Projektowanie obiektowe oprogramowania
Wzorce architektury aplikacji (2)
Wykład 10 – Inversion of Control
Wiktor Zychla 2013

1 Dependency Injection

Dependency Injection = zestaw technik pozwalających tworzyć struktury klas o luźniejszym powiązaniu.

Trzy kluczowe skojarzenia:

1. **Późne wiązanie** – możliwość modyfikacji kodu bez rekompilacji, wyłącznie przez rekonfigurację, „*programming against interfaces*" (DIP)
2. **Ułatwienie tworzenia testów jednostkowych** – zastąpienie podsystemów przez ich stuby/fake'i
3. ***Uniwersalna fabryka***– tworzenie instancji dowolnych typów według zadanych wcześniej reguł

2 … więc przypomnijmy sobie przykład dla Dependency Inversion Principle

Zalety:

1. **Rozszerzalność** (OCP) – teoretycznie możliwe rozszerzenia o konteksty nie znane w czasie planowania
2. **Równoległa implementacja** – dobrze zdefiniowany kontrakt zależności pozwala rozwijać oba podsystemy niezależnie
3. **Konserwowalność (maintainability)** – dobrze zdefiniowana odpowiedzialność to zawsze łatwiejsza konserwacja
4. **Łatwość testowania** - obie klasy mogą być testowane niezależnie; ta z wstrzykiwaną zależnością może być testowana przez wstrzyknięcie stuba/fake'a
5. **Późne wiązanie** – możliwość określenia konkretnej klasy bez rekompilacji

3 Twarde zależności vs miękkie zależności

Jeszcze inne spojrzenie na modularność:

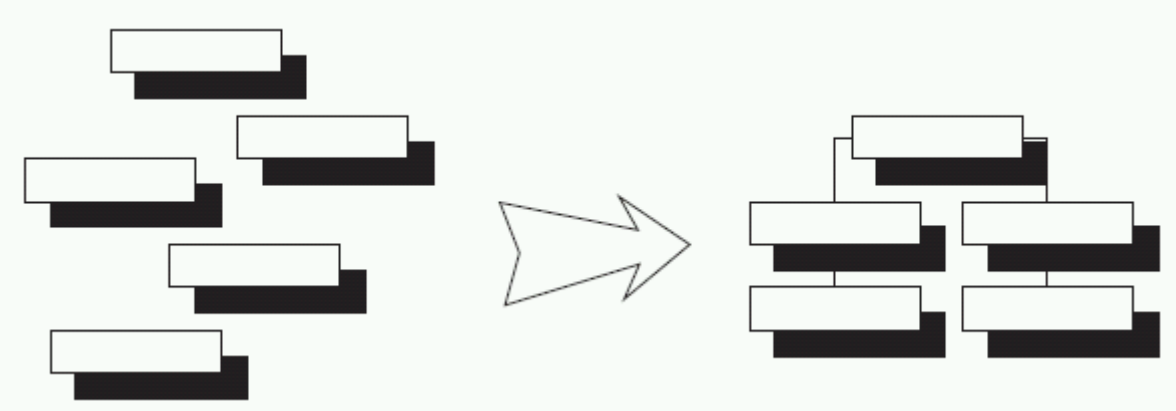
1. **Sztywna zależność** (stable dependency) – klasyczna modularność; zależne moduły już istnieją, są stabilne, znane i przewidywalne (np. biblioteka standardowa)

2. **Miękka zależność** (volatile dependency) – modularność dla której zachodzi któryś z powodów wprowadzenia spoiny:
    a. Konkretne środowisko może być konfigurowane dopiero w miejscu wdrożenia (późne wiązanie)
    b. Moduły powinny być rozwijane równolegle
3. **Spoina** (seam) – miejsce, w którym decydujemy się na zależność od interfejsu zamiast od konkretnej klasy

