

Paweł Rajba

pawel@cs.uni.wroc.pl

<http://pawel.ii.uni.wroc.pl/>

Architektura aplikacji

Architektura

Bass, Clements i Kazman, 2003:

„Architektura oprogramowania programu lub systemu informatycznego jest strukturą lub zbiorem struktur tego systemu, obejmującym elementy oprogramowania, widoczne dla pozostałych elementów oprogramowania oraz zależności między nimi.”

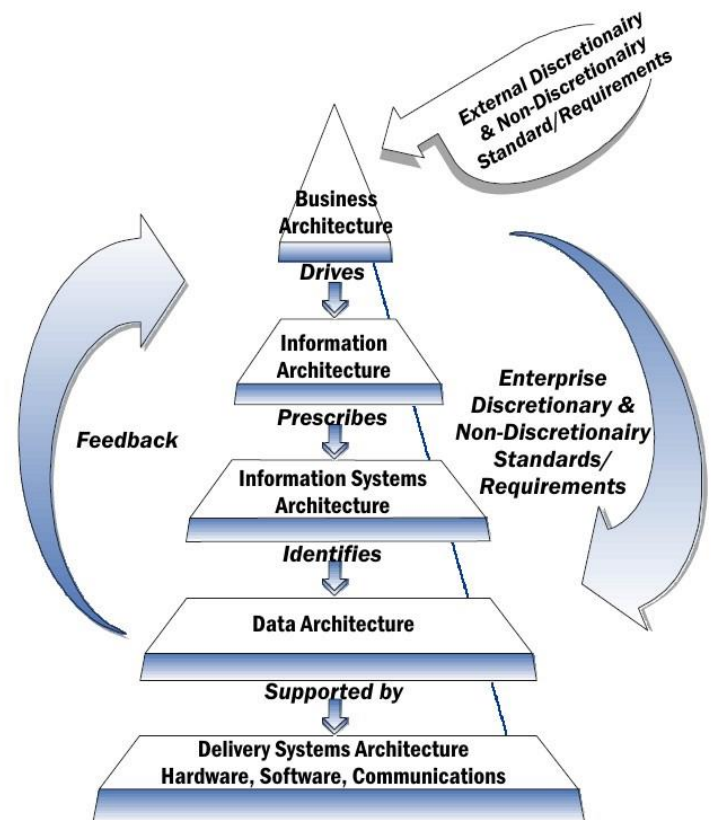
ISO/IEC 42010:2007:

„Architektura systemu informatycznego jest to podstawowa organizacja systemu wraz z jego komponentami, wzajemnymi powiązaniem, środowiskiem pracy i regułami ustanawiającymi sposób jej budowy i rozwoju”

Architektura

- O architekturze systemu można mówić na różnym poziomie szczegółowości i różnych punktów widzenia

- Enterprise
- Information
- Integration
- Infrastructure
- Software
- Security
- ...



Architektura

- Interakcja z danymi jest jednym z aspektów
 - Dane przepływają przez wszystkie warstwy systemu
 - Od bazy danych do interfejsu użytkownika
 - (lub odwrotnie)
 - Pomędzy systemami
 - Przepływ jest automatyczny lub manualny
 - Dane wpisywane do Excela i wysyłane e-mailem

Architektura

- Dlaczego?
 - Łatwiej zrozumieć całość rozwiązania
 - W tym ułatwia zrozumienie, jak potencjalne zmiany wpłyną na rozwiązanie
 - Ułatwia tworzenie komponentów wielokrotnego zastosowania
 - Redukuje koszt utrzymania i rozwoju systemu
 - Pozwala kontrolować wymagania niefunkcjonalne
 - Jest niezbędna, aby zapewnić wysoką jakość oprogramowania

Architektura

■ Dlaczego?

- Accessibility
- Adaptability
- Auditability and control
- Availability (see service level agreement)
- Backup
- Capacity, current and forecast
- Certification
- Compliance
- Configuration management
- Cost, initial and Life-cycle cost
- Data integrity
- Data retention
- Dependency on other parties
- Deployment
- Development environment
- Disaster recovery
- Documentation
- Durability
- Efficiency (resource consumption for given load)
- Effectiveness (resulting performance in relation to effort)
- Emotional factors (like fun or absorbing or has "Wow! Factor")
- Environmental protection
- Escrow
- Exploitability
- Extensibility (adding features, and carry-forward of customizations at next major version upgrade)
- Failure management
- Fault tolerance (e.g. Operational System Monitoring, Measuring, and Management)
- Legal and licensing issues or patent-infringement-avoidability
- Interoperability

