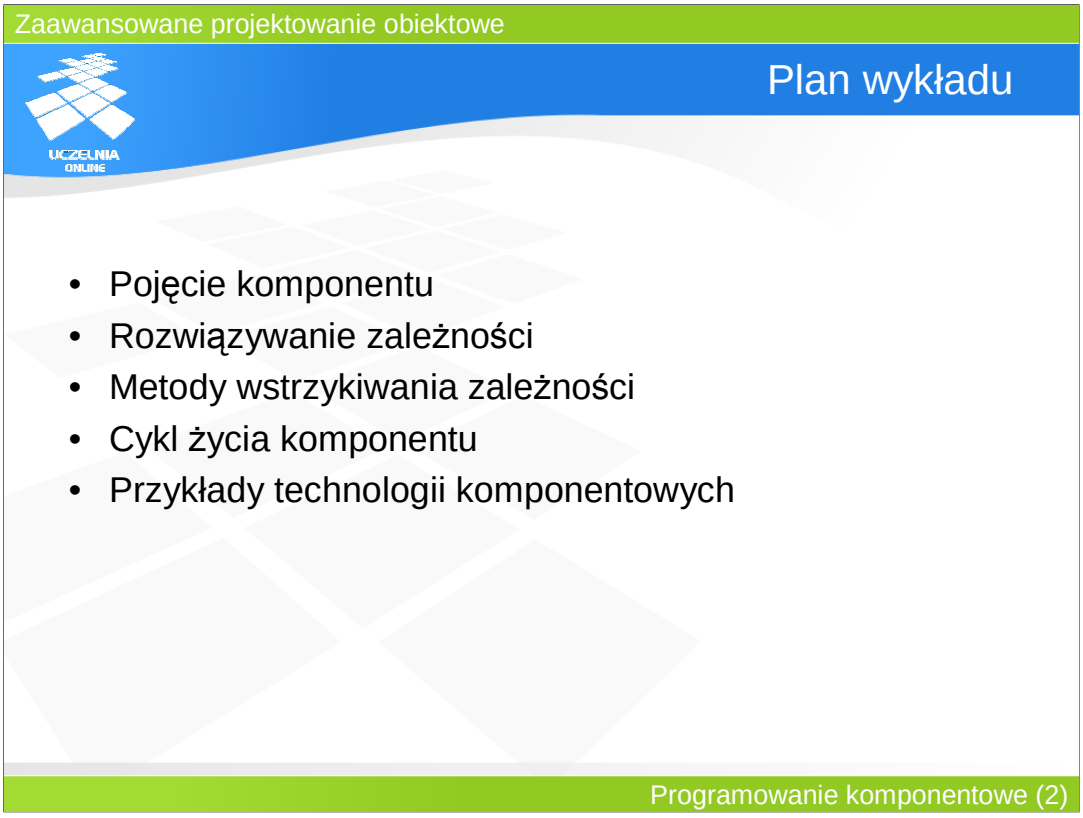
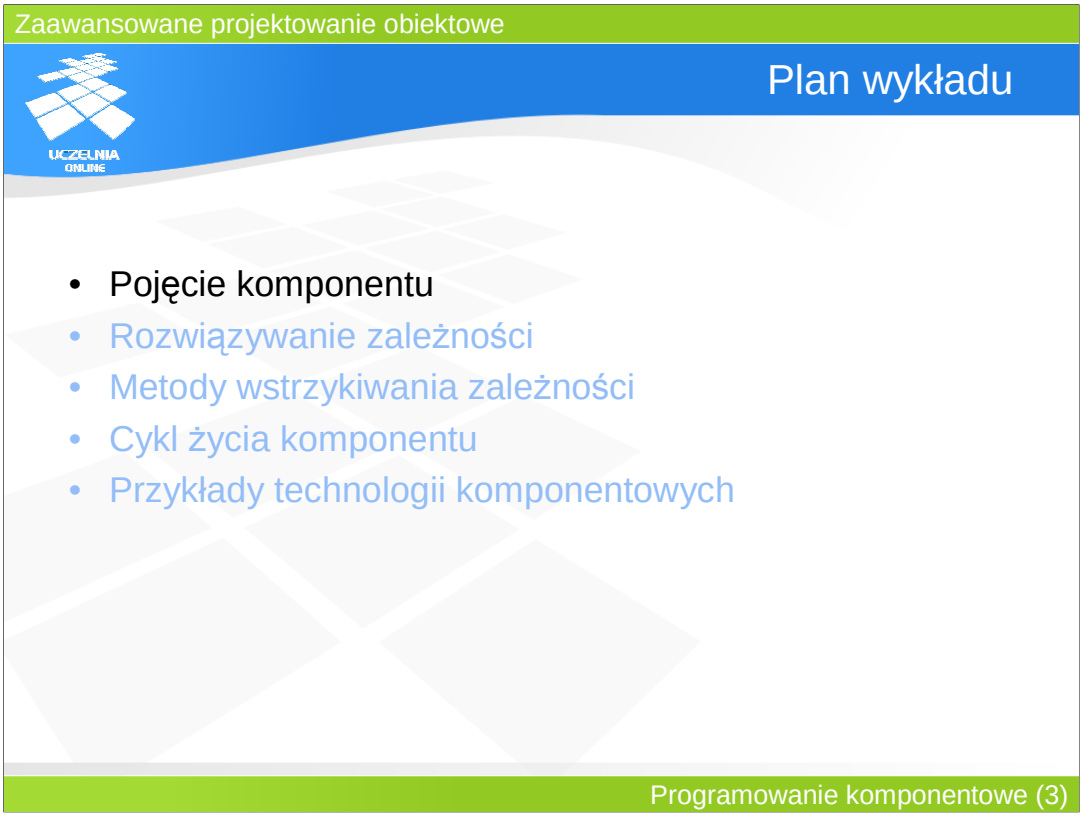

