Projektowanie i Konstrukcja Systemów Rozproszonych

Wzorce projektowe dla systemów rozproszonych

Projektowanie – problemy

- „trzeba projektować dobrze, bo inaczej jest źle”
  - błędne decyzje projektowe są najdroższymi błędami
- brak powtarzalności projektów
- złożoność rozwiązań
- olbrzymia przestrzeń rozwiązań
  - brak możliwości ich pewnej oceny *a priori*
- ograniczony czas przeznaczony na rozpoznanie problemu
- doświadczenie, wyczucie, wzorce

Wzorce projektowe

- powtarzające się problemy mają podobne rozwiązania
- nie są specyficzne dla konkretnych sytuacji
- stanowią wspólny zbiór definicji dla projektantów
- pomagają w szybki i skuteczny sposób ukierunkować projekt w stronę rozwiązań sprawdzonych przez innych
- zostały „wymyślone” na potrzeby budownictwa

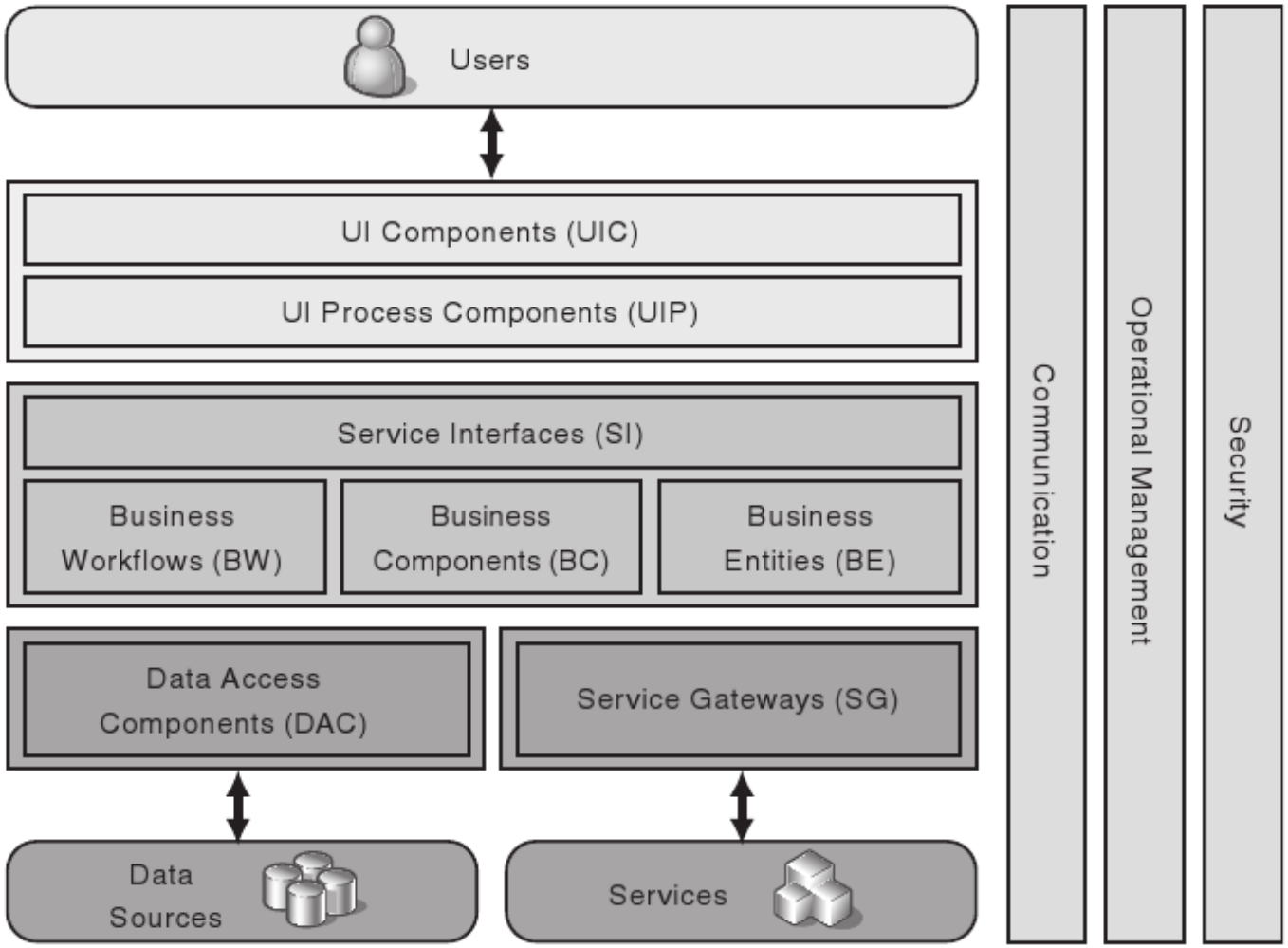
Organizacja wzorców

- podział logiczny top-down
  - *wzorce architektury* – podstawowa organizacja struktury projektu, podział obowiązków i powiązania pomiędzy podsystemami
  - *wzorce projektu oprogramowania* – schemat podsystemów, modułów i ich związków; opisują powtarzającą się strukturę komunikacji modułów wykonujących zadanie
  - *wzorce implementacyjne* – zależne od platformy zalecenia implementacji poszczególnych aspektów komponentów
- „punkt widzenia”
  - baza danych, aplikacja, rozmieszczenie, infrastruktura

Przegląd ważniejszych wzorców

- Aplikacja 3-warstwowa z usługami
- Model-View-Controller
- Zdalna fasada & Data Transfer Object
- Cluster – *Load Balancing* i *Failover*
- Logika biznesowa – *Transaction Script, Domain Model, Table module*
- Mapowanie danych – *Table Gateway, Row Gateway, Active Record, Data Mapper*
- Zachowanie obiektów i danych – *Unit of Work, Identity Mapper, Lazy Load, Query Object*
- Stabilność i pojemność systemu