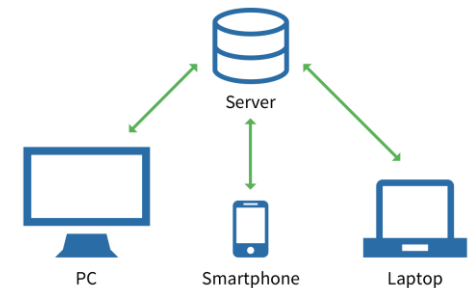
- Maintainability (e.g. Mean Time To Repair - MTTR)
- Management
- Modifiability
- Network topology
- Open source
- Operability
- Performance / response time (performance engineering)
- Platform compatibility
- Privacy (compliance to privacy laws)
- Portability
- Quality (e.g. faults discovered, faults delivered, fault removal efficacy)
- Readability
- Reliability (e.g. Mean Time Between/To Failures - MTBF/MTTF)
- Reporting
- Resilience
- Resource constraints (processor speed, memory, disk space, network bandwidth, etc.)
- Response time
- Reusability
- Robustness
- Safety or Factor of safety
- Scalability (horizontal, vertical)
- Security (cyber and physical)
- Software, tools, standards etc. Compatibility
- Stability
- Supportability
- Testability
- Throughput
- Transparency
- Usability (Human Factors) by target user community
- Integrability ability to integrate components

Architectural styles

- Bardzo ogólne podejście do architektury
- Opisują architekturę z różnych punktów widzenia
- Podział na kategorie
 - Communication
 - Service-Oriented Architecture (SOA), Message Bus
 - Deployment
 - Client/Server, N-Tier, 3-Tier
 - Domain
 - Domain Driven Design
 - Structure
 - Component-Based, Object-Oriented, Layered Architecture

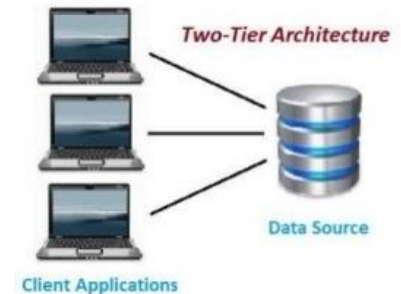
Architektura klient-serwer

- Dwie strony komunikacji
 - Klient – strona żądająca dostępu do danych, usługi
 - aktywny: wysyła żądanie do serwera i oczekuje odpowiedzi
 - Serwer – udostępnia dane i usługi
 - pasywny: czeka na żądania od klientów
 - w momencie otrzymania żądania, przetwarza je, a następnie wysyła odpowiedź
- Rodzaje architektury klient serwer
 - dwuwarstwowa, trójwarstwowa, wielowarstwowa



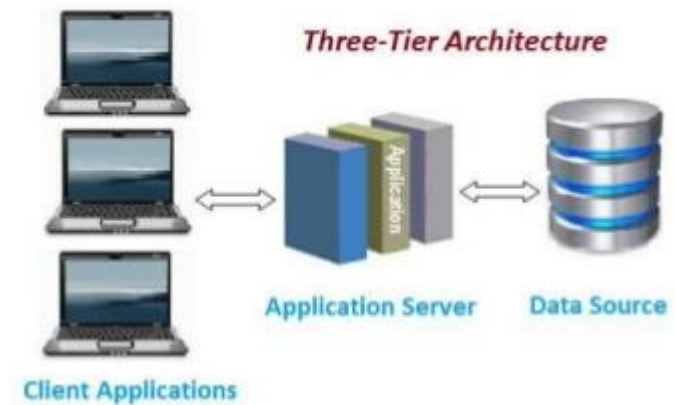
Architektura dwuwarstwowa

- Przetwarzanie i składowanie danych odbywa się w jednym module
- Dawniej bardzo popularny, obecnie wypierany przez model trójwarstwowy
- W tym modelu jest zapewniony wielodostęp
- Tańsze niż w modelu wielowarstwowym, ale
 - Problem ze skalowalnością
 - Gdzie logika biznesowa?
 - Trudniej zapewnić różne sposoby prezentacji
 - np. web client, mobile client