Zaawansowane projektowanie obiektowe
Programowanie komponentowe
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE



Wykład jest ostatnim z serii dotyczącej różnych aspektów projektowania obiektowego. Jest poświęcony programowaniu komponentowemu – technice pozwalającej w większym stopniu niż w przypadku obiektowości wykorzystać jej atrybuty: hermetyzację, abstrakcję i powtórne użycie kodu.

Wykład rozpocznie się od zdefiniowania pojęcia komponentu, jego cech wyróżniających go od obiektu oraz możliwości, jakie stwarza jego użycie. Następnie przedstawiony zostanie problem rozwiązywania zależności pomiędzy komponentami, co umożliwia ich współpracę. W szczególności przedstawiony będzie typowy mechanizm wyszukiwania zależności oraz, w opozycji do niego, wzorzec odwróconego sterowania. Pozwala on na definiowanie zależności w bardziej deklaratywny sposób, co ogranicza liczbę tych zależności pomiędzy komponentami oraz zwiększa poziom abstrakcji. Kolejnym elementem będzie cykl życia komponentu i jego sposób reprezentacji w istniejących rozwiązaniach. Ostatnia część wykładu będzie poświęcona przeglądowi wybranych technologii komponentowych, z uwzględnieniem oferowanych przez nie możliwości w zakresie definiowania komponentów i rozwiązywania zależności pomiędzy nimi.



Pierwsza część wykładu jest poświęcona wprowadzeniu do programowania komponentowego, określeniu pojęcia komponentu i cech, które wyróżniają go od zwykłego obiektu.

Komponent

Komponent jest

- podstawową jednostką oprogramowania
- z kontraktowo (deklaratywnie) opisanymi interfejsami, i
- podanymi wprost zależnościami.

Komponent może zostać skonfigurowany i wdrożony niezależnie od programisty, który go stworzył.