Aplikacja 3-warstwowa z usługami



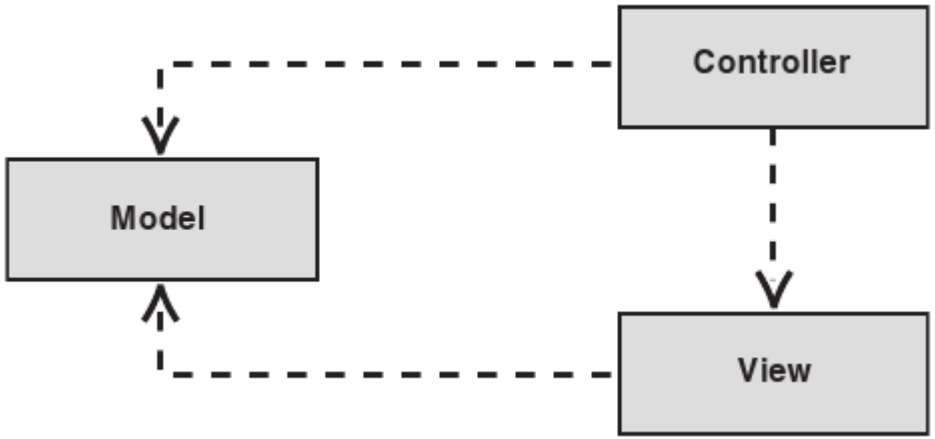
Aplikacja 3-warstwowa z usługami

- prezentacja
  - komponenty interfejsu użytkownika
  - proces kontrolujący obsługę programu przez użytkownika (*Application Controller*)
- logika biznesowa (domenowa)
  - komponenty biznesowe – implementacja autonomicznych obiektów i prostych procesów biznesowych
  - procesy biznesowe – wysokopoziomowe reprezentacje działań biznesowych, np. obsługa zamówienia
  - encje biznesowe – kontenery danych
  - interfejsy usług – interakcja ze światem zewnętrznym ukrywająca szczegóły implementacyjne

Aplikacja 3-warstwowa z usługami

- warstwa danych – interfejs pomiędzy bazą danych a logiką biznesową
  - komponenty danych – zbierają kod logiki dostępu do danych w jedno miejsce i minimalizują wpływ zmiany źródła danych na logikę biznesową
  - bramy do usług – ukrywają szczegóły komunikacyjne niezbędne do zdalnej komunikacji
- usługi podstawowe – zbiór usług wykorzystywanych przez każdą warstwę aplikacji

Model View Controller



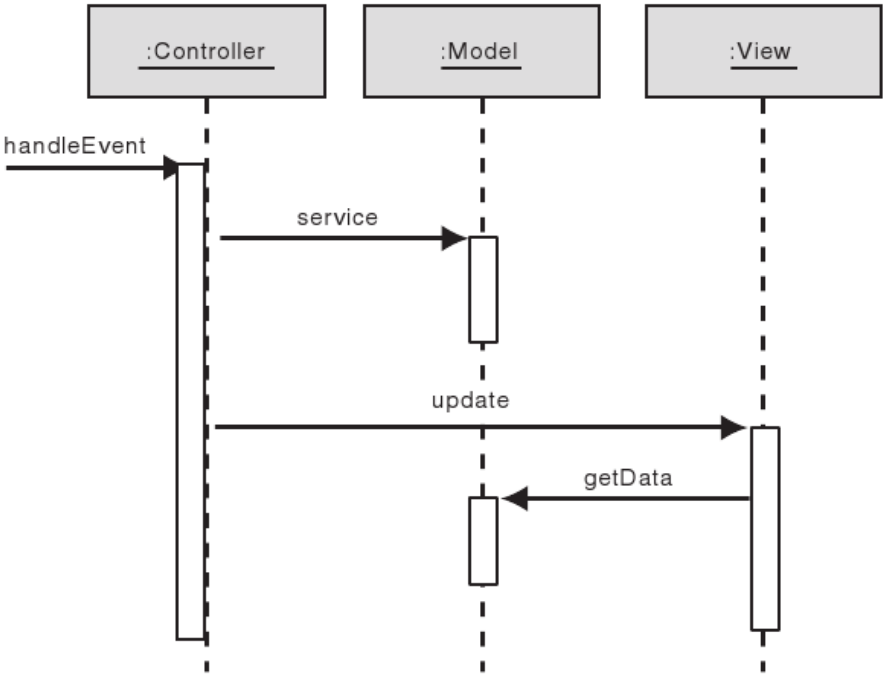
- model – niezwiązane z wyglądem UI obiekty warstwy biznesowej
- view – reprezentacja danych dla użytkownika
- controller – przetwarzanie poleceń użytkownika, manipulacja *model* i kontrolowanie odświeżeń *view*

Model View Controller

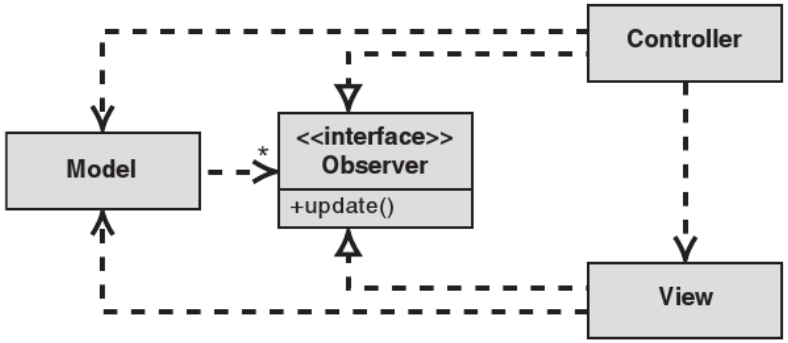
- separacja *view* i *model* – jedna z najważniejszych heurystyk projektowych
  - tworzenie modelu i widoku to 2 osobne zagadnienia – kwestia podziału zadań, preferencji, umiejętności
  - użytkownicy chcą widzieć te same dane niezależnie od prezentacji – łatwe tworzenie wielu widoków dla modelu
  - testowanie kodu modelu jest łatwiejsze niż prezentacji
- separacja *view* i *controller*
  - mniej istotne, użyteczne np. przy aplikacjach webowych i tworzeniu zachowania edytowalnego i nieedytowalnego widoku – 1 widok, ale 2 kontrolery
  - *controller MVC* nie jest kontrolerem całej aplikacji

