

# Nowoczesne architektury aplikacji

Wprowadzenie



# Poznajmy się!

Imię i nazwisko  
Stanowisko  
Doświadczenie  
Oczekiwania

# Co to jest architektura?

# Architecture

**things that are hard to change later**  
**[Martin Fowler]**

**shared understanding of the system design**  
**[Ralph Johnson]**

# Architektura aplikacji

najbliższa programistom  
struktura klas i komponentów  
wzorce projektowe  
frameworki

# Architektura systemu

systemy często się składają z wielu aplikacji  
np. obiekt dokumentów w firmie ubezpieczeniowej

poszczególne aplikacje mogą być w różnych  
technologiach

np. każda warstwa to osobny podsystem

obejmuje integrację z innymi systemami

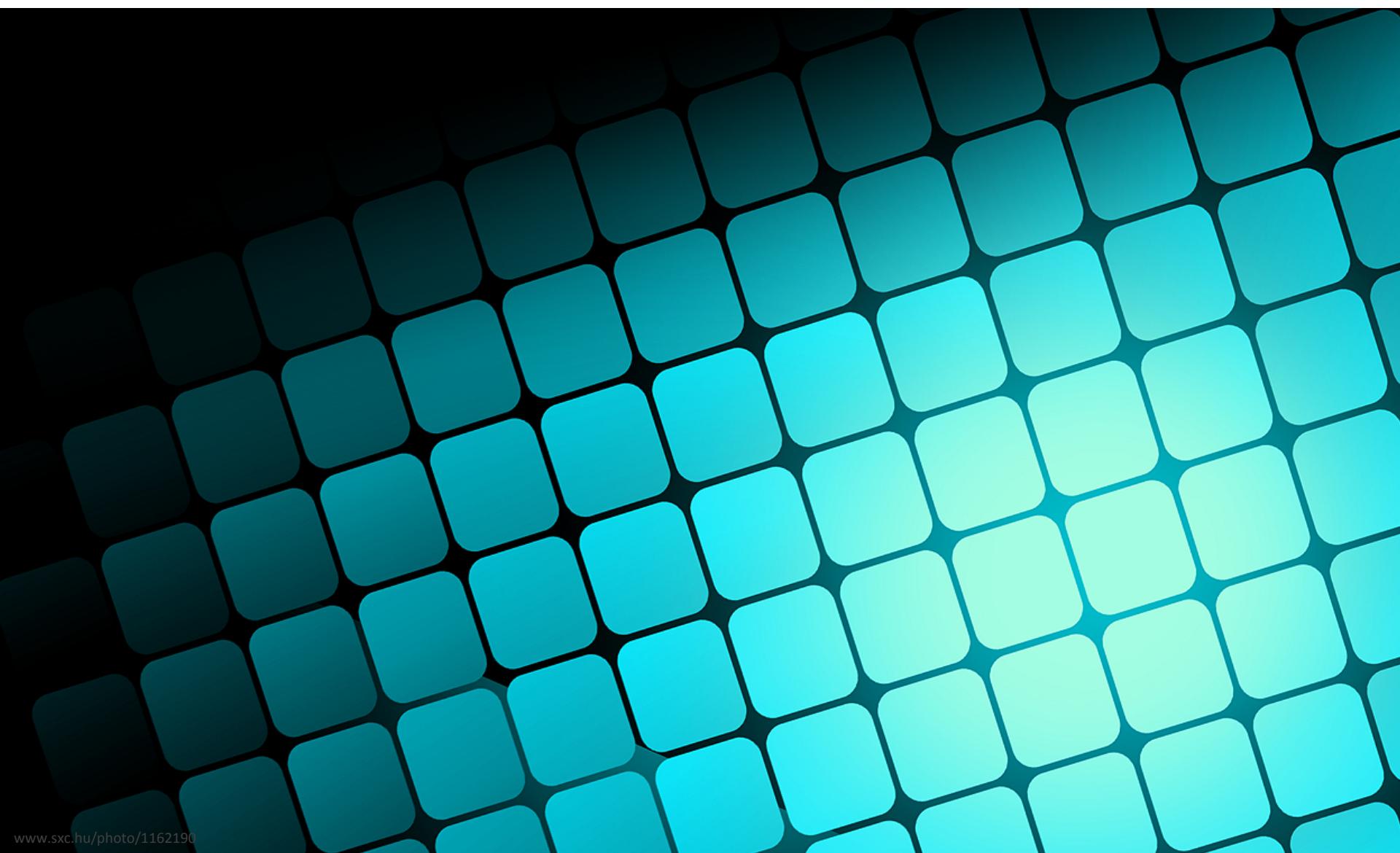
# Architektura aplikacji

+

# Architektura systemu

= Architektura oprogramowania

↪ Architektura korporacyjna