Clemens Szyperski
"Oprogramowanie komponentowe. Obiekty to za mało" (1998)

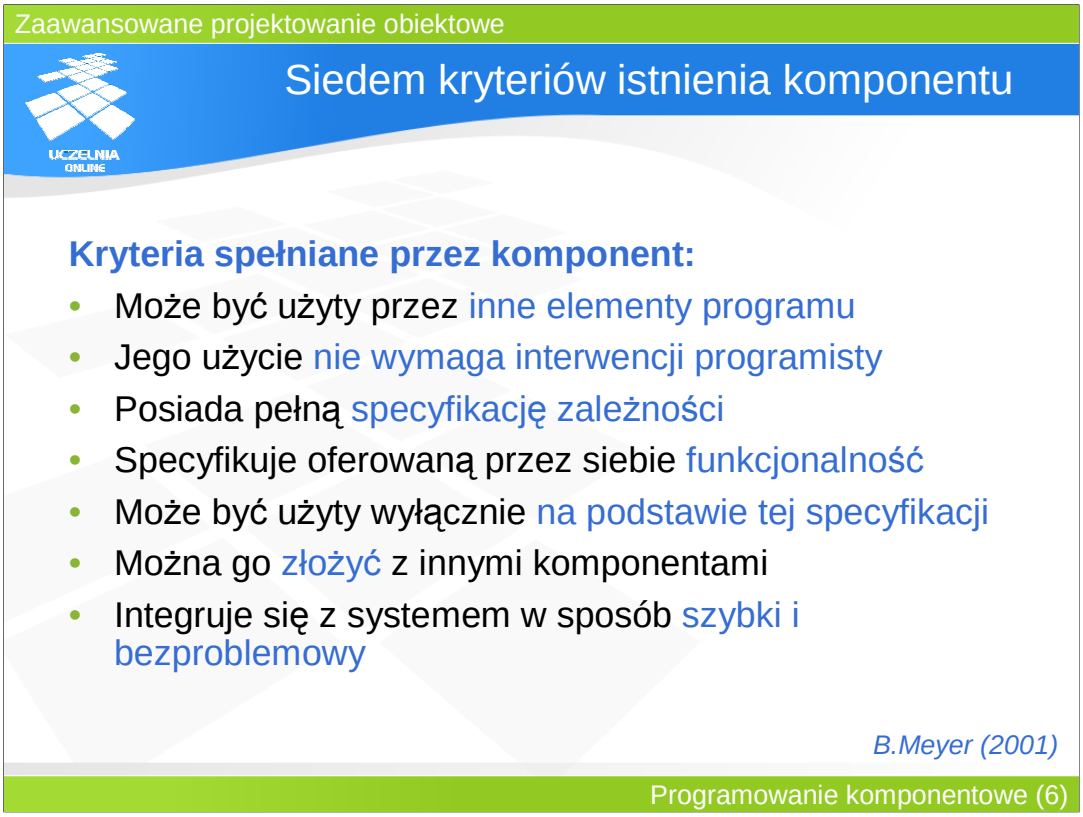
Definicja komponentu podana przez C. Szyperskiego brzmi następująco. Wg niego, komponent jest podstawowym elementem, z którego budowane jest oprogramowanie, posiada opisane deklaratywnie interfejsy, a wszystkie wymagane przez niego zależności są podane wprost. Dzięki temu możliwe jest przekazanie go do wdrożenia osobie, która nie jest jego twórcą.

Komponenty na poziomie implementacyjnym są zwykle rozszerzeniem obiektów (choć istnieją także technologie komponentowe zbudowane w oparciu o języki strukturalne), dlatego komponent jest zbudowany z jednej lub kilku klas. Warto zwrócić uwagę na dwie cechy wyróżnione przez Szyperskiego: deklaracje oferowanych interfejsów i żądanych zależności. Dzięki nim komponent może być traktowany jako zamknięta całość, która oferuje usługi i żąda ich od zewnętrznych komponentów, jednocześnie ukrywając swoją strukturę wewnętrzną i nie wnikając w budowę innych. To czyni z komponentu jednostkę bardziej abstrakcyjną niż obiekt.