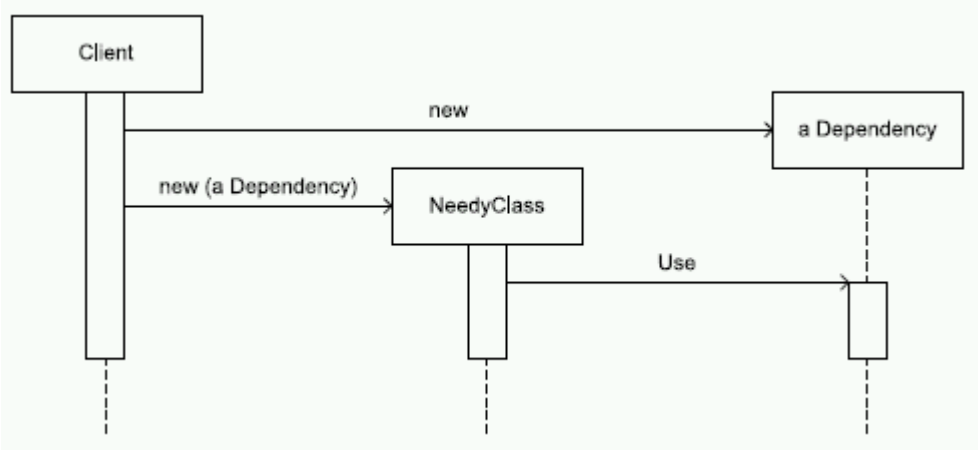
Uwaga. O ile zastosowanie technik DI pozwala na wprowadzenie miękkich zależności w miejscach spoin, o tyle zwykle zależności do samych ram DI mają charakter sztywny.

4 Kluczowe podwzorce DI (na przykładzie Unity)

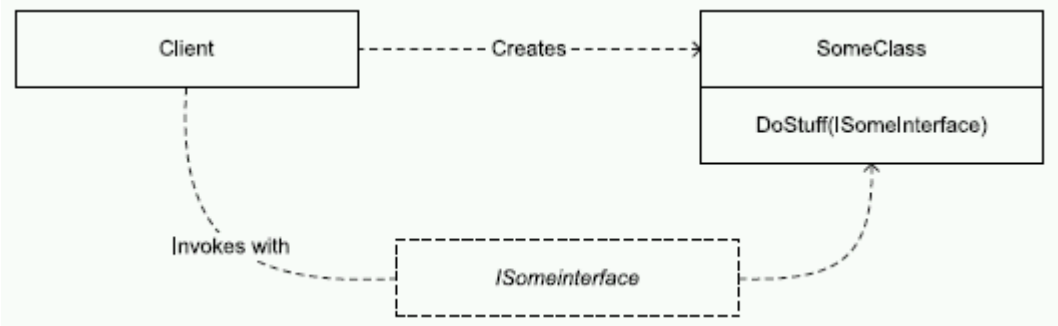
4.1 Składanie obiektów (Composition)



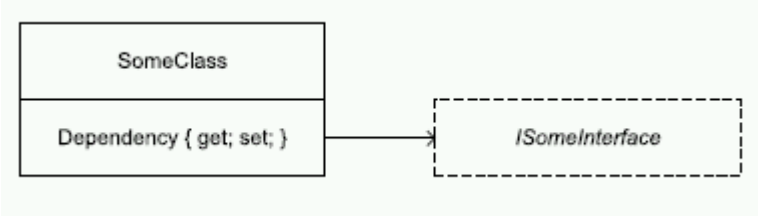
1. Kontener
2. Rozwiązywanie zależności sztywnych
3. Rozwiazywanie zależności miękkich
4. Rozwiązywanie instancji
5. Rozwiązywanie grafu zależności
6. Wstrzykiwanie przez konstruktor – najdłuższy lub wskazany ([InjectionConstructor]) – zapewnienie że zależność jest zawsze dostępna



7. Wstrzykiwanie przez metodę – atrybut [InjectionMethod] – zapewnienie że w różnych kontekstach (metodach) wstrzykiwane mogą być inne zależności

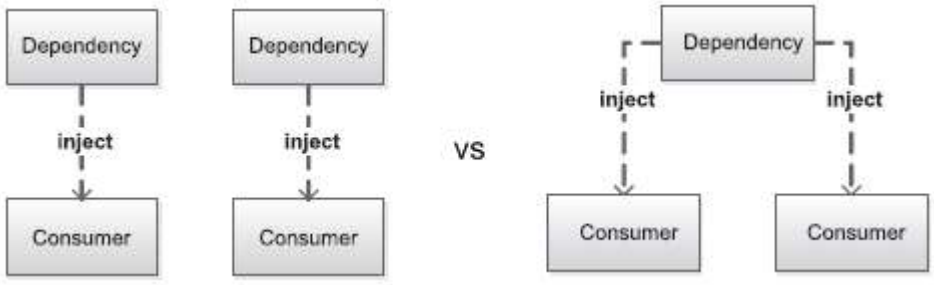


8. Wstrzykiwanie przez właściwości ([Dependency]) – zapewnienie że domyślna zależność jest dostępna, ale może być zmodyfikowana



9. „Budowanie” obiektu wyprodukowanego na zewnątrz
10. Rejestracja metody fabrykującej – zapewnienie możliwości tworzenia zależności przez dowolną metodę fabrykującą