Architektura trójwarstwowa

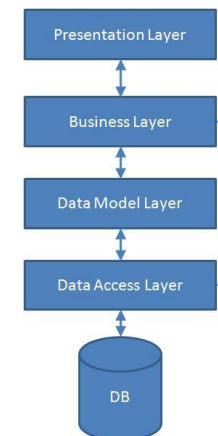
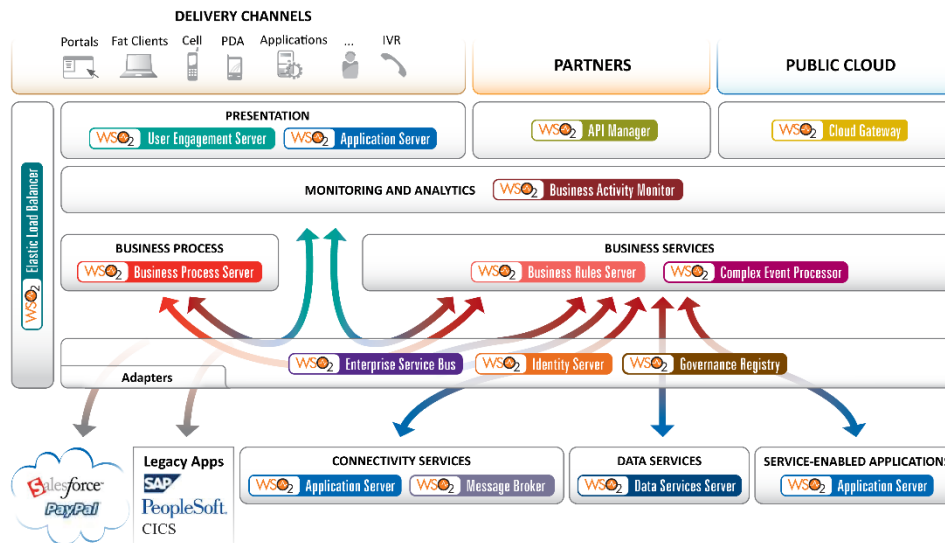
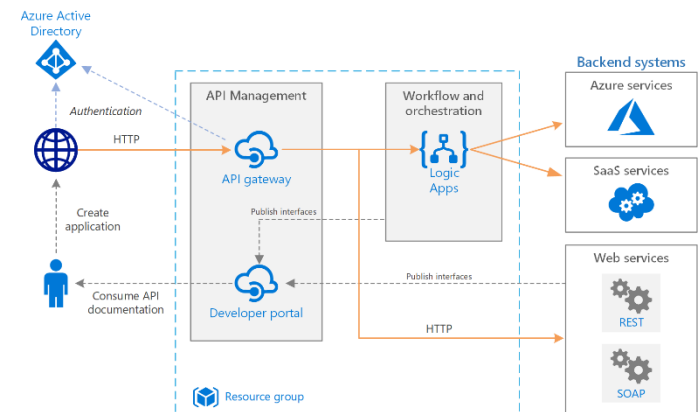
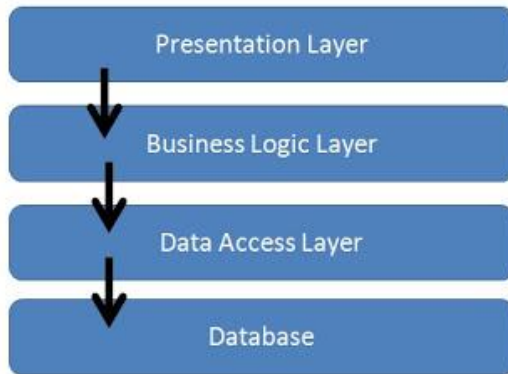
- Przetwarzanie i składowanie danych następuje w dwóch osobnych modułach
- Występują następujące warstwy
 - Prezentacji
 - Logiki biznesowej
 - Danych
- Zalety:
 - Skalowalne
 - Lepsze bezpieczeństwo
 - Mniej powiązane
 - Łatwiej zapewnić różne formy prezentacji



Architektura wielowarstwowa

- Przetwarzanie, składowanie i inne operacje na danych odbywają się w wielu osobnych modułach
- Zwykle tworzy się hierarchie warstw, gdzie warstwa n komunikuje się z warstwami $n-1$ i $n+1$
 - Chociaż ostatnio coraz rzadziej jest to hierarchia
- Ważne cechy podejścia warstwowego
 - Abstrakcja, enkapsulacja, podział funkcjonalności pomiędzy warstwy, reużywalność, luźne powiązania
- Główne zalety takiego podejścia
 - Abstrakcja, izolacja, zarządzalność, wydajność, reużywalność, testowalność

