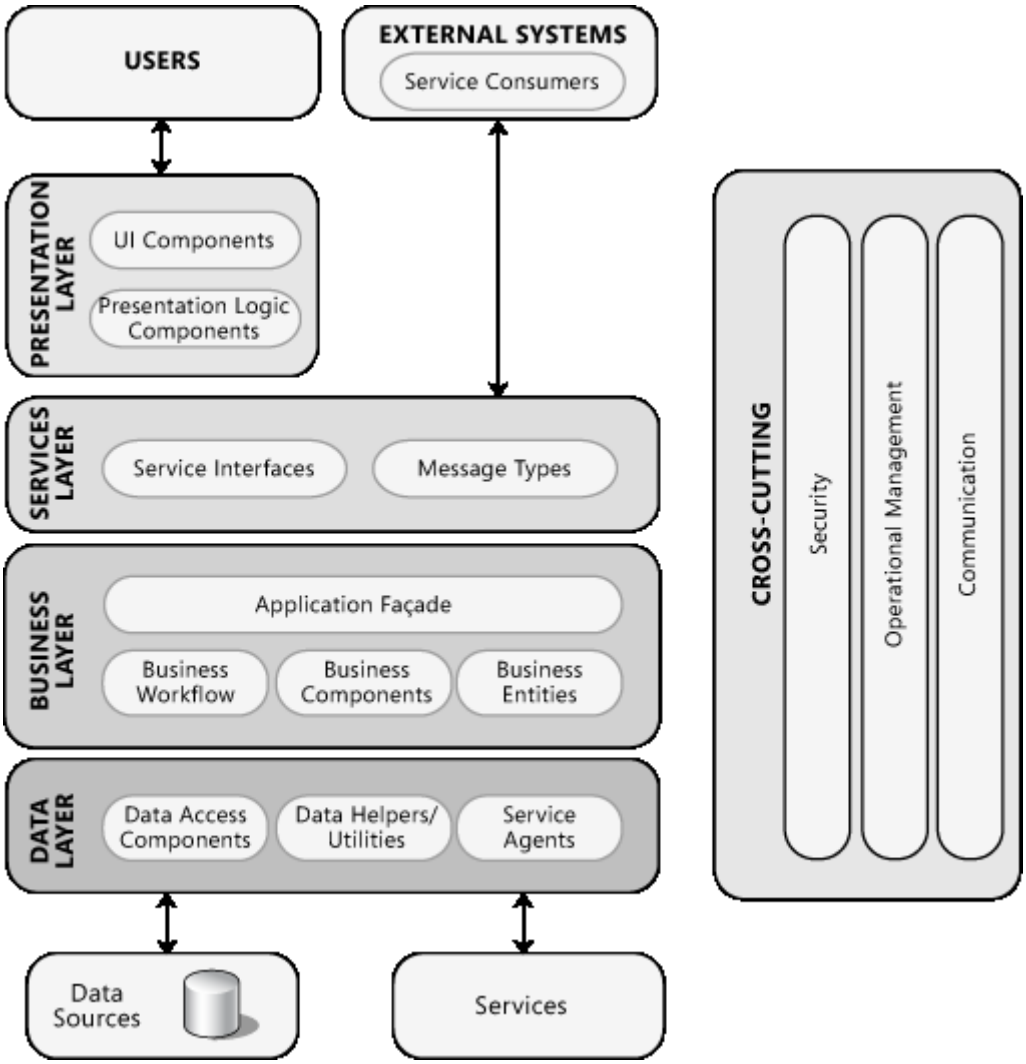
Projektowanie obiektowe oprogramowania
Wzorce architektury aplikacji (4)
Wykład 12 – MVC/MVP
Wiktor Zychla 2014

1 Architektura aplikacji

1.1 Diagram referencyjny architektury aplikacji

O przekroju architektury aplikacji, od dołu (np. od warstwy danych), do samej góry (np. do warstwy interfejsu użytkownika) mówimy często **stos aplikacyjny**, często mając na myśli konkretny zestaw technologii połączonych w taki sposób żeby zapewniać możliwość implementacji poszczególnych warstw.