4.2 Zarządzanie czasem życia obiektów (Lifecycle Management)

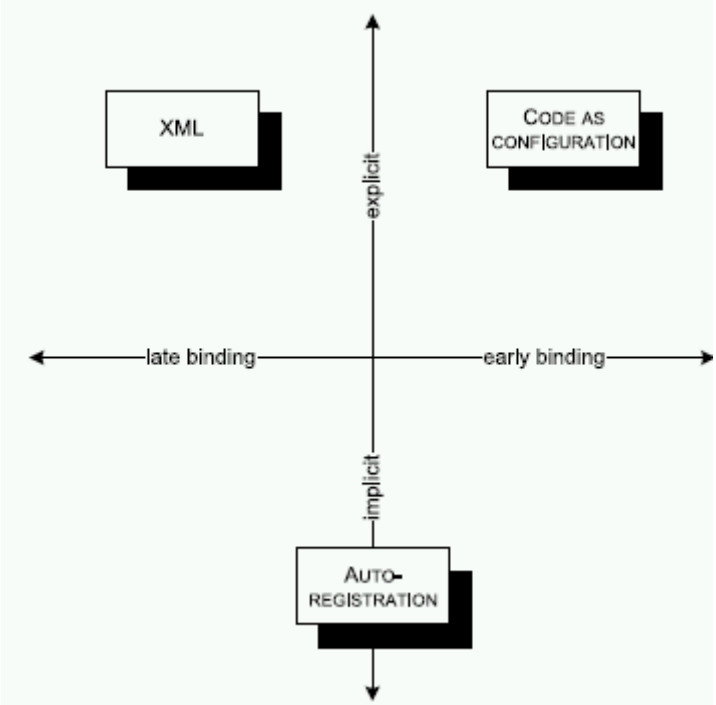


http://msdn.microsoft.com/en-us/library/ff660872(PandP.20).aspx

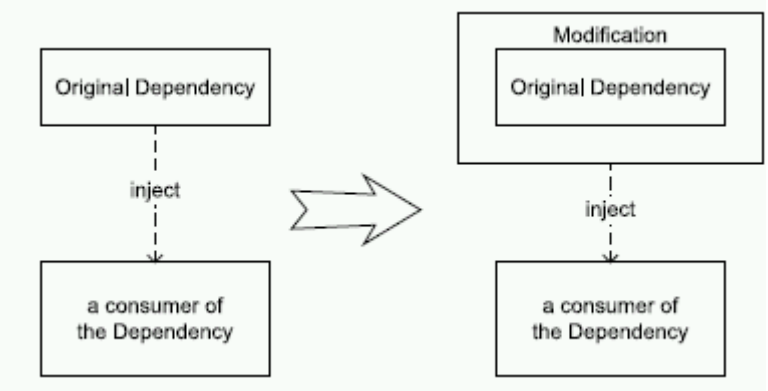
1. Transient – ulotne
2. ContainerControlled – singletony
3. Hierarchical – singletony, ale inne w dziedziczonych kontenerach

4. PerThread – inny obiekt per wątek
5. Custom (przykład)

4.3 Konfiguracja kontenera (Configuration)



4.4 Przechwytywanie żądań (Proxy)



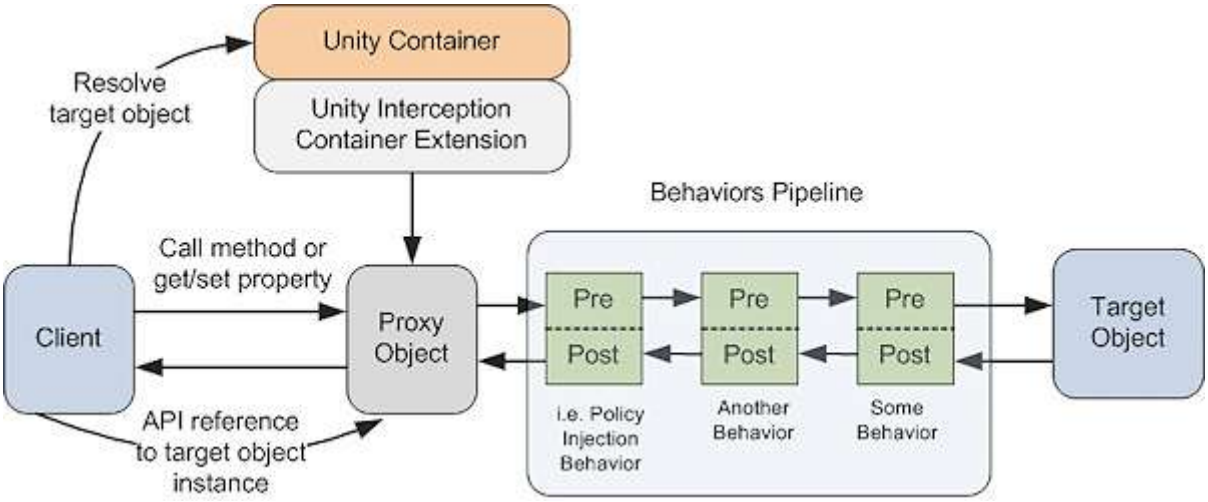
Typowe zagadnienia przekrojowe (cross-cutting concerns):