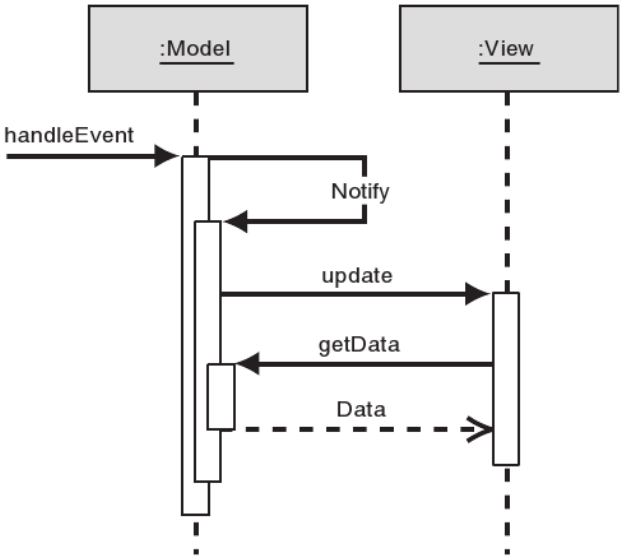
Pasywny i aktywny MVC

- pasywny



- aktywny – zewnętrzne zmiany w modelu



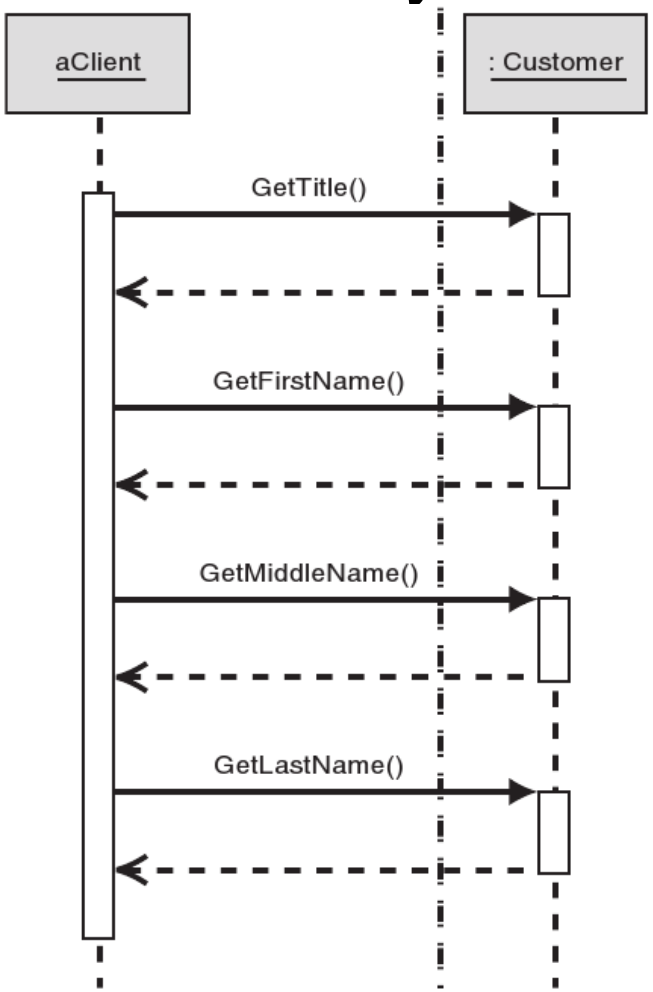


Zdalna fasada & Data Transfer Object

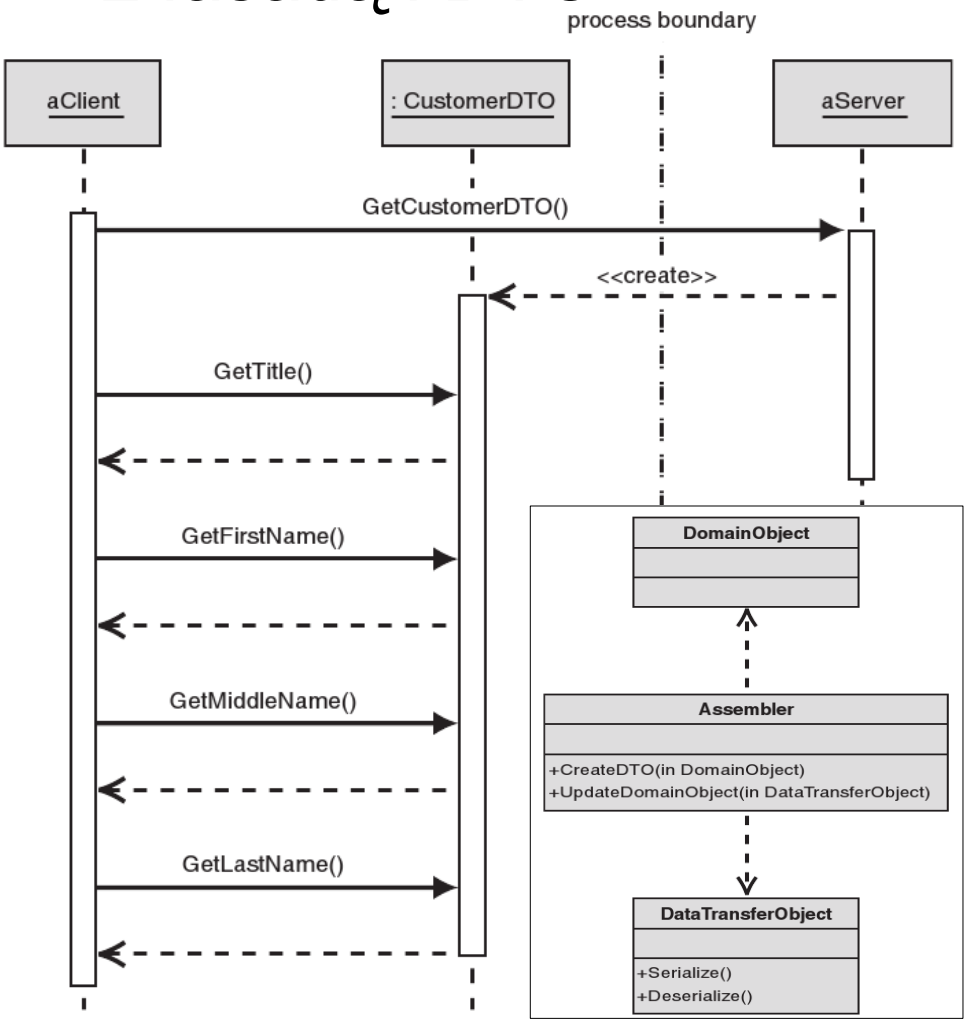
- zdalna fasada – dostarcza gruboziarnistą fasadę do drobnoziarnistych wywołań komponentów biznesowych
  - zbiera wiele zdalnych operacji na obiektach biznesowych w jedno wywołanie
  - często tworzona odpowiednie do konkretnej operacji biznesowej wywoływanej z UI
- Data Transfer Object – obiekt przenoszący dane pomiędzy węzłami w celu minimalizacji liczby wywołań i parametrów
  - serializacja DTO
  - konstrukcja DTO z/do obiektów biznesowych (*Assembler*)