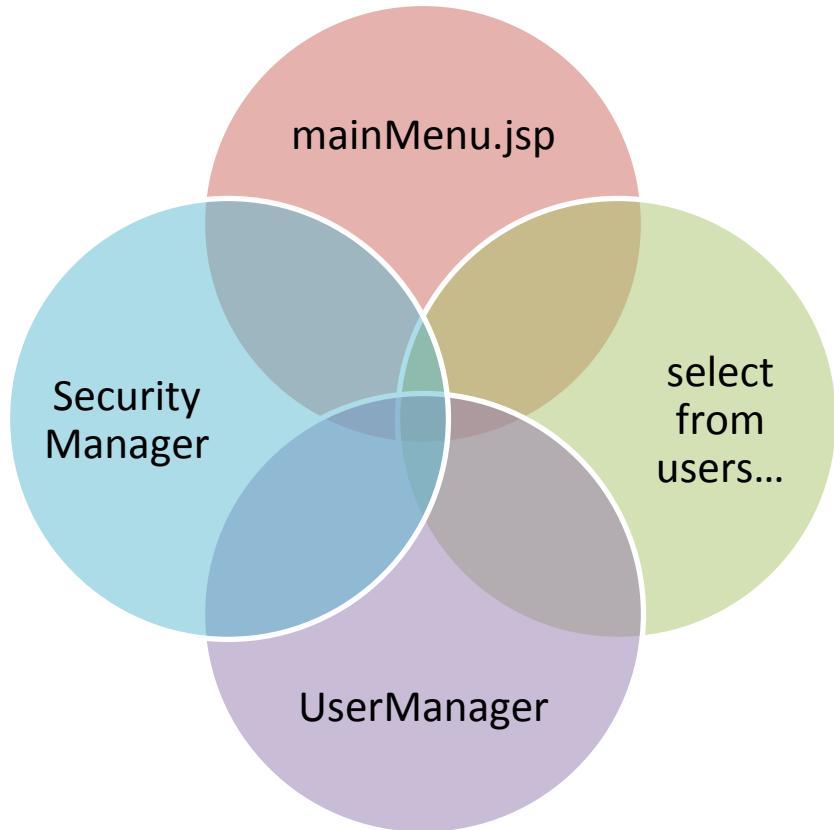




# Silne zależności

Strukturyzowanie kodu

# Zależności w kodzie



- # Jeśli wymiana lub modyfikacja fragmentu systemu powoduje kaskadowe zmiany w innych częściach systemu, to sytuację nazywamy **silną zależnością** (*coupling*) pomiędzy częściami systemu
- # Silne zależności utrudniają utrzymanie oprogramowania



- # Osłabianie zależności w kodzie to podstawowa wytyczna architektoniczna
- # Wzorce projektowe koncentrują się na tworzeniu kodu ze słabymi zależnościami

# Silna zależność: warstwy

```
<html><body>
<table>
<%
while( rs.next() ){
%><tr>
<td><%=rs.getString("id") %></td>
<td><%=rs.getString("date") %></td>
<td><%=rs.getString("email") %></td>
</tr>
<% }%>
<%}
catch(Exception e){e.printStackTrace();}
finally{
if(con!=null) con.close();
}
%>
</body></html>
```