1.2 Rodzaje aplikacji

Application type	Description
Mobile Application	• Can be developed as a Web application or a rich client application. • Can support occasionally connected scenarios. • Runs on devices with limited hardware resources.
Rich Client Application	• Usually developed as a stand-alone application. • Can support disconnected or occasionally connected scenarios. • Uses the processing and storage resources of the local machine.
Rich Internet Application	• Can support multiple platforms and browsers. • Can be deployed over the Internet. • Designed for rich media and graphical content. • Runs in the browser sandbox for maximum security. • Can use the processing and storage resources of the local machine.
Service Application	• Designed to support loose coupling between distributed components. • Service operations are called using XML-based messages. • Can be accessed from the local machine or remotely, depending on the transport protocol.
Web Application	• Can support multiple platforms and browsers. • Supports only connected scenarios. • Uses the processing and storage resources of the server.

1.3 Typy architektury aplikacji

Architecture style	Description
Client-Server	Segregates the system into two computer programs where one program, the client, makes a service request to another program, the server.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that are location-transparent and expose well-defined communication interfaces.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message-Bus	A software system that can receive and send messages that are based on a set of known formats, so that systems can communicate with each other without needing to know the actual recipient.
Model-View-Controller (MVC)	Separates the logic for managing user interaction from the UI view and from the data with which the user works.
N-tier / 3-tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.

1.4 Kryteria ewaluacji architektury aplikacji

Category	Description
Availability	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
Conceptual Integrity	Conceptual integrity defines the consistency and coherence of

	the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.
Flexibility	Flexibility is the ability of a system to adapt to varying environments and situations, and to cope with changes to business policies and rules. A flexible system is one that is easy to reconfigure or adapt in response to different user and system requirements.
Interoperability	Interoperability is the ability of diverse components of a system or different systems to operate successfully by exchanging information, often by using services. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
Maintainability	Maintainability is the ability of a system to undergo changes to its components, services, features, and interfaces as may be required when adding or changing the functionality, fixing errors, and meeting new business requirements.
Manageability	Manageability defines how easy it is to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
Performance	Performance is an indication of the responsiveness of a system to execute any action within a given interval of time. It can be measured in terms of latency or throughput. *Latency* is the time taken to respond to any event. *Throughput* is the number of events that take place within given amount of time.
Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified interval of time.
Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Scalability	Scalability is the ability of a system to function well when there are changes to the load or demand. Typically, the system will be able to be extended by scaling up the performance of the server, or by scaling out to multiple servers as demand and load increase.
Security	Security defines the ways that a system is protected from disclosure or loss of information, and the possibility of a successful malicious attack. A secure system aims to protect assets and prevent unauthorized modification of information.
Supportability	Supportability defines how easy it is for operators, developers, and users to understand and use the application, and how easy it is to resolve errors when the system fails to work correctly.
Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
Usability	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, able to provide good access for disabled

	users, and able to provide a good overall user experience.

1.5 Kluczowe decyzje projektowe

Category	Key problems
Authentication and Authorization	• How to store user identities • How to authenticate callers • How to authorize callers • How to flow identity across layers and tiers
Caching and State	• How to choose effective caching strategies • How to improve performance by using caching • How to improve availability by using caching • How to keep cached data up to date • How to determine the data to cache • How to determine where to cache the data • How to determine an expiration policy and scavenging mechanism • How to load the cache data • How to synchronize caches across a Web or application farm
Communication	• How to communicate between layers and tiers • How to perform asynchronous communication • How to communicate sensitive data
Composition	• How to design for composition • How to design loose coupling between modules • How to handle dependencies in a loosely coupled way
Concurrency and Transactions	• How to handle concurrency between threads • How to choose between optimistic and pessimistic concurrency • How to handle distributed transactions • How to handle long-running transactions • How to determine appropriate transaction isolation levels • How to determine whether compensating transactions are required
Configuration Management	• How to determine the information that must be configurable • How to determine location and techniques for storing configuration information • How to handle sensitive configuration information • How to handle configuration information in a farm or cluster
Coupling and Cohesion	• How to separate concerns • How to structure the application • How to choose an appropriate layering strategy • How to establish boundaries
Data Access	• How to manage database connections • How to handle exceptions • How to improve performance • How to improve manageability • How to handle binary large objects (BLOBs) • How to page records • How to perform transactions
Exception Management	• How to handle exceptions • How to log exceptions
Logging and Instrumentation	• How to determine the information to log

	• How to make logging configurable
User Experience	• How to improve task efficiency and effectiveness • How to improve responsiveness • How to improve user empowerment • How to improve the look and feel
Validation	• How to determine location and techniques for validation • How to validate for length, range, format, and type • How to constrain and reject input • How to sanitize output
Workflow	• How to handle concurrency issues within a workflow • How to handle task failure within a workflow • How to orchestrate processes within a workflow

2 Wzorce architektury warstwy interfejsu użytkownika

Wzorce warstwy interfejsu użytkownika mają na celu zapewnienie możliwości łatwiejszego utrzymania kodu oraz podniesienie wiarygodności – osiągają to **oddzielając** logikę przetwarzania od logiki prezentacji.