Zdalna fasada & Data Transfer Object

- bez fasady i DTO



- z fasadą i DTO



Cluster

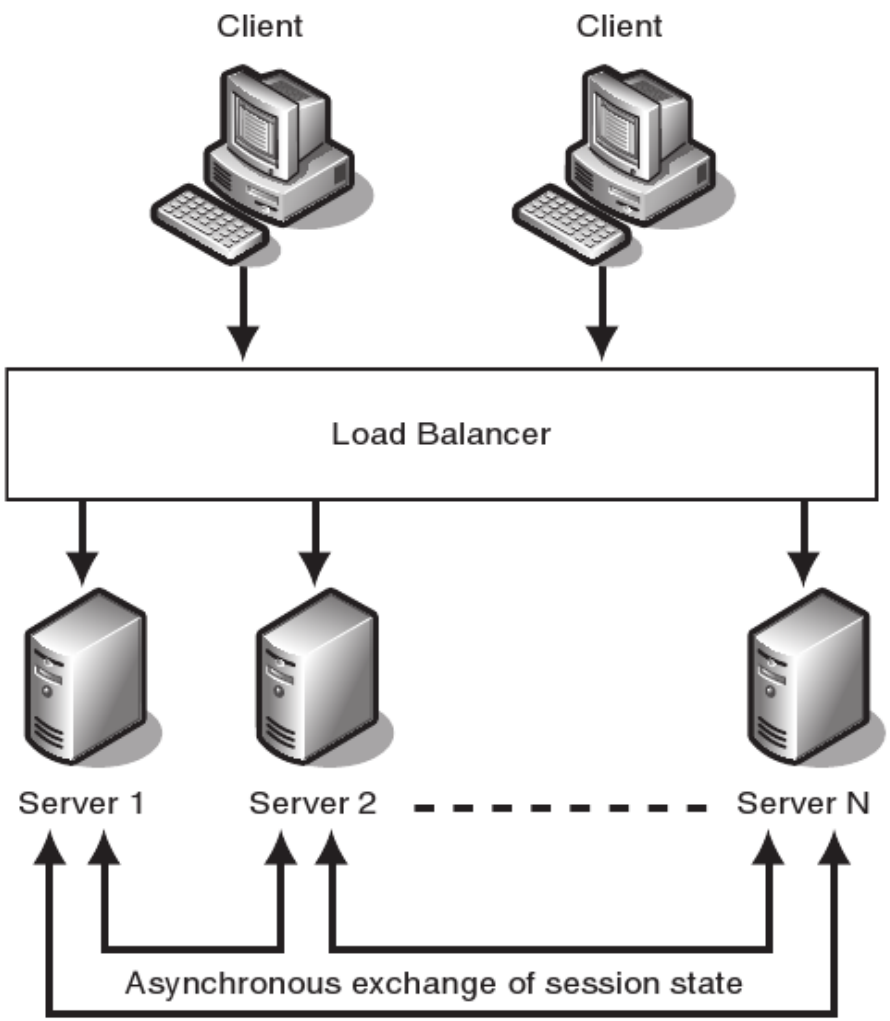
Cluster z równoważeniem obciążenia

- dodatkowa warstwa lub aplikacja pomiędzy żądaniami klienta a aplikacją bazową – *load balancer*
  - wybór algorytmu równoważenia – (ważony) algorytm karuzelowy, monitorowanie obciążenia, równoważenie liczby połączeń, ...
  - możliwość wykrywania awarii i rekonfiguracja
- aplikacja bazowa instalowana na *n* maszynach, pomiędzy które należy równomiernie rozłożyć całe obciążenie systemu
- problem: aplikacje stanowe i zarządzenie sesją użytkownika

Cluster z równoważeniem obciążenia – sesje

- sesja po stronie użytkownika
  - nie wymaga rozszerzania clustra, jednak obciąża klienta i łącza
- sesja w bazie danych
  - traktowana jako transakcja od przesłania żądania do wysłania odpowiedzi; za wydajność odpowiada baza
- sesja po stronie serwera
  - przywiązanie sesji – każde wywołanie sesji jest kierowane przez *load balancer* do tego samego serwera
  - centralny niezależny serwer sesji
  - rozproszone replikowane sesje

Cluster z równoważeniem obciążenia – sesje



Cluster o podwyższonej odporności na awarie

- aplikacja instalowana jest na podstawowych i zapasowych serwerach; w razie awarii serwera podstawowego jego funkcje przejmuje odpowiadający mu serwer zapasowy
- konieczna jest detekcja błędów
  - modele *push* i *pull*
  - modele zaawansowane, np. *flexible fanout*
  - dokładność, poprawność, czas reakcji

Cluster o podwyższonej odporności na awarie

- synchronizacja stanów serwera – po awarii serwer zapasowy musi odtworzyć stan przetwarzania serwera podstawowego
  - log transakcji – odtwarzanie stanu analogiczne do odtwarzania bazy danych po awarii
  - hot standby – operacje serwera podstawowego są natychmiast kopiowane na serwer zapasowy
  - współdzielony dysk – dane o stanie serwera zapisywane są w sieciowym systemie plików czy rozwiązaniach SAN
- wykrywanie aktywnego serwera
  - rejestr znaczników aktywności serwerów
- failover musi być uwzględniony po stronie klienta