- # Zmiana interfejsu użytkownika powoduje konieczność zmiany w niemal całej aplikacji
- # Niemal na pewno pojawią się duplikacje kodu na wielu stronach *jsp*
- # Strony *jsp* będą rozrastać się w niekontrolowany sposób

# bns it} Silna zależność: warstwy

```
public class Department {  
    public void addEmployee( String name, long depId ) {  
        //...  
        st.executeUpdate( "insert into empls values( 1" + "," + name + "" + ")" );  
        st.executeUpdate( "update deps set empnum=" + 7 + " where id=" + depId + ")" );  
    }  
}  
  
public class EmployeesService {  
    public ResultSet findAllEmployees() {  
        //...  
        return st.executeQuery( "select * from employees;" );  
    }  
}
```

- # System uzależniony jest od konkretnej bazy danych
- # Duże prawdopodobieństwo pomyłek przy pisaniu zapytań SQL
- # Trudne tworzenie i utrzymywanie testów
- # Słaba czytelność kodu

# Silna zależność: elementy statyczne

```
public class OrderProcessor {  
  
    public void process() {  
        PricingService pricingService = PricingService.getInstance();  
        //...  
    }  
}
```

- # Tworzenie zależności, których zmiana ma charakter globalny
- # Trudne testowanie
- # Brak możliwości korzystania mechanizmów obiektowych
- # Utrudnione zrozumienie kodu

# Silna zależność: ukryte zależności

```
public class OrderProcessor {  
  
    public void process() {  
        PricingService pricingService = new PricingService();  
        //...  
    }  
}
```

- # Utrudnione testowanie
- # Zmiana implementacji *PricingService* pociąga za sobą konieczność modyfikacji i rekompilacji kodu
- # W przypadku konstruktorów programista jest „skazany” na logikę zaszytą w komponencie

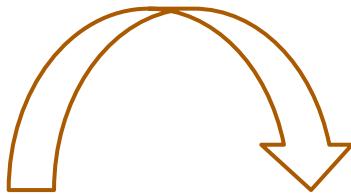
# Silna zależność: środowisko

```
public class Employee {  
  
    public void changeAddress( HttpServletRequest req ) {  
        String street = (String) req.getAttribute( "ADDRESS_STREET" );  
        //...  
    }  
}
```

- # Uruchomienie aplikacji w innym środowisku niż webowe niesie za sobą wiele zmian w całym systemie
- # Utrudnione testowanie ze względu na konieczność symulowania środowiska webowego

bns it }    # }    Struktura kodu  
Strukturyzowanie kodu

# Poszukiwanie struktury



<http://expresskaszubski.pl/pictures/2008-08/dom-rudera-duze.jpg>

[http://knaufblog.pl/wp-content/uploads/2008/04/dom\\_w\\_prymulkach.jpg](http://knaufblog.pl/wp-content/uploads/2008/04/dom_w_prymulkach.jpg)

