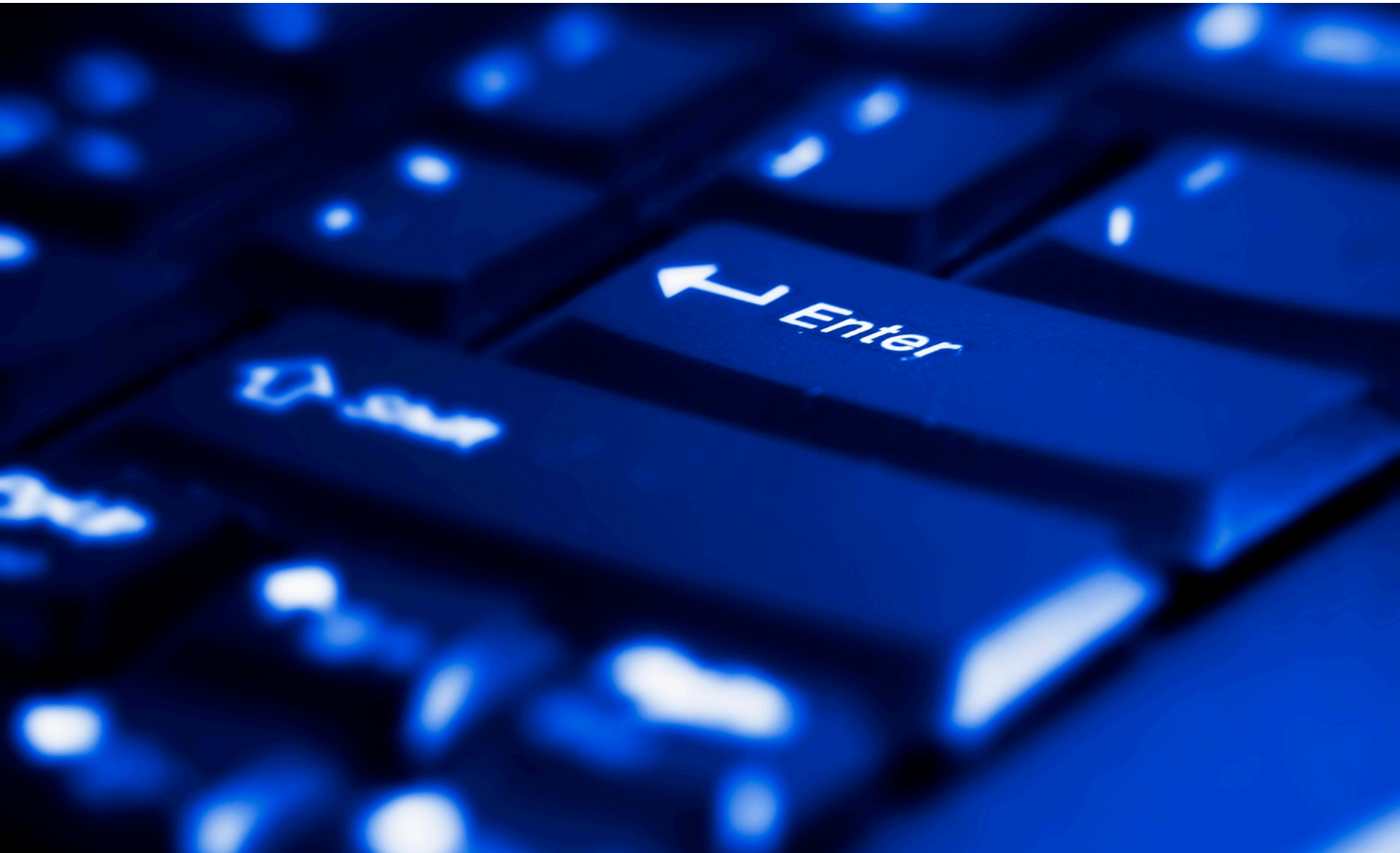
bns it

Oprogramowanie gotowe na wszystko

Wzorce projektowe i refaktoryzacja do wzorców



Problematyka oprogramowania dla wielu klientów

- Elastyczność => Złożoność
- Reużywalność zmniejsza używalność
  - UML i RAD Case Software
- To jest trudne

Zagadnienie biznesowe i techniczne

- Dobry design produktu dla wielu klientów wymaga zaangażowania i leadershipu biznesowego
- Analiza biznesowa i systemowa
- Ewolucja architektury