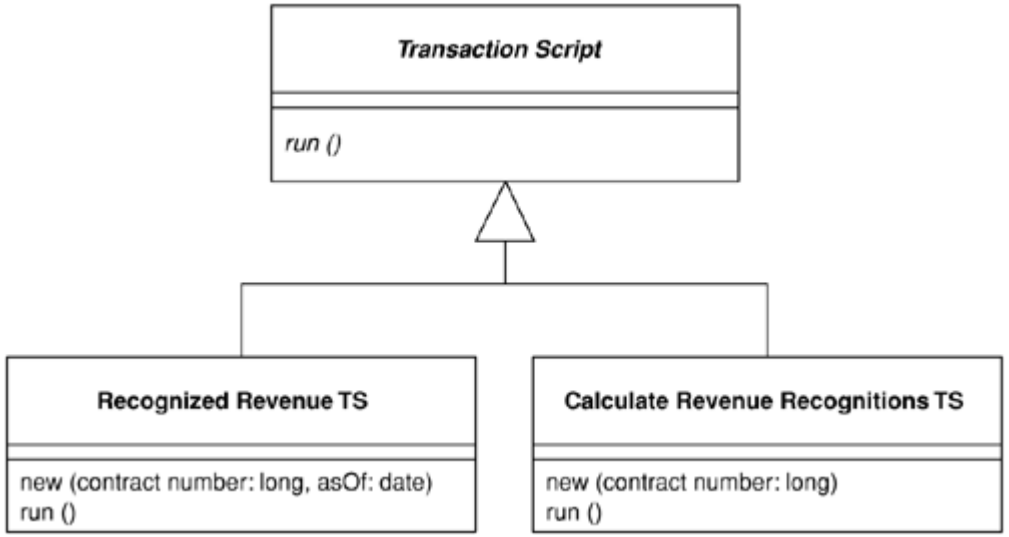
Logika biznesowa a dostęp do danych

Logika biznesowa – *transaction script*

- organizuje logikę biznesową w procedury, z których każda odpowiada za wykonanie jednego żądania od warstwy prezentacji
  - procedura zawiera całą logikę potrzebną do wykonania procesu, w tym manipulację danymi i ich utrwalanie
- wzorzec odpowiedni tylko dla bardzo małych rozwiązań
  - ✓ likwiduje nakład pracy związany z dokładnym modelowaniem obiektów logiki domenowej
  - ✓ przeważnie zawiera całą transakcję
  - ✗ łatwo o duplikację kodu
  - ✗ trudno rozwijalne podejście

Logika biznesowa – *transaction script*

- pozwala na wyprowadzenie wspólnych interfejsów ułatwiających przekazywanie żądań z UI



Logika biznesowa – *table module*

- pojedyncza instancja wzorca zajmuje się obsługą całej logiki biznesowej związanej z jedną tabelą w bazie
- działa przeważnie na obiektach typu *Record Set*, odwzorowujących zbiór krotek pobranych z bazy, jednak nie koniecznie z niej pochodzących
- odpowiada podejściu zastosowanemu przez Microsoft przy projektowaniu ADO.NET i kontrolek SWF
  - ✓ bardzo prosto integruje się z bazą danych poprzez *Table Gateway*
  - ✗ wymaga innej niż referencja identyfikacji obiektów (przeważnie klucz podstawowy)

Logika biznesowa – *domain model*

- obiektowy model dziedziny programu
- logika biznesowa może być bardzo złożona – łatwiej nią zarządzać, gdy jest siecią mniejszych obiektów
  - model prosty – w mniejszych rozwiązaniach, łatwo mapuje się na bazę danych, np. przez *Active Record*
  - model złożony – zawiera dużą liczbę różnych obiektów, nie mapujących się bezpośrednio na bazę danych – może się z nią integrować np. przez *Data Mapper*
- tworzy osobną warstwę architektury, którą należy uniezależniać od innych warstw

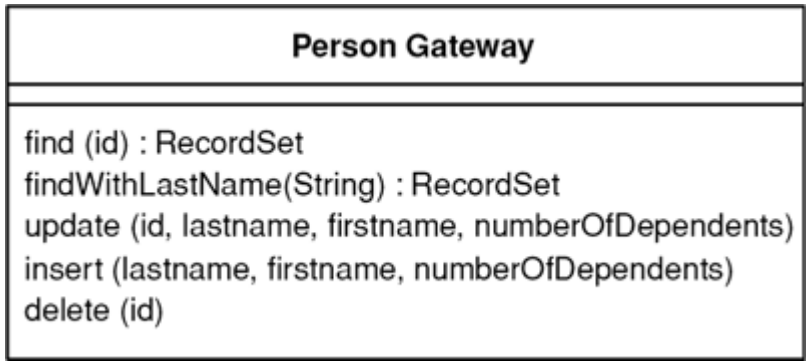
Logika biznesowa – *domain model*

- ✔ łatwo otoczyć model dziedziny fasadami – np. *Remote Fasade* lub *Service Fasade*
- ✔ pozwala na łatwe rozwijanie aplikacji
- ✔ intensywnie używa wzorców dziedziczenia, strategii, itd.
- ✗ większy koszt tworzenia małych rozwiązań
- ✗ „przerastanie” obiektów – tendencja obiektów do rozszerzania o kolejne metody, w tym używane sporadycznie
- ✗ skomplikowana wymiana danych z bazą danych

Mapowanie danych

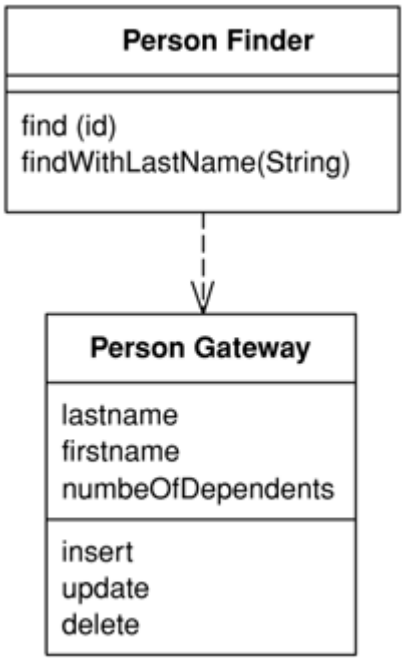
Mapowanie danych – *Table Gateway*

- obiekt działający jako brama do komunikacji z tabelą bazy danych; jedna instancja obsługuje wszystkie wiersze
  - realizuje wszystkie operacje pobrania i edycji danych (CRUD)
  - ukrywa szczegóły dostępu do danych (np. zapytania SQL)
  - ✔ łatwy w użyciu interfejs do bazy danych
  - ✔ dobra integracja z *Transaction Script* i *Table Module*
  - ✗ integracja z *Domain Model*