Na czym polega zatem różnica między komponentem a obiektem? Dobrze zaprojektowany obiekt (lub ich grupa) w wielu przypadkach może być użyty jako komponent. Jednak stopień abstrakcji i hermetyzacji obiektu zależy w dużej mierze od projektanta. Błędy popełnione przy projektowaniu obiektów można do pewnego stopnia zamaskować, zwiększając liczbę powiązań między nimi, obniżając ich spójność etc. W przypadku komponentu jest to niemożliwe z uwagi na abstrakcyjne interfejsy, poprzez które komponenty się komunikują ze sobą.

Różnice dotyczą przede wszystkim wymuszonej (a nie jedynie postulowanej) hermetyzacji komponentów oraz nacisku na definiowanie abstrakcyjnych interfejsów, które określają sposób komunikacji. Szeroko stosowane w świecie obiektów mechanizmy dziedziczenia są wykorzystywane przez komponenty jedynie w niewielkim stopniu, przede wszystkim w odniesieniu do interfejsów.

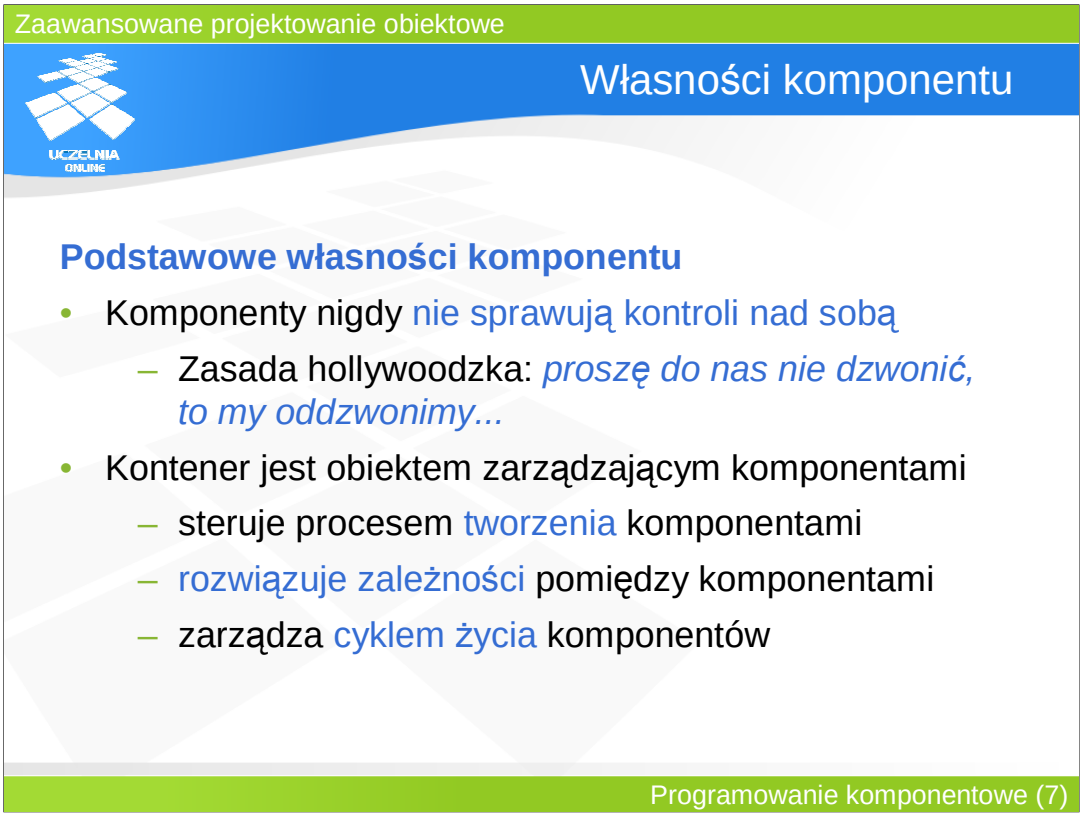
Podsumowując: programowanie komponentowe nie posiada zupełnie nowych atrybutów w stosunku do programowania obiektowego, a jedynie konsekwentnie i w pełni wykorzystuje cechy obiektowości.