Analiza wymagań

- Wymaganie określa potrzebę lub oczekiwania klienta wobec produktu
- Wymagania w produkcie manifestują się jako funkcjonalność (feature)
- W przypadku wielu klientów dokonujemy analizy:
  - Funkcjonalności wspólnych
  - Funkcjonalności zmiennych

Typy funkcjonalności

Funkcjonalności wspólne

- dla wszystkich klientów/produktów (np. podwozie)

Funkcjonalności opcjonalne

- dla wybranych lub jednego klienta/produktu (np. opcja sportowa, automat, odkryty dach)

Funkcjonalności alternatywne

- wybierz jedną z możliwości (np. silnik 2.0, 2.5, diesel, benzyna; kolorowy vs czarno-biały wyświetlacz)
- często alternatywy mają domyślną wersję (np. typ silnika)

Typy funkcjonalności

- Funkcjonalności parametryzowane
  - wymaga sparametryzowania na poziomie konfiguracji (per klient/produkt)
  - np. poziom autentykacji: ilość prób wpisywania hasła, długość hasła, dozwolone znaki
- Funkcjonalność zależna
  - jedna funkcjonalność zależy od obecności innej funkcjonalność

Ćwiczenie analiza funkcjonalności

- Dla wybranego systemu, który obsługuje wielu klientów lub wiele produktów dokonaj analizy funkcjonalności:
  - Funkcjonalności wspólne
  - Funkcjonalności opcjonalne
  - Funkcjonalności alternatywne
  - Funkcjonalności parametryzowane
  - Funkcjonalność zależna

Wzorce projektowe w służbie rozszerzalności

- W jaki sposób poszczególne wzorce mogą wspierać rozszerzalność?
- Dla określonych w poprzednim ćwiczeniu funkcjonalności, określ jakie wzorce mogłyby pomóc przygotować do różnicowania produktu?

Wzorce projektowe w służbie rozszerzalności

- # Strategy
- # Template method
- # Decorator
- # Factory
- # Facade
- # State
- # Observer/Events/PubSub
- # Command
- # Bridge/Drivers

Dependency Injection

- Dependency Injection jako baza do konfigurowalności „szkieletu systemu”
- Uzupełnia mechanizm kompozycji umożliwiający składanie (komponowanie aplikacji) na poziomie konfiguracji
- Narzędziowo wspierane przez kontener DI

Wzorce architektoniczne - Warstwy

- Wydzielają odpowiedzialności

- Pomagają podmienić warstwę dla konkretnego klienta
  - np. z pomocą wzorca DAO/Repository zmiana bazy danych
  - np. warstwy prezentacji dla różnych klientów (UI jest najtrudniej reużywalną warstwą)

DDD i Bounded Contexts

- Im mniejsza granulacja tym większa reużywalność
- Aplikacja jako zbiór autonomicznych modułów
- Strategie dzielenia domeny
  - im chcemy większą niezależność tym bardziej odważnie dzielimy (to ma swój koszt)
  - **Przykład: podział systemu na Bounded Countext włącznie z modelem dziedziny**

Komunikacja sterowana zdarzeniami

- Architektura Event-Driven uniezależnia od siebie elementy, które komunikują się zdarzeniami
- Umożliwiają realizację opcjonalnych modułów dla aplikacji
- Do wykorzystania: koncepcje architektury Event-Driven i Reactive
- Ale uwaga
  - sytuacje wyjątkowe trudniejsze w obsłudze
  - trudny monitoring

Multitenancy

- Strategia tworzenia aplikacji, która w ramach jednej instancji obsługuje różnych klientów
- Charakterystyczne dla systemów typu Saas wdrażanych w modelu Cloud
- Większość danych w systemie zawiera dodatkową informację (TenantId)

Konfiguracja zewnętrzna

- Za pomocą pliku konfiguracyjnego określamy opcjonalne lub parametryzowane elementy systemu
- Mogą w efekcie prowadzić do wprowadzenia instrukcji warunkowych w kodzie (if)
- Mogą stanowić informację wejściową dla fabryk, które w zależności od konfiguracji tworzą adekwatny zestaw obiektów (w zależności od zastosowanego wzorca)