Mapowanie danych – *Row Gateway*

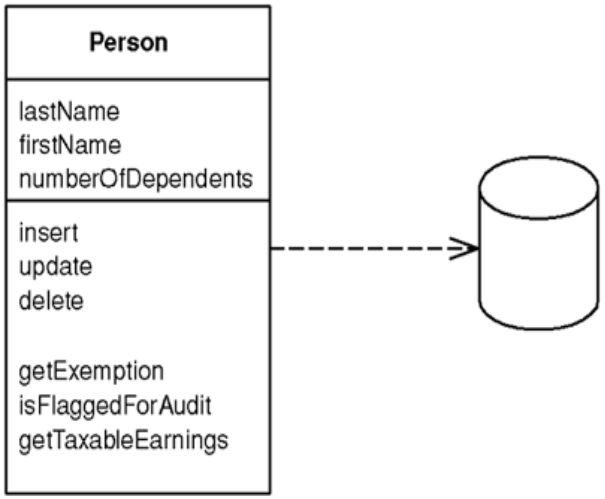
- obiekt działający jako brama do komunikacji z tabelą bazy danych; odwzorowuje pojedyncze wiersze; jedna instancja odpowiada jednej krotce



- – ukrywa szczegóły dostępu do danych (np. zapytania SQL)
- – konwertuje kolumny na pola i odwrotnie
- ✔ rozdziela obiekt od jego źródła danych
- ✔ bardzo dobra integracja z *Transaction Script*
- ✗ problem jednoczesnych operacji na bazie wykonywanych przez wiele instancji
- ✗ drobnoziarnistość zapytań

Mapowanie danych – *Active Record*

- obiekt działający jak *Row Gateway*, jednak zawierający również zależną od danych logikę



- proste podejście – integracją z bazą bezpośrednio w obiekcie logiki domenowej
- konwertuje kolumny na pola i odwrotnie
- zawiera operacje CRUD
- poprzez metody statyczne przeszukuje źródła danych

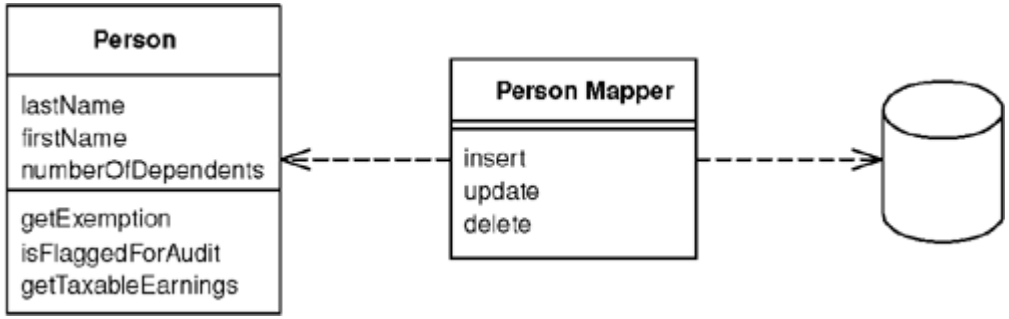
✔ tworzy bezpośrednio logikę domenową

✗ kiepska integracja ze złożonymi *Domain Models* – mapowanie na/z wielu tabel

✗ wiąże schemat bazy ze schematem klas

Mapowanie danych – *Data Mapper*

- obiekt zajmujący się mapowaniem pomiędzy bazą danych, a logiką domenową, uniezależniający je od siebie nawzajem i od samego *Data Mapper*



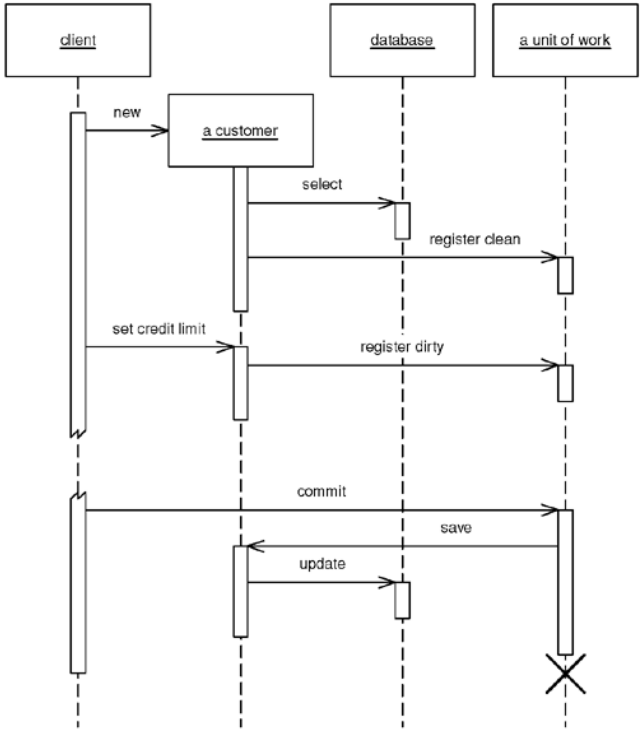
Mapowanie danych – *Data Mapper*

- izoluje obiekty biznesowe od jakiejkolwiek wiedzy o źródle danych
- pozwala na realizację dziedziczenia, składowania w wielu tabelach itp
- ✔ pozwala na optymalizacje, np. prefetching
- ✔ dobrze integruje się z innymi wzorcami: *Unit of Work*, *Identity Map*, *Lazy Load*
- ✔ pozwala na niezależną ewolucję schematu bazy i klas
- ✔ ułatwia testowanie niezależne od źródła danych
- ✗ koszt implementacji – przeważnie opłacalne tylko przy złożonych *Domain Models*