Aby stwierdzić, który obiekt może pełnić rolę komponentu, Bertand Meyer podał siedem kryteriów, jakie musi on spełniać.

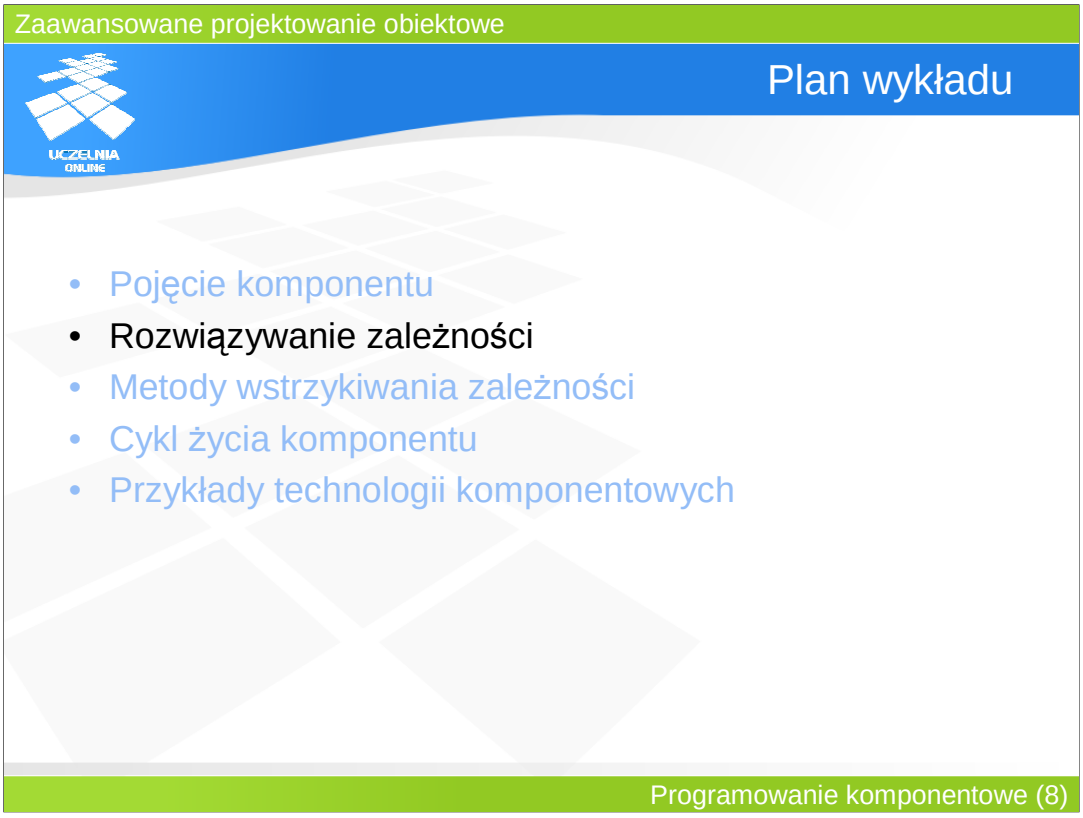
Podstawowym warunkiem jest możliwość użycia go przez inne elementy programu (przede wszystkim inne komponenty), przy czym wykorzystanie komponentu nie wymaga modyfikacji jego kodu źródłowego. Aby możliwe było przekazanie mu z zewnątrz wymaganych zależności, muszą one być podane w pełni i jawnie. Z drugiej strony komponent podobnie specyfikuje funkcje oferowane przez siebie, a specyfikacja ta jest wystarczającą podstawą do jego wykorzystania. Integracja komponentu z systemem powinna przebiegać bez nadmiernego nakładu pracy, np. wyłącznie poprzez jego konfigurację.

Kryteria te w zasadzie pokrywają się z definicją podaną przez Szyperskiego.