Dzięki lepszej izolacji, możliwe jest **testowanie** obu warstw niezależnie za pomocą testów zautomatyzowanych, **nie wymagających interakcji użytkownika**.

Mówiąc kolokwialnie: chodzi o tak zbudowaną warstwę widoków, żeby "klikać po nich" (= prowadzić testy) mógł automat bez konieczności posiadania rzeczywistego interfejsu użytkownika. Normalne aplikacje wymagające interfejsu użytkownika są trudno testowalne w scenariuszach, w których testujący automat działa w trybie usługi (system service), który to tryb ze względu na swoją charakterystykę nie pozwala łatwo automatyzować interfejsu użytkownika.

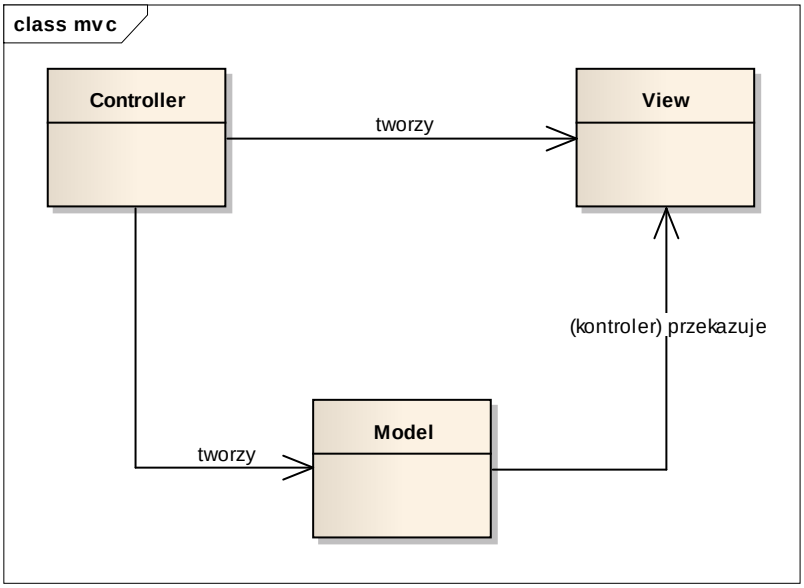
Omówimy trzy wzorce:

- Model-View-Controller
- Model-View-Presenter
- Model-View-ViewModel

2.1 Model-View-Controller (MVC)

Wzorzec architektury interfejsu użytkownika zarezerwowany dla aplikacji typu **Web Application.**

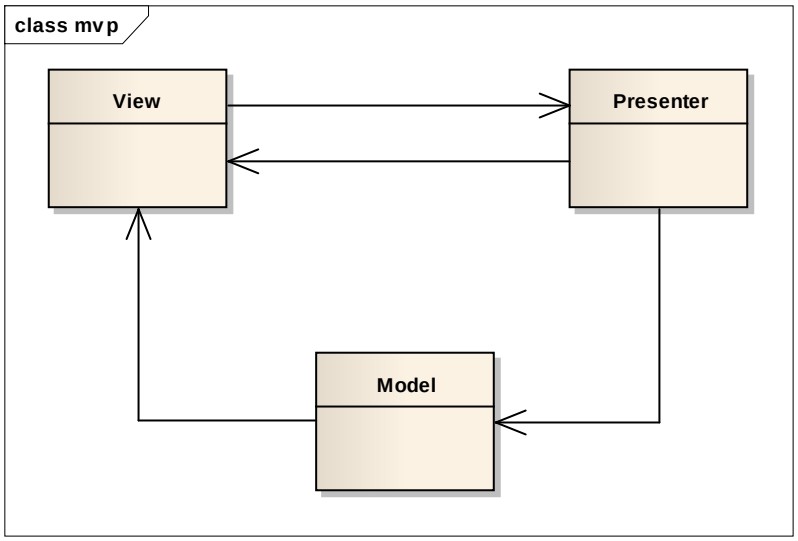
- Interakcja użytkownika Controller → Model + View
- Kontroler i widoki są połączone relacją 1-wiele (1 kontroler obsługuje wiele widoków)
- Kontroler obsługuje logikę akcji użytkownika; kontroler jest sterowany przez środowisko uruchomieniowe (tu: serwer aplikacji)
- Kontroler na podstawie akcji użytkownika wybiera widok do wyrenderowania i do widoku przekazuje model



2.2 Model-View-Presenter (MVP)

Wzorzec architektury interfejsu użytkownika zarezerwowany dla aplikacji typu **Rich Client Application**.

- Interakcja View ←→ Presenter → model
- Widok i prezenter są połączone 1-1 (jeden prezenter ma jeden widok)
- Logikę obsługuje prezenter, to on rejestruje się na powiadomienia, tworzy model
- Widok ma tylko warstwę prezentacji (sterowaną przez prezenter)
- Widok jest wstrzykiwany do prezentera przez interfejs – głównym celem takiego podejścia jest zapewnienie możliwości wstrzykiwania do prezentera innych implementacji widoków (takich które nadają się do testowania bo nie wymagają interakcji użytkownika)



Wskazówki do refaktoryzacji w kierunku MVP:

- Widoki (formularze) nie powinien nigdy bezpośrednio dostawać się do modelu (danych), model powinien być im przekazywany przez prezentera (jakoś, np. przez wywołanie metody na widoku która przyjmuje model jako parametr)

- Widoki nie mają żadnej logiki, cała logika jest obsługiwana przez prezentera. Prawdziwe widoki w zdarzeniach formularzy wywołują odpowiednie metody prezentera.
- To prezentery rejestrują się jako odbiorcy powiadomień w EventAggregatorze, to również prezentery podnoszą powiadomienia dla EventAggregatora
- Kod całej aplikacji używa Dependency Resolvera/Service Locatora lub innego mechanizmu wsparcia IoC, żeby zapewnić rozwiązywanie zależności do właściwych implementacji widoków dla prezenterów
- Po zarejestrowaniu widoków testowych, powinno dać się pisać imperatywne scenariusze testów jednostkowych (skrypt testu) za pomocą odwołań do prezenterów. To oznacza, że jeżeli okno A tworzy okno B – to metoda jego tworzenia jest metodą prezentera A i zwraca prezenter B.

2.3 Model-View-ViewModel (MVVM)

Wariacja na temat MVP – rozwinięcie idei. Wprowadzony szerzej w kontekście technologii WPF/XAML na platformie .NET.

- W MVP widok może mieć normalny kod imperatywny obsługujący dane przekazywane z prezentera
- W MVVM widok (idealnie) nie powinien mieć żadnej logiki, jedyny dozwolony mechanizm odwołań do danych z prezentera (view modelu) to deklaratywny *data-binding* (czyli wiązanie danych, opisane statycznie w strukturze widoku)
- Ponieważ widok bezpośrednio odwzorowuje dane wystawiane z prezentera (view modelu), prezentera nie nazywa się prezenterem tylko właśnie view-modelem
- w MVP widok ma prawo w dowolny sposób otrzymać dane od prezentera, na przykład w taki sposób że prezenter wywołuje jakąś metodę na interfejsie opisującym widok. W MVVM jest trochę odwrotnie – to ViewModel „wystawia" składowe modelu, do których widok może podwiązać (*data-binding*) komponenty interfejsu użytkownika

Więcej: http://msdn.microsoft.com/en-us/library/ff798384.aspx

3 Przykład na żywo

Podczas wykładu zbudujemy prostą aplikację – rejestr użytkowników. Aplikacja będzie posiadać dwa rodzaje okien:

- okno główne z listą użytkowników,
- okno dodawania/edycji użytkownika.

Wyposażeni w wiedzę z poprzedniego wykładu, najpierw zrefaktoryzujemy aplikację do wzorca Repository, wprowadzając abstrakcję na sposób obsługi danych.

Następnie wprowadzimy EventAggregator do zbudowania architektury komunikacji wewnątrzaplikacyjnej (np. do komunikacji między oknami).

W kolejnym kroku zrefaktoryzujemy widoki rozdzielając warstwę prezentacji i warstwę obsługi logiki do warstw odpowiednio V i P (Views/Presenters). Pokażemy jak widoki implementują interfejsy, a prezentery odwołują się do widoków przez ich abstrakcje.

To pozwoli nam na wprowadzenie widoków zastępczych oraz testy jednostkowe prezenterów na widokach zastępczych. Z zachowaniem wszystkich wcześniej wprowadzonych elementów.

4 Literatura

Microsoft Patterns & Practices – Application Architecture Guide