - E.g. autowire in Spring framework

IoC Container – Example (2)

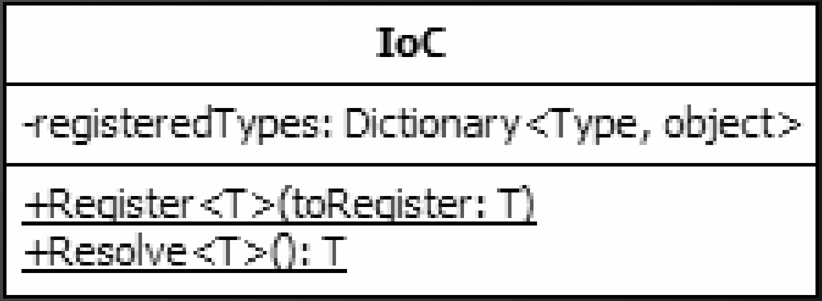
- Consider the following code:

```
public interface ILogger
{
    void LogMessage(string msg);
}

public class ConsoleLogger : ILogger
{
    public void LogMessage(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

- We want to use IoC container to resolve the dependency between our code and the logger

IoC Container – Example (3)

- Consider the IoC container provides the following methods:



IoC

-registeredTypes: Dictionary<Type, object>

+Register<T>(toRegister: T)
+Resolve<T>(): T

- Registering the logger:

```
IoC.Register<ILogger>(new ConsoleLogger());
```

- Using the registered logger:

```
ILogger logger = IoC.Resolve<ILogger>();
logger.LogMessage("Hello, world!");
```

IoC Containers for .NET

- Microsoft ObjectBuilder; Microsoft Unity
  - Open-source projects at CodePlex
  - Part of Patterns & Practices Enterprise Library
- Spring.NET – www.springframework.net
  - .NET port of the famous Spring framework from the Java world (currently owned by VMware)
- Castle Windsor –www.castleproject.org
  - Open-source IoC container, part of the Castle project

Microsoft Prism

- Patterns and Practices: Prism
  - Patterns For Building Composite Applications With WPF and Silverlight
  - Composite applications – consists of loosely coupled modules discoverable at runtime
- Prism components
  - Prism Library
  - Stock Trader Reference Implementation
  - MVVM Reference Implementation
  - QuickStarts

Managed Extensibility Framework (MEF)

- Managed Extensibility Framework (MEF)
  - Simplifies the design of extensible applications and components
  - Official part of .NET Framework 4
  - Allows developers to discover and use extensions with no configuration at runtime
  - lets extension developers easily encapsulate code and avoid fragile hard dependencies

SOA (Service-Oriented Architecture)

SOA and Cloud Computing

What is SOA?

- Service-Oriented Architecture (SOA) is a concept for development of software systems
  - Using reusable building blocks (components) called "services"
- Services in SOA are:
  - Autonomous, stateless business functions
  - Accept requests and return responses
  - Use well-defined, standard interface

SOA Services

- Autonomous
  - Each service operates autonomously
  - Without any awareness that other services exist
- Statelessa
  - Have no memory, do not remember state
  - Easy to scale
- Request-response model
  - Client asks, server returns answer

SOA Services (2)

- Communication through standard protocols
  - XML, SOAP, JSON, RSS, ATOM, ...
  - HTTP, FTP, SMTP, RPC, ...
- Not dependent on OS, platforms, programming languages
- Discoverable
  - Service registries
  - Could be hosted "in the cloud" (e.g. in Azure)

What is Cloud Computing?

- Cloud computing is a modern approach in the IT infrastructure that provides:
  - Software applications, services, hardware and system resources
  - Hosts the applications and user data in remote servers called "the cloud"
- Cloud computing models:
  - IaaS – infrastructure as a service (virtual servers)
  - PaaS – platform as a service (full stack of technologies for UI , application logic, data storage)
  - SaaS – software as a service (e.g. Google Docs)

Loose Coupling

- Loose coupling is the main concept of SOA
- Loosely coupled components:
  - Exhibits single function
  - Independent of other functions
  - Through a well-defined interface
- Loose coupling programming evolves:
  - Structural programming
  - Object-oriented programming
  - Service-oriented architecture (SOA)

SOA Design Patterns

- SOA Patterns – www.soapatterns.org
  - Inventory Foundation, Logical Layer, Implementation, Governance Patterns
  - Service Foundational, Implementation, Security, Contract, Governance, Messaging Patterns
  - Legacy Encapsulation Patterns
  - Capability Composition Patterns
  - Composition Implementation Patterns
  - Transformation Patterns
  - Common Compound Design Patterns

Questions?