Kernel

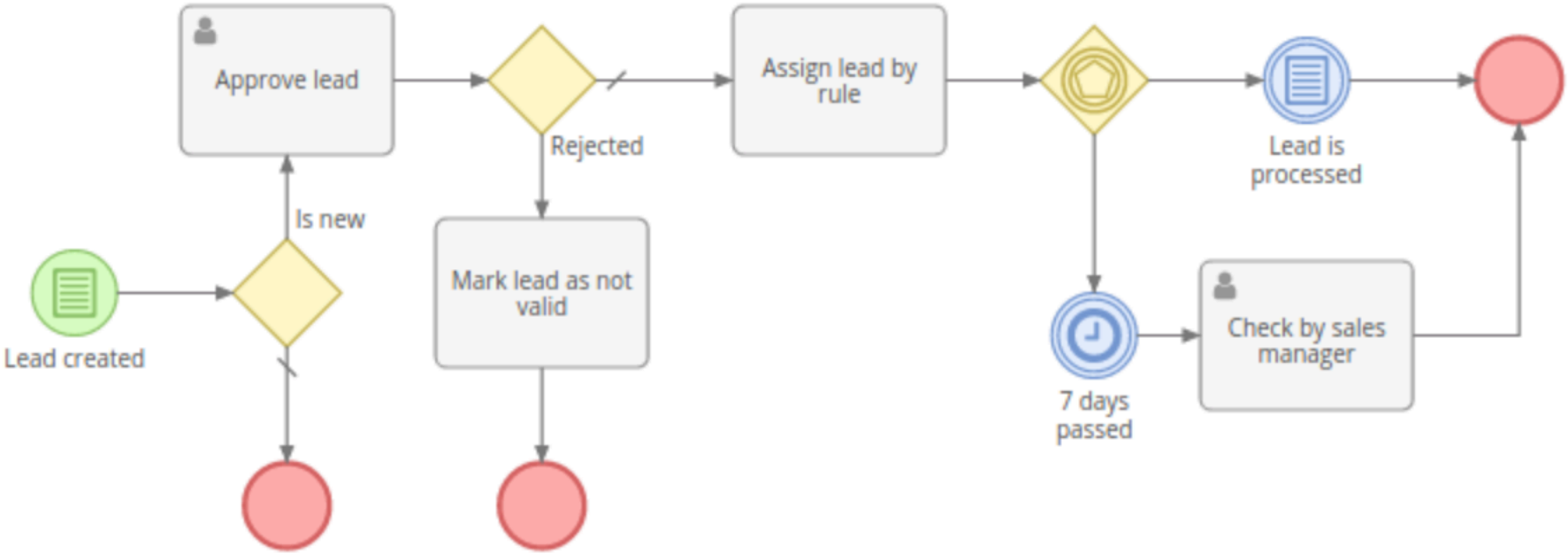
- Oddzielamy część wspólną systemu od pozostałych elementów
  - nawet na poziomie repozytorium kodu
- Umieszczamy tutaj wspólne funkcjonalności
- Kernel może zawierać wspólne klasy narzędziowe
- Kernel może zawierać wspólne elementy modelu

Dopasowanie wersje interfejsów użytkownika per klient

- Tworzymy jedną bazę kodu na poziomie logiki biznesowej oraz infrastruktury
- Rozdzielamy bazę kodu na poziomie UI
- UI dla różnych klientów jest w osobnych repozytoriach
- Uwaga
  - ryzykowne
  - jest sens rozważać gdy wersji jest mało 2-3

Dopasowanie procesów biznesowych za pomocą BPM

- Modelujemy proces biznesowy zewnętrznie, z użyciem narzędzia, który pozwala go zmodyfikować (np. workflowengine.io, dwkit)



Fasada per klient

- Na poziomie Application Service tworzymy wersje dla różnych klientów
- Dopasowujemy operacje i parametru, które są adekwatne dla danego typu klient
- Strategia ta też się dobrze sprawdza w przypadku różnicowania dla różnych klientów technicznych (web/mobile/desktop)

Komponentyzacja UI

- Interfejs jest kompozytem relatywnie niezależnych komponentów (paneli)
- Komponenty mogą komunikować się między sobą zdarzeniami lub z użyciem mediatora, aby uniknąć zależności
- W wariancie DDD/MSA – można dążyć do powiązania interfejsu z częścią domenową

Ewolucja architektury jako element procesu pracy

- Nie ma możliwości zaprojektowania niezmiennej architektury
- Istotna jest jej bezustanna analiza/proces, szczególnie pod kątem uwspólniania elementów dla różnych klientów
- **=> proces rozwoju architektury**