# Poszukiwanie struktury

Interakcja z otoczeniem

Organizacja  
przetwarzania żądań

Zakres zmienności  
systemu

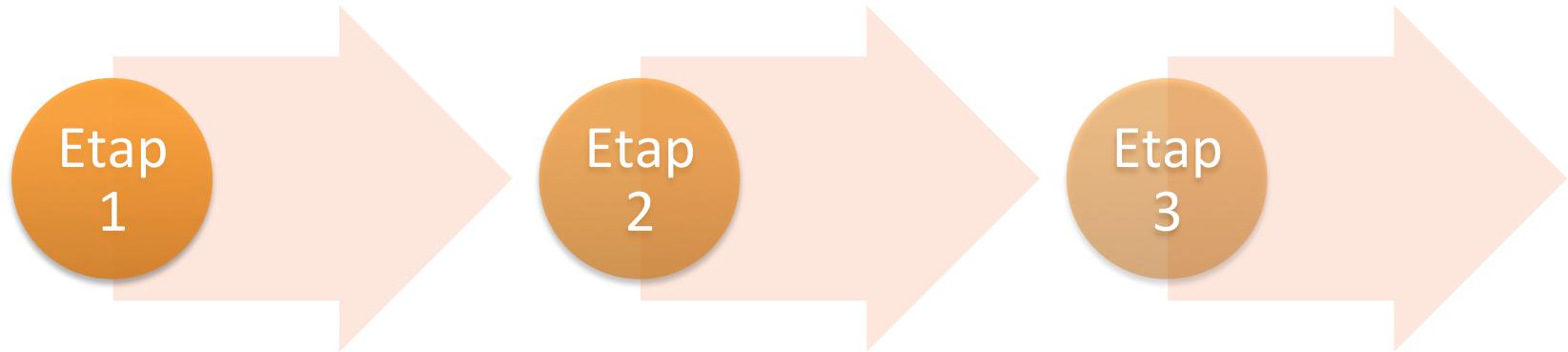
Bezpieczeństwo

Czas życia  
i dostępność

Trwałe  
przechowywanie  
danych



- # W jaki sposób system komunikuje się z otoczeniem?
- # Jak korzystają z niego użytkownicy?
- # Jak współpracuje z urządzeniami?
- # Jak współpracuje z innymi systemami



- # Czy żądanie obsługiwane jest od razu?
- # Czy komunikaty wymieniane są asynchronicznie?
- # Czy można zdefiniować *workflow* dla systemu?
- # Czy system zajmuje się wyłącznie przetwarzaniem danych (*batch processing*)



- # Jak często będą przybywać nowe funkcjonalności?
- # Jak często będą dodawane nowe lub zmieniane systemy współpracujące
- # Jak skaluje się docelowa grupa użytkowników?

# Trwałe przechowywanie danych



- # Gdzie dane będą przechowywane?
- # Jak długo będą przechowywane?
- # Jak często dane muszą być dostępne?
- # Jaka ilość danych będzie przechowywana?
- # Jakie są reguły spójności danych?

# Czas życia i dostępność



- # Czy system ma być dostępny non stop online?
- # Czy jest to aplikacja pudełkowa?
- # Czy jest to system krytyczny?



- # Kto może korzystać z systemu?
- # Które funkcjonalności będą dostępne poszczególnym użytkownikom?
- # Kiedy funkcjonalności mają być dostępne?
- # Na jakich zasadach funkcjonalności mają być udostępniane?