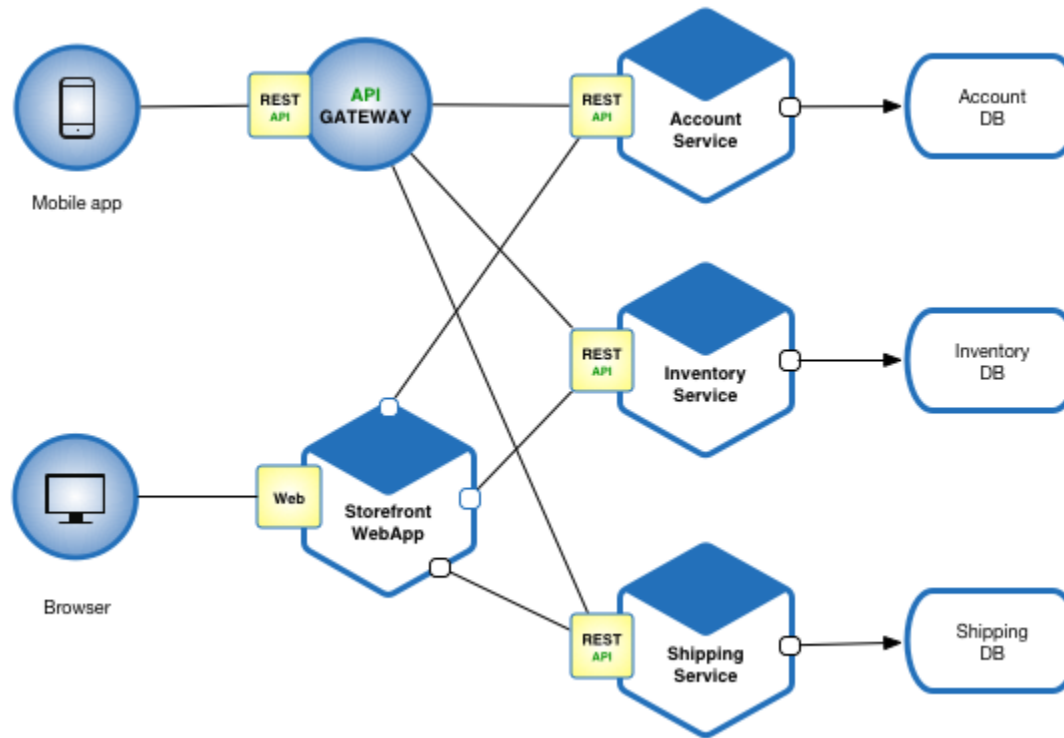
Architektura wielowarstwowa



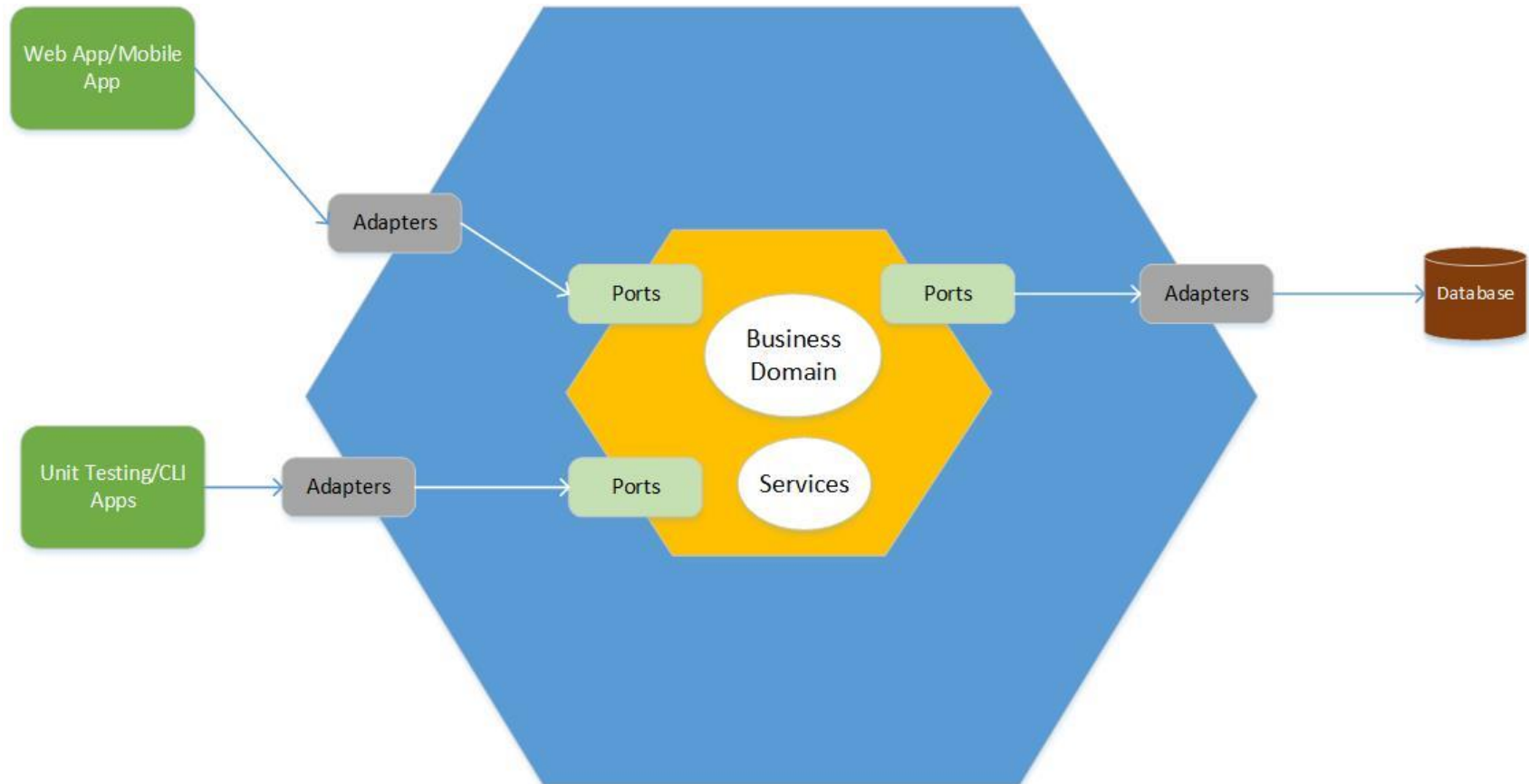
Architektura wielowarstwowa

- Przykładowy podział na warstwy
 - Warstwa interfejsu użytkownika
 - np. web client, mobile client
 - Warstwa zarządzania treścią, tzw. backend
 - np. CMS
 - Warstwa usług: web services & data services
 - np. dostarczanie danych innym modułom
 - Warstwa uwierzytelnienia i autoryzacji
 - np. delegacja na zupełnie zewnętrzny serwer
 - Warstwa pamięci masowej, czyli baza danych

Microservices



Hexagonal architecture



Przetwarzanie danych

- W architekturze wielowarstwowej przetwarzanie może być
 - Na serwerze bazy danych (funkcje i procedury)
 - Na serwerze aplikacji
 - Na kliencie
 - Rola hurtowni danych
- Zalety i wady poszczególnych rozwiązań