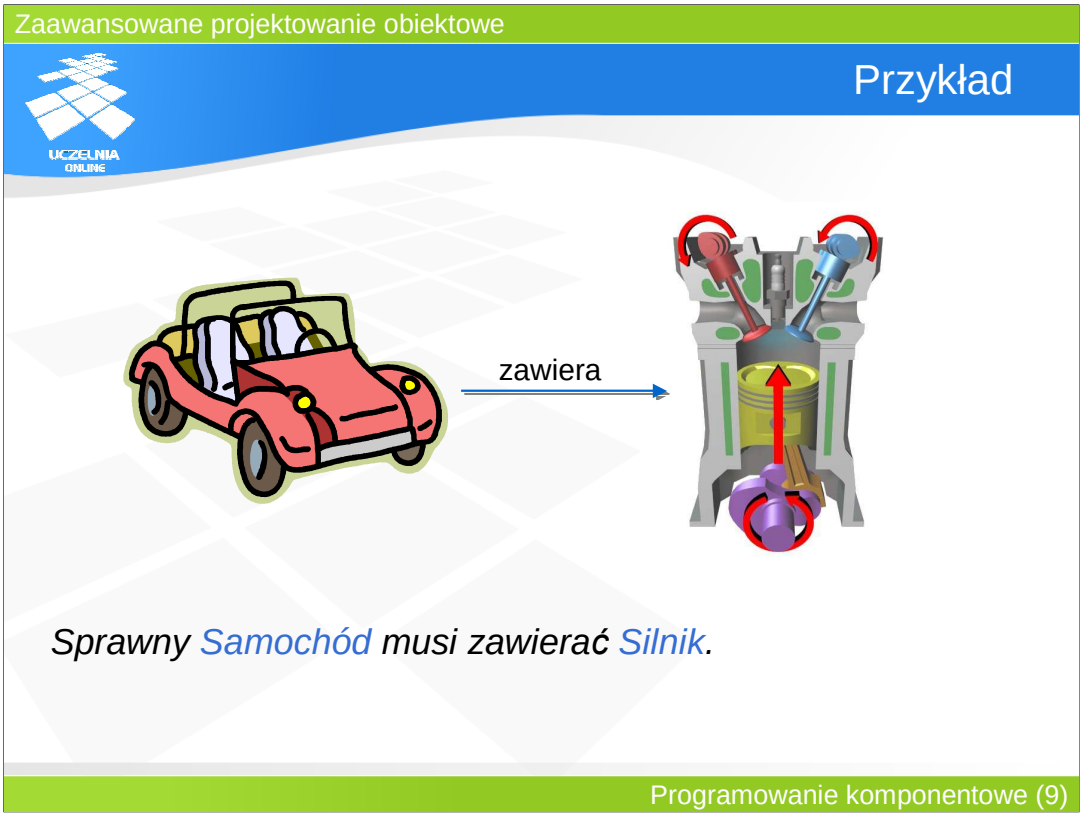


Komponenty, w odróżnieniu od obiektów, nie są jednostkami aktywnymi. Nie kontrolują więc swojego życia, nie tworzą samodzielnie swoich instancji.

Zarządzaniem nimi zajmuje się specjalizowany obiekt – kontener, który kontroluje niemal każdy aspekt ich życia: tworzeniem (lub odzyskiwaniem) instancji komponentu, rozwiązywaniem zależności pomiędzy nimi oraz zarządzaniem cyklem ich życia. W niektórych zastosowaniach (np. EJB) zarządza również niezbędnymi zasobami, zapewnia własności pozafunkcjonalne (np. transakcyjność) i równoważy obciążenie nadchodzących żądań ze strony klientów. Kontener w programowaniu komponentowym pozwala na ograniczenie zadań stojących przed komponentami jedynie do realizacji funkcji wynikających z przypisanej im odpowiedzialności "biznesowej", natomiast aspektami technicznymi zajmuje się sam.



Druga część wykładu dotyczy interakcji pomiędzy komponentami i sposobów określania i rozwiązywania zależności, jakie między nimi występują.