# Poszukiwanie struktury

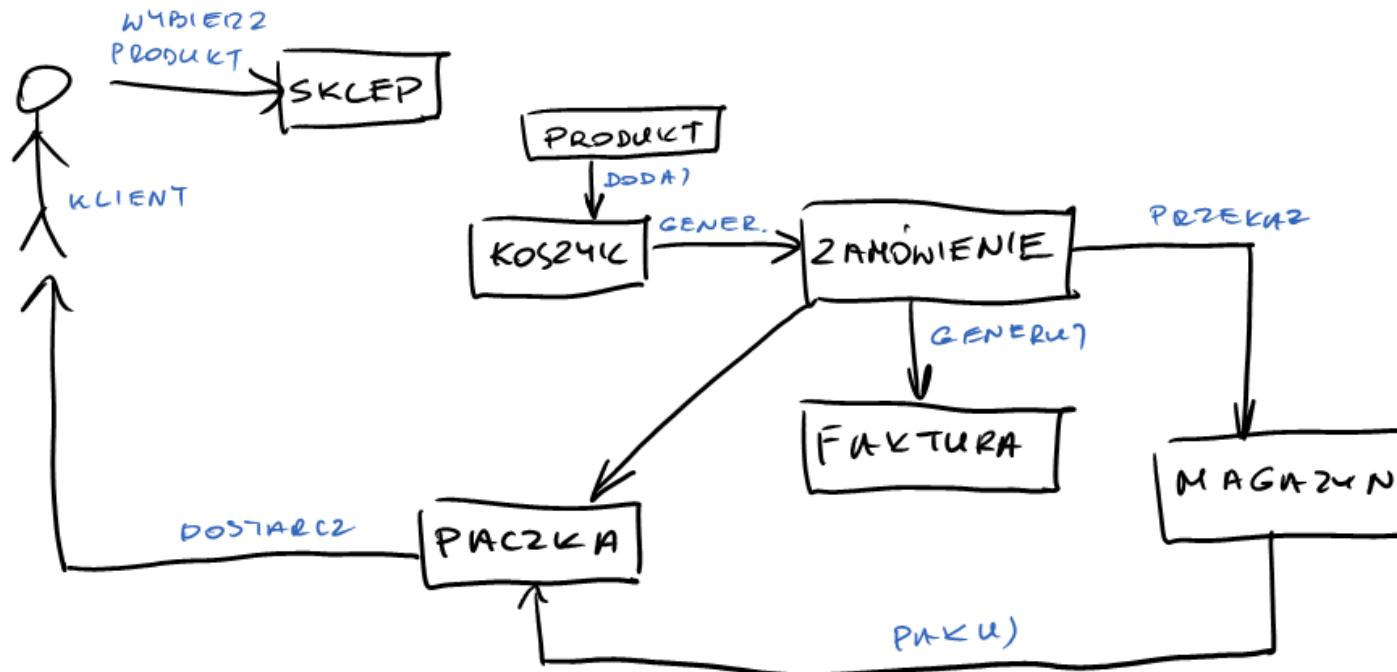
Poprzez definiowanie struktury, wzorce projektowe pomagają programistom skupić się przede wszystkim na **dziedzinie**, na tym **co system ma robić dla użytkownika**



# Podstawowe struktury w kodzie

Strukturyzowanie kodu

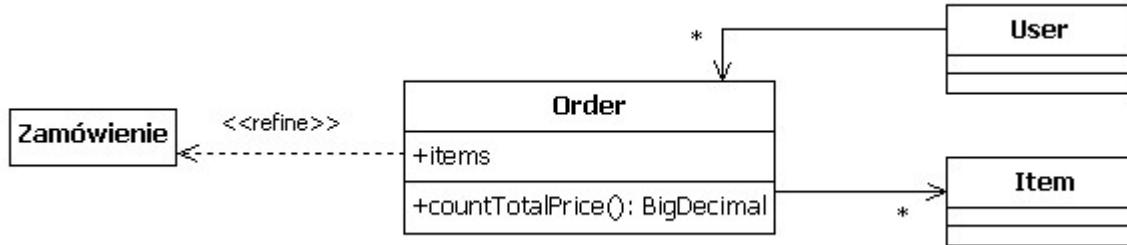
# Model dziedziny



## Model dziedziny (Domain Model)

- # Opisuje fragment struktury oraz dynamiki informatyzowanego procesu
- # Definiuje zakres systemu

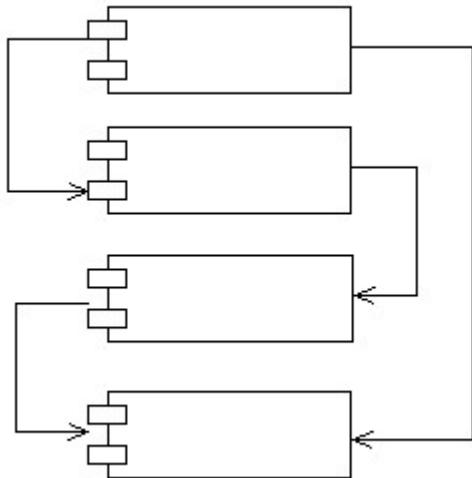
# Obiekt dziedziny



Element modelu dziedziny jest przekształcany w jeden lub wiele **obiektów dziedziny** (Domain Object), które są programistycznym odwzorowaniem rzeczywistego bytów