Master Data Management

Governance

Consolidation

Standardization

Cleansing

Policies

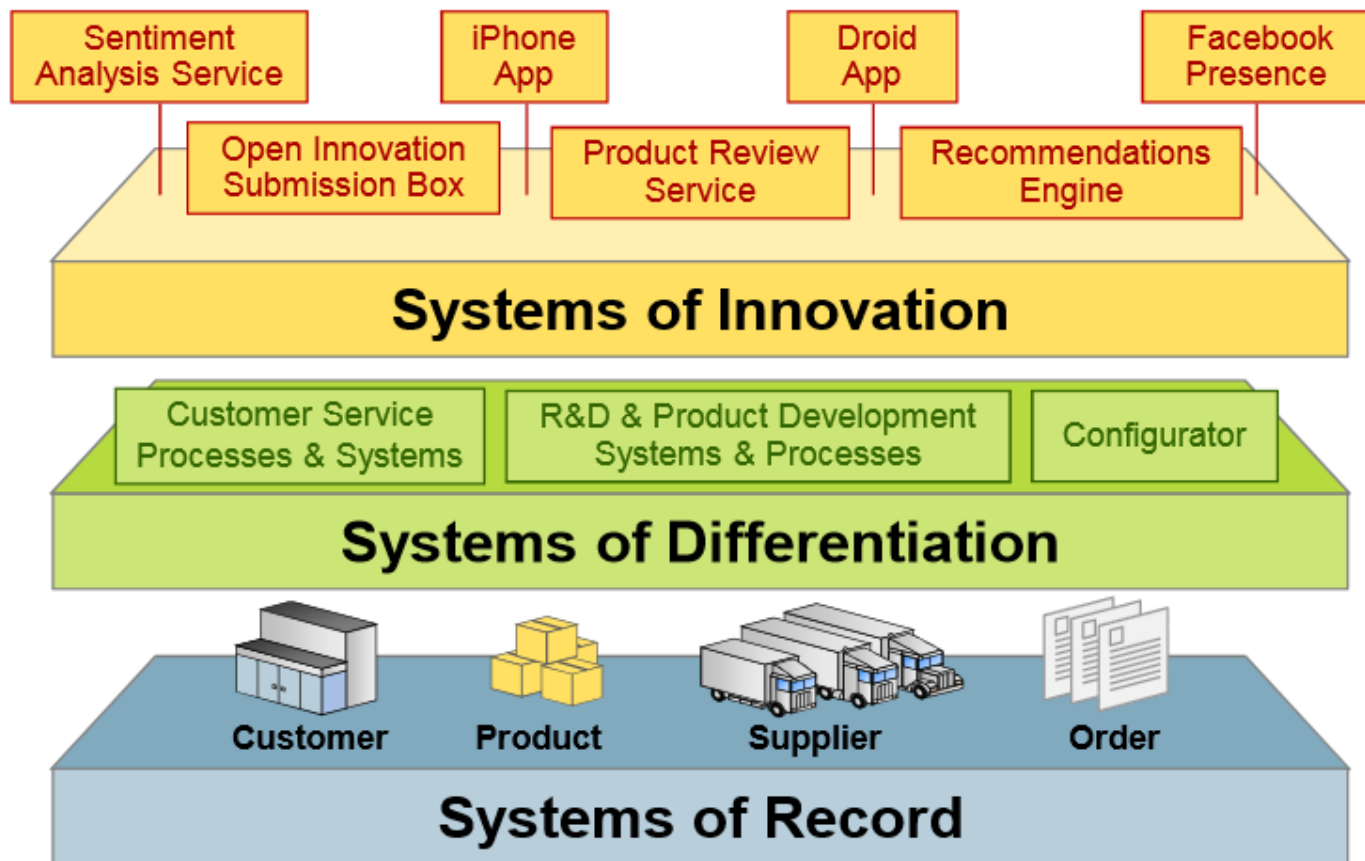
Distribution

Processes

Tools

Pace layered architecture

Facebook, Twitter, YouTube, Communities, Portal, etc.



MVP vs. MMP vs. MLP

■ Minimum Viable/Marketable/Loveable Product

MVP	MMP	MLP
Fastest to develop	Fast to develop	Fast to develop (but can be slower than MVP)
Minimum features to test the idea	Minimum features to sell the product	Minimum features to entice an emotional response
Built on the idea	Built on several MVPs	Built on the MVP concept
Cheapest	Relatively cheap	Price depends on the features selected
Non-marketable (designed to test the idea)	Marketable	Marketable and remarkable
Business-oriented	Business-oriented	User-oriented
Addresses the need	Solves the problem	Solves the problem and creates an emotional connection

Wzorce dostępu do danych

- Sporo wzorców jest w książce guru Martina Fowlera Patterns of Enterprise Application Architecture, część dotyczy danych
- Katalog wzorców zebrany na stronie <http://martinfowler.com/eaCatalog/>
- Przyjrzymy się chwilę dwóm kategoriom
 - Domain Logic Patterns
 - Data Source Architectural Patterns
 - Pozostałe kategorie dot. będą omawiane przy okazji tematów, których dotyczą, np. przy omawianiu ORM

Wzorce dostępu do danych

- Domain Logic Patterns
 - Transaction Script
 - *Organizes business logic by procedures where each procedure handles a single request from the presentation.*
 - Domain Model
 - *An object model of the domain that incorporates both behavior and data.*
 - Table Module
 - *A single instance that handles the business logic for all rows in a database table or view.*
 - Service Layer
 - *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.*