Zachowanie obiektów i danych

Zachowanie obiektów i danych – *Unit of Work*

- przechowuje listę obiektów zmodyfikowanych przez transakcję biznesową i koordynuje zapis zmian i kontrolę współbieżności
  - rejestruje zmiany wynikające z transakcji biznesowej, a po jej zakończeniu wykonuje odpowiednią transakcję bazodanową
  - musi kontrolować współbieżność, w tym spójność odczytów
  - ✔ pozwala uniknąć wielu małych zapytań do bazy, wysyłanych po każdej modyfikacji
  - ✔ grupowanie i optymalizacja zapytań
  - ✗ wymaga pamiętania o rejestracji obiektu lub modyfikacji obiektu



Zachowanie obiektów i danych – *Identity Map*

- zarządza ładowaniem obiektów z bazy danych, zapewniając, że każdy zostanie załadowany tylko raz

– przeważnie 1 mapa na 1 tabelę

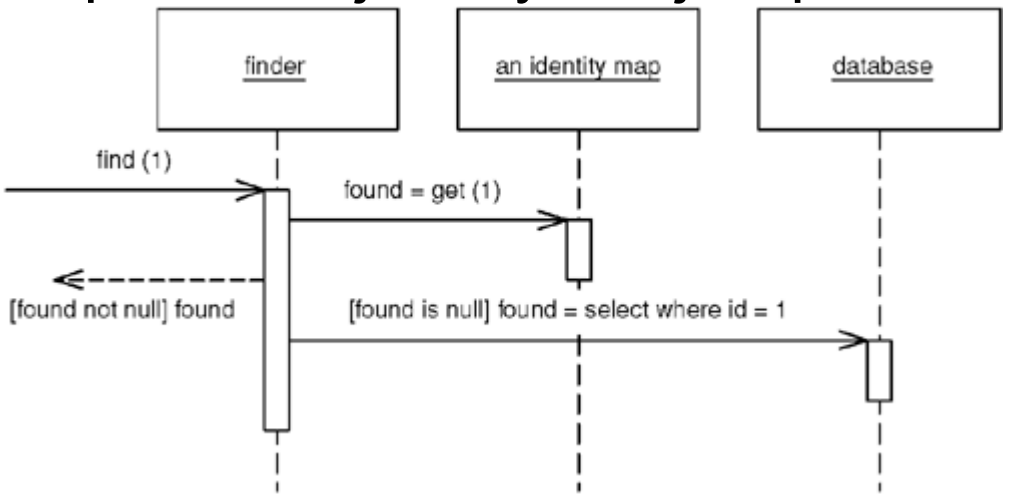
– wybór klucza mapowania

✔ eliminuje niepotrzebne odwołania do źródeł danych

✔ eliminuje błędy wynikające ze współbieżnej modyfikacji kopii tego samego obiektu

✔ pełni rolę *cache*

✗ może wymagać zarządzania współbieżnością



Zachowanie obiektów i danych – *Lazy Load*

- ładowanie obiektu w sposób, który nie pobiera wszystkich obiektów powiązanych, jednak dostarcza je automatycznie, gdy stają się potrzebne
  - zależności między obiektami – mogą powodować niezamierzone pobieranie dużych ilości danych
  - musi uważać na problem wielu małych zapytań
  - możliwe realizacje
    - *Lazy Initialization* – każde pole jest początkowo *null*, a przy 1. odwołaniu do niego pobierana jest rzeczywista wartość
      - sprawdza się w *Active Record, Table Gateway, Row Gateway*
      - tendencja do wiązania bazy danych z obiektem – wada w przypadku *Data Mapper*

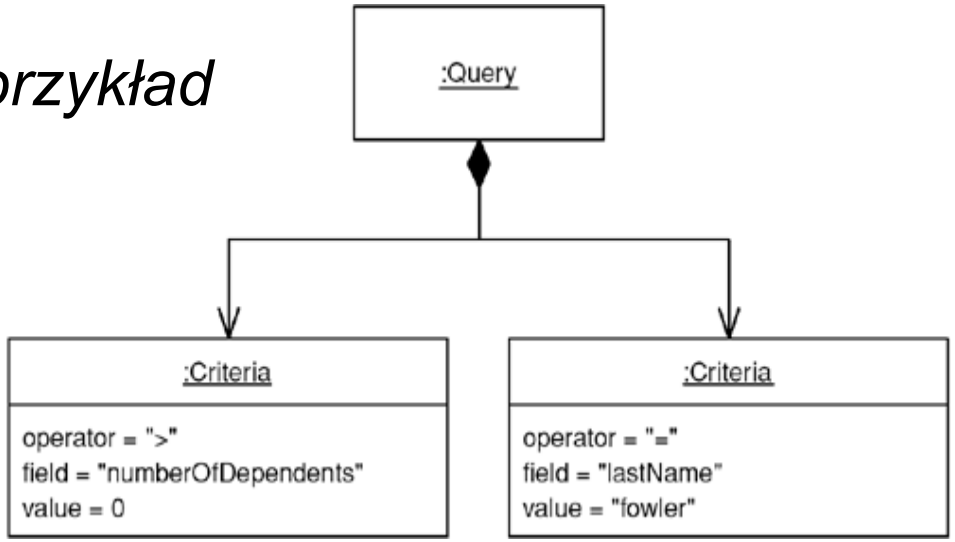
Zachowanie obiektów i danych – *Lazy Load*

- *Virtual Proxy* – obiekt wyglądający jak obiekt podstawowy, jednak ładujący swoje pola w miarę wykonywania metod
  - ✓ wygląda identycznie jak obiekt, który powinien być na jego miejscu
  - ✗ trudna identyfikacja
  - ✗ możliwe jest wiele proxy do tego samego obiektu podstawowego
- *Value Holder* – obiekt-wrapper wokół podstawowego obiektu, zawiera pole *Value* ładujące ten obiekt
  - ✗ ukrycie typu obiektu wewnętrznego
- *Ghost* – prawdziwy obiekt, jednak istniejący tylko częściowo – ma wczytane pola ID, reszta wczytywana przy odwołaniu
  - pozwala na natychmiastowe użycie identyfikacji, np. dla potrzeb *Identity Map*

Zachowanie obiektów i danych – *Query Object*

- obiekt reprezentujący zapytanie
  - reprezentuje warunki zapytania wyrażone w kodzie
    - nie narzuca znajomości języka zapytań na użytkowników
    - pomaga unikać literówek
    - pozwala zmienić źródło danych bez zmiany logiki specyfikującej zapytania
    - możliwość *zapytania-przez-przykład*

```
QueryObject query = new QueryObject
(typeof(Person));
query.addCriteria (Criteria.greaterThan
("numberOfDependents", 0));
query.addCriteria (Criteria.matches
("lastName", "f%"));
query.Execute ();
```



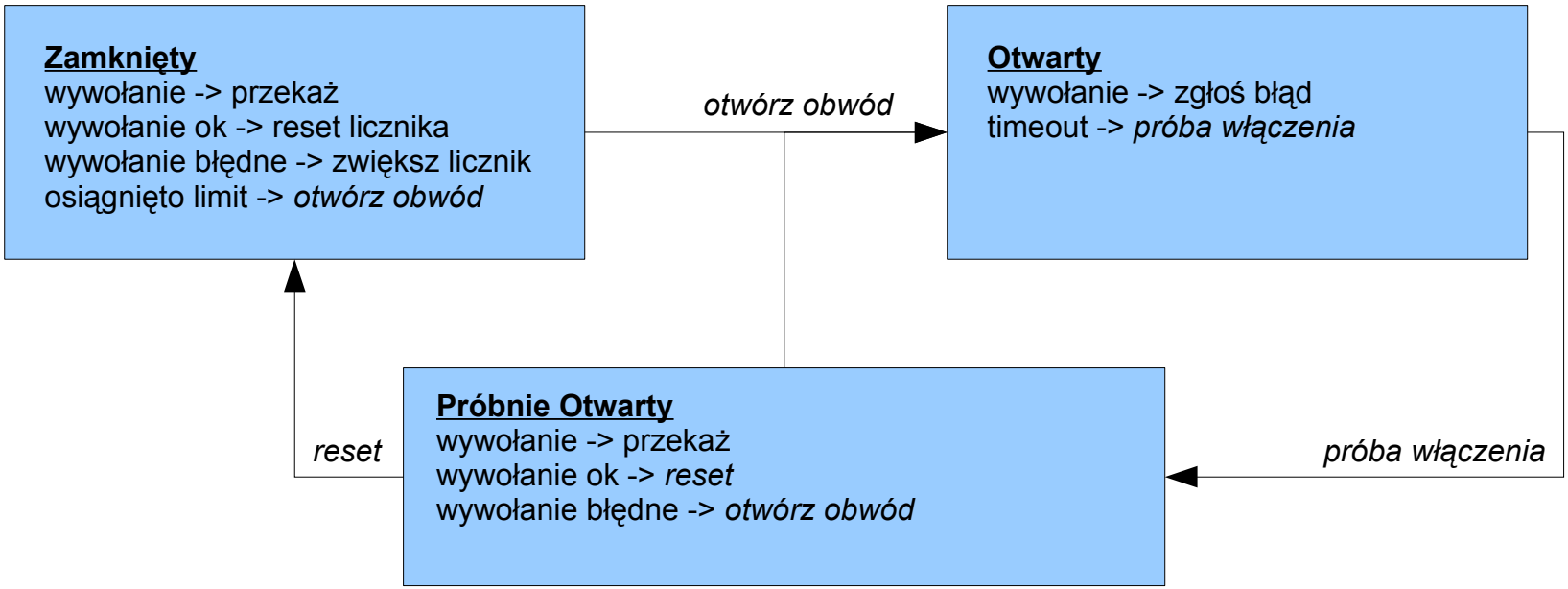
Zachowanie systemów

Stabilność Systemu

- Timeouts
  - sieci są zawodne – propagacja błędów na system
  - izolacja systemu od innych błędów
  - dostęp do puli zasobów – nie zawsze możliwy
  - rozsądne ponowienia (np. mail server)
- Fail Fast
  - najgorsza jest wolna odpowiedź z błędem
  - jeśli da się przewidzieć błąd – Fail Fast
  - najpierw sprawdź dostępność zasobów
  - walidacja wejścia

Stabilność Systemu

- Circuit Breaker
  - zapewnić awarię podsystemu zamiast całości
  - „nie rób tego, co boli” – Fail Fast
  - integracja z innymi systemami



Stabilność Systemu

- Bulkheads
  - „uratuj chociaż część okrętu”
  - podział pojemności operacyjnej systemu na części
  - każda część potrzebuje swoich rezerw zasobów
  - problem wyboru granularności
  - np. osobna pula zasobów dla klientów kluczowych
- Steady State
  - systemy produkcyjne powinny obejść się bez człowieka
  - czyszczenie danych – baza danych, archiwizacja, pliki, ...
  - czyszczenie logów

Stabilność Systemu

- Handshaking
  - możliwość uzgodnienia zasad wykonania operacji między 2 stronami
  - sprawdzenie „stanu zdrowia” – możliwości wykonania żądania
  - wprowadza dodatkowy koszt
- Test Harness
  - środowisko do testowania systemu jako całości, „złośliwe testy”, zachowania niekreślone specyfikacją
  - testy awarii (pod)systemu i systemów zależnych
  - nie Mock Objects, ale „złośliwe” emulatory systemów

Pojemność Systemu

- Connection Pool
  - podstawowe podejście do obsługi połączeń
  - każde wywołanie na połączeniu z puli musi mieć Timeout
  - dostosowanie rozmiaru puli dla maksymalnej wydajności
- Cache
  - ostrożne cache'owanie
    - limit rozmiaru
    - czyszczenie
    - pomijanie obiektów trywialnych
    - pomijanie szybko zmieniających się obiektów

Różne

- Precompute – obliczanie treści statycznych „z góry”
- konfiguracja garbage collector'a
- Virtual IP – tanie i proste rozwiązanie
- Sensowne pliki konfiguracyjne
- Start i zatrzymanie systemu, a jego interakcje
- Przezroczystość systemu – dane o jego stanie dostępne dla administratorów

Literatura

- „Patterns of Enterprise Application Architecture” – Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford; Addison-Wesley
- „Release It!” – Michael T. Nygard
- „Enterprise Solution Patterns Using Microsoft .NET” – MS Corporation (David Trowbridge i in.); MS Press