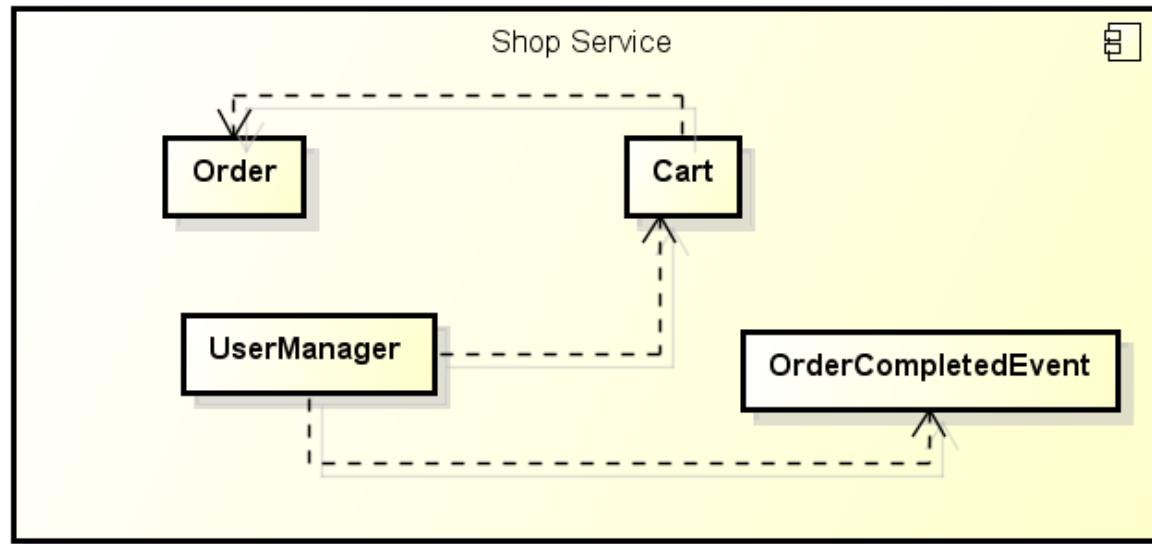
**Obiekt dziedziny** jest zamkniętym komponentem programistycznym, z jednoznacznie zdefiniowanym interfejsem

Obiekt dziedziny realizuje **fragment procesu** zdefiniowanego dla systemu



- # Warstwa grupuje obiekty o analogicznej odpowiedzialności np.: dostęp do danych, interakcja z użytkownikiem
- # Dzielenie na warstwy jest podstawową techniką strukturyzowania kodu
- # Bezpośrednia komunikacja odbywa się od **warstwy wyższej do warstwy niższej**
- # Brak warstw usztywnia architekturę systemu
- # Zbyt duża ilość warstw wprowadza zbyteczne komplikacje

# Dekomponowanie warstwy



powered by astah

- # Warstwa jest dekomponowana na części przez **obiekty dziedziny**
- # W każdym systemie oprócz elementów modelu znajdują się również **obiekty abstrakcyjne**, które łączą system w całość

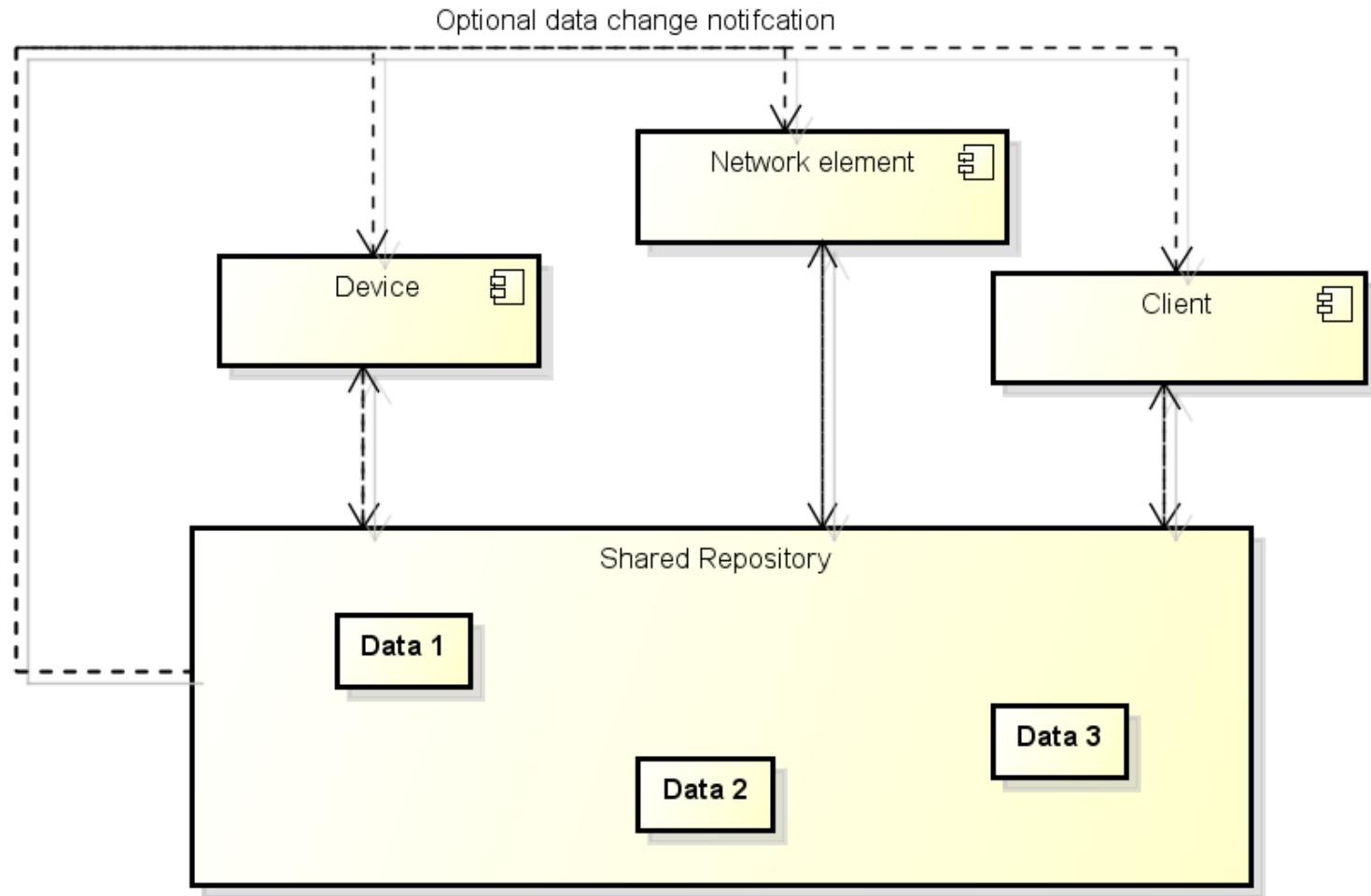
## Obiekty modelu dziedziny

- Modelują fragmenty rzeczywistości

## Obiekty abstrakcyjne

- Wyjątki, zdarzenia, usługi, klasy narzędziowe
- Przejmują część odpowiedzialności, która jest w systemie niezbędna lecz nie należy do żadnego z obiektów dziedziny
- Pełnią funkcję łącznika pomiędzy elementami

# Shared Repository



powered by astah\*

## # Kiedy używać

- Systemy typu *data-driven*
- Komponenty systemu muszą przekazywać między sobą duże ilości danych
- Dane w systemie często się zmieniają

## # Konsekwencje

- Komponenty nie wiedzą o swoim istnieniu
- Komunikacja odbywa się wyłącznie poprzez współdzielone repozytorium
- Repozytorium nie jest tożsame z bazą danych

## # Zagadnienia dla architekta

- Podział funkcjonalności systemu na komponenty
- Zdefiniowanie struktur danych, którymi zasilane będą komponenty
- Zdefiniowanie struktur danych współdzielonego repozytorium
- Problem konkurencyjnego dostępu do danych i synchronizacji
- W jaki sposób komponenty dowiadują się o zmianie danych (odpytywanie, powiadamianie)