Wzorce dostępu do danych

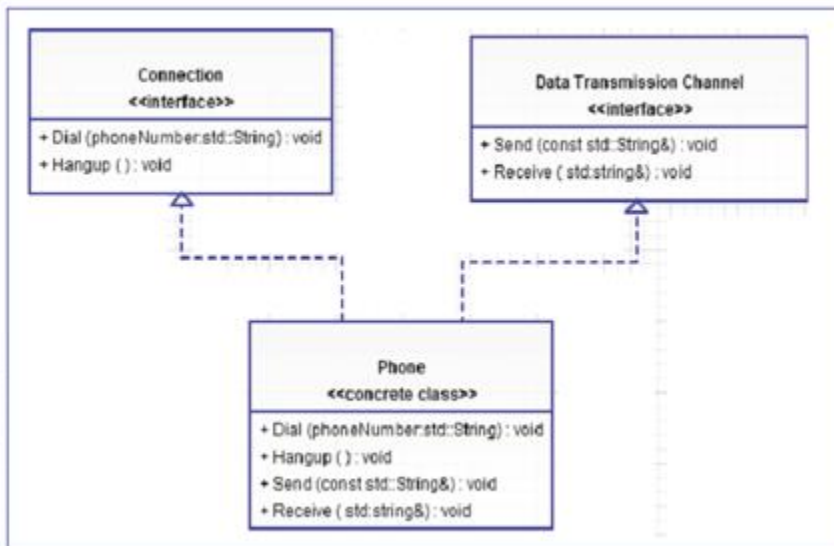
- Data Source Architectural Patterns
 - Table Data Gateway
 - *An object that acts as a Gateway (466) to a database table. One instance handles all the rows in the table.*
 - Row Data Gateway
 - *An object that acts as a Gateway (466) to a single record in a data source. There is one instance per row.*
 - Active Record
 - *An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.*
 - Data Mapper
 - *A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.*

Design principles: SOLID

■ S: Single Responsibility Principle

SRP says

"Every software module should have only one reason to change".



```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }

    public void Insert(Employee e)
    {
        //Database Logic written here
    }

    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }
}
```

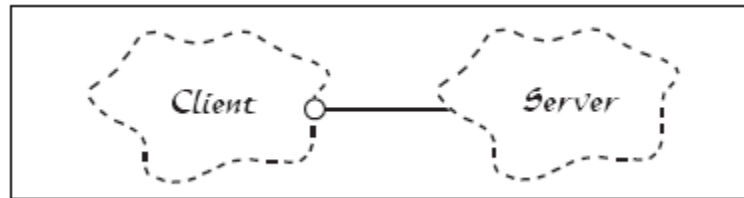
Responsibility 1

Responsibility 2

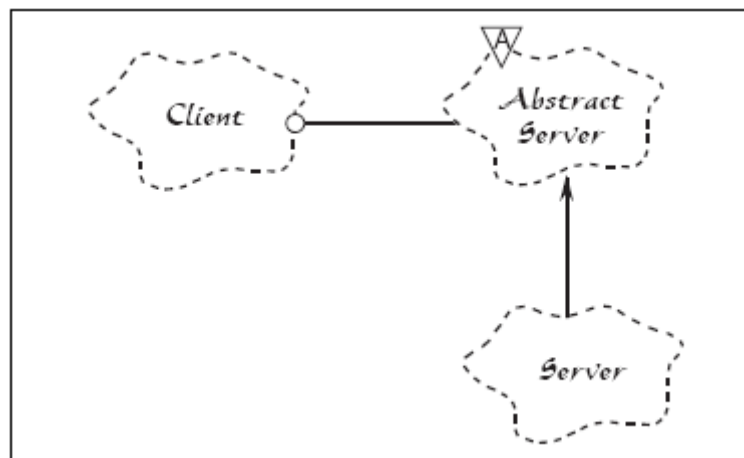
Design principles: SOLID

- O: Open/Closed Principle

- Tu klienta trzeba zmienić



- Tutaj można podmienić serwer bez zmiany klienta



Design principles: SOLID

■ L: Liskov substitution principle

```
public class Rectangle
{
    protected int _width;
    protected int _height;
    public int Width
    {
        get { return _width; }
    }
    public int Height
    {
        get { return _height; }
    }

    public virtual void SetWidth(int width)
    {
        _width = width;
    }
    public virtual void SetHeight(int height)
    {
        _height = height;
    }
    public int getArea()
    {
        return _width * _height;
    }
}
```

```
public class Square : Rectangle {
    public override void SetWidth(int width)
    {
        _width = width;
        _height = width;
    }

    public override void SetHeight(int height)
    {
        _height = height;
        _width = height;
    }
}
```

```
Rectangle objRectangle = new Rectangle();
objRectangle.SetWidth(2);
objRectangle.SetHeight(3);
Console.WriteLine("Area of rectangle : " +
    objRectangle.getArea());

Output :
Area of rectangle : 6
```

```
Rectangle square = new Square();
square.SetWidth(2);
square.SetHeight(3);
Console.WriteLine("Area of square : " +
    square.getArea());

Output :
Area of square is : 9
```

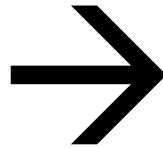
```
Rectangle square = new Square();
square.SetHeight(3);
square.SetWidth(2);
Console.WriteLine("Area of square : " +
    square.getArea());

Output :
Area of square is : 4
```