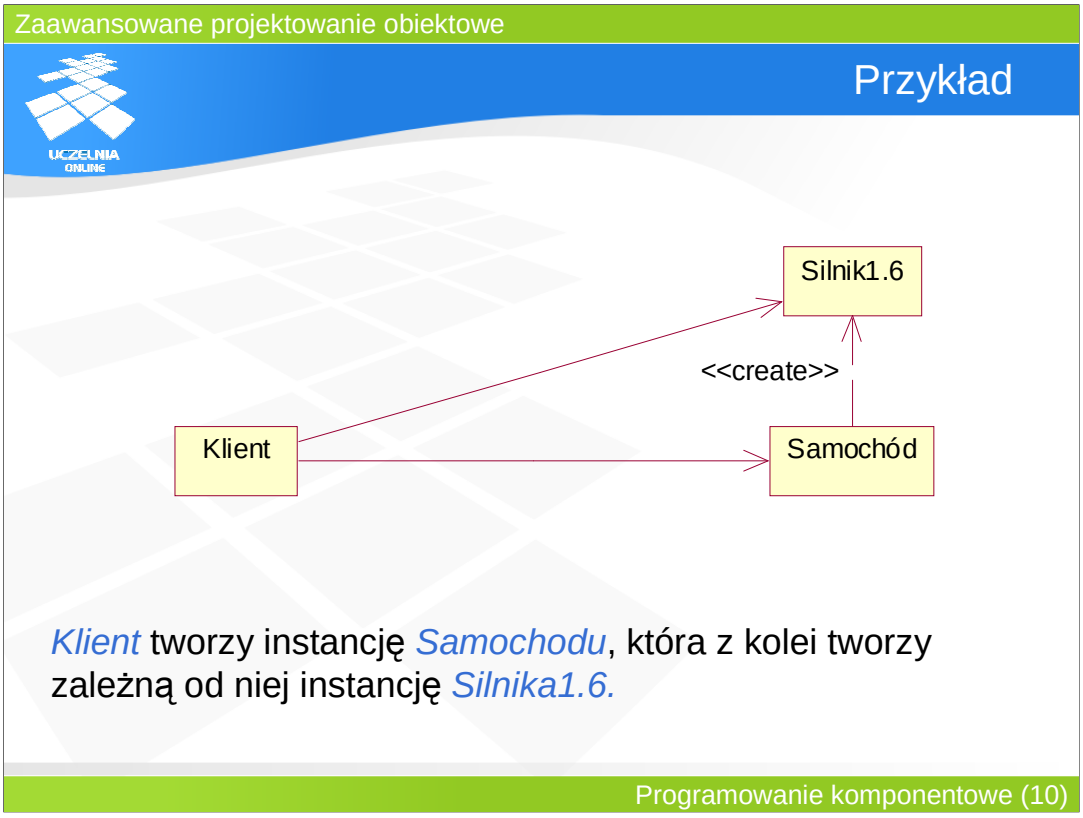
Komponenty nie istnieją w odosobnieniu – aby wykonywać swoje funkcje, muszą ze sobą współpracować. Istotą programowania komponentów jest możliwość ich konfigurowania i łączenia bez konieczności modyfikacji kodu źródłowego, dlatego istotnym problemem jest definiowanie zależności pomiędzy nimi, a następnie ich rozwiązywanie. Zadania te muszą być rozwiązane w sposób zapewniający spełnienie podanych wcześniej warunków.

Problem zależności pomiędzy komponentami zostanie omówiony na prostym przykładzie.

Samochód jest zbudowany z wielu podzespołów, które są także powiązane pomiędzy sobą, w tym m.in. Silnika. Z punktu widzenia Klienta jednak Samochód tworzy zamkniętą całość, której wewnętrzna struktura i sposób współpracy poszczególnych jej elementów mają niewielkie znaczenie: kupuje Samochód w całości, gotowy do drogi, i oczekuje, że będzie mógł nim wyjechać z salonu, a nie rozpoczynać jego użytkowanie od samodzielnego złożenia go z części