1. Audytowanie
2. Logowanie
3. Monitorowanie wydajności
4. Bezpieczeństwo
5. Cache'owanie
6. Obsługa błędów

Frameworki DI często obsługują podzbiory tak zdefiniowanego AOP dzięki temu że zamiast obiektu mogą zwracać proxy do niego. Przykład w Unity:

1. **InterfaceInterceptor** – tworzy proxy przez delegowanie, pozwala przechwycić tylko metody interfejsu

2. **VirtualMethodInterceptor** – tworzy proxy przez dziedziczenie, pozwala przechwycić tylko metody wirtualne



5 ServiceLocator vs Composition Root+Factory

Jak w rozbudowanej, wielomodułowej aplikacji radzić sobie z rozwiązywaniem zależności do usług?

Żeby rozwiązać zależność potrzebny jest kontener. Innymi słowy, w kodzie, w miejscu w którym potrzebujemy instancji usługi, potrzebny jest kontener.

Najgorsze rozwiązanie – przekazywać kontener jako parametr do klas/metod.

Trochę lepsze rozwiązanie – Service Locator.

Service Locator = schowanie singletona kontenera DI za fasadą, pozwalającą z dowolnego miejsca aplikacji na rozwiązanie zależności do usługi. Pozwala znacznie zredukować jawne zależności miedzy klasami. Service Locator nie musi być przekazywany jako zależność, bo jako singleton, może być osiągalny z dowolnego miejsca.

Uwaga! Service Locator uważa się za antywzorzec z uwagi na dwa niepożądane zjawiska:

1. Service Locator powoduje owszem zredukowanie zależności między klasami, ale kosztem wprowadzenia zależności do podsystemu DI. To bardzo nieeleganckie. Zastosowanie DI powinno być **przezroczyste** dla kodu – struktura klas powinna być taka sama bez względu na to czy wspomagamy się ramą DI czy nie.
2. Zależności rozwiązywane przez SL są niejawne – rozwiązywanie pojawia się w implementacji. Na poziomie struktury (metadanych) nie ma jawnej informacji że klasa A zależy od B – A sobie samo wykonstruuje B za pomocą SL kiedy jest mu to potrzebne. Problem w tym, że ponieważ tej zależności nie widać na poziomie struktury, może być trudna do wychwycenia i przez to powodować **błędy w czasie wykonania programu** (wtedy, gdy zapomni się zarejestrować implementację B w kontenerze).

Alternatywą dla SL jest **Compositon Root.**

Composition Root = fragment kodu wykonywany zwykle na starcie aplikacji, odpowiedzialny za zdefiniowanie wszystkich zależności. W idealnej rzeczywistości, tylko w Composition Root pojawia się zależność do DI, a cała reszta aplikacji jest jej pozbawiona.

Sam CR jest zbyt słaby żeby rozwiązać problem rozwiązywania zależności. Naiwne zastosowanie spowodowałoby konieczność wytworzenia **wszystkich instancji** obiektów ze wstrzykiwanymi zależnościami już na starcie aplikacji. To oczywiście niemożliwe.

W praktyce CR należy wesprzeć „miękką fabryką" – fabryką z miękką zależnością do implementacji dostawcy obiektów.

W CR tworzy się dostawcę wykorzystującego kontener, klasa kiedy potrzebuje usługi używa fabryki.

Takie podejście eliminuje całkowicie problem „przezroczystości" ramy DI dla kodu, można sobie bowiem wyobrazić dostawcę usługi, który używa DI i innego dostawcę który nie używa DI.

6 Literatura uzupełniająca

1. Dhanji R. Prasanna – *Dependency Injection* (2009, Java)
2. Mark Seemann – *Dependency Injection in .NET* (2012, C#) (źródło ilustracji i planu prezentacji)