Design principles: SOLID

■ I: Interface Segregation Principle

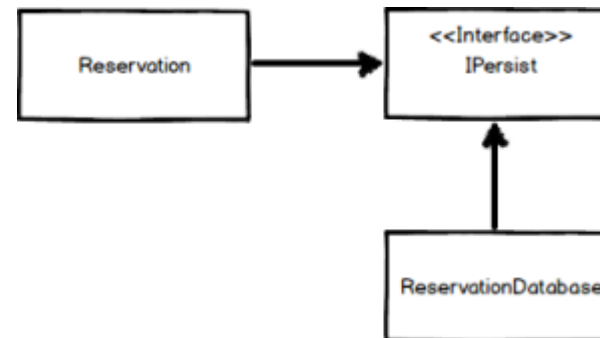
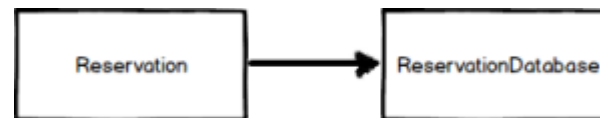
```
interface IWorker
{
    void Work();
    void Eat();
}
class Worker: IWorker
{
    public void Work()
    {
    }
    public void Eat()
    {
    }
}
class Robot: IWorker
{
    public void Work()
    {
    }
    public void Eat()
    {
        throw new NotImplementedException();
    }
}
```



```
interface IWorkable
{
    public void Work();
}
interface IFeedable
{
    public void Eat();
}
class Worker: IWorkable, IFeedable
{
    public void Work()
    {
    }
    public void Eat()
    {
    }
}
class Robot: IWorkable
{
    public void Work()
    {
    }
}
```

Design principles: SOLID

- D: Dependency Inversion Principle



- Other concepts:
 - Dependency Injection
 - Inversion of Control

Design principles: SOLID



Single Responsibility Principle

A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)



Open / Closed Principle

A software module (it can be a class or method) should be open for extension but closed for modification.



Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.



Interface Segregation Principle

Clients should not be forced to depend upon the interfaces that they do not use.



Dependency Inversion Principle

Program to an interface, not to an implementation.

Design principles summary

- KISS

- http://en.wikipedia.org/wiki/KISS_principle

- DRY

- http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

- SoC

- http://en.wikipedia.org/wiki/Separation_of_concerns
 - TCP/IP stack as an example

- YAGNI

- http://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

Design Principles vs Design Patterns

Principles

- More abstract
- Rather guide
- Examples
 - Program to interfaces, not to implementations.
 - Depend upon abstractions. Do not depend upon concrete classes.

Patterns

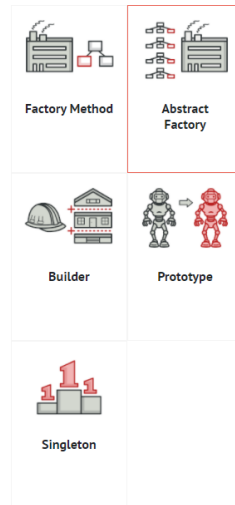
- More specific
- Reusable solution for common problem
- Examples
 - Singleton Pattern
 - One class can only have one instance at a time
 - Adapter Pattern
 - Match interface of different classes

Design Patterns

The Catalog of Design Patterns

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



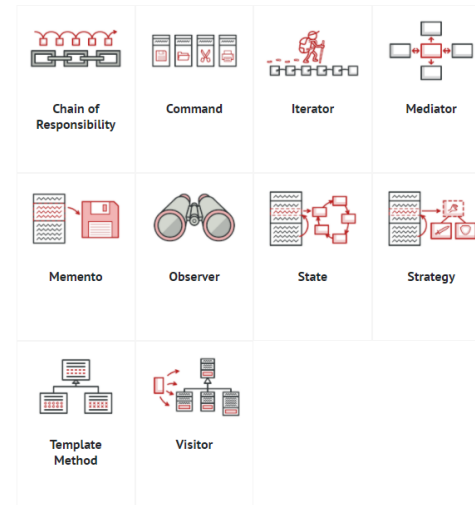
Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.



Literatura

- Software Architecture and Design

<http://msdn.microsoft.com/en-us/library/ee658093.aspx>

- SOLID

- <http://www.oodesign.com/design-principles.html>

- <http://joelabrahamsson.com/a-simple-example-of-the-open-closed-principle/>

- <http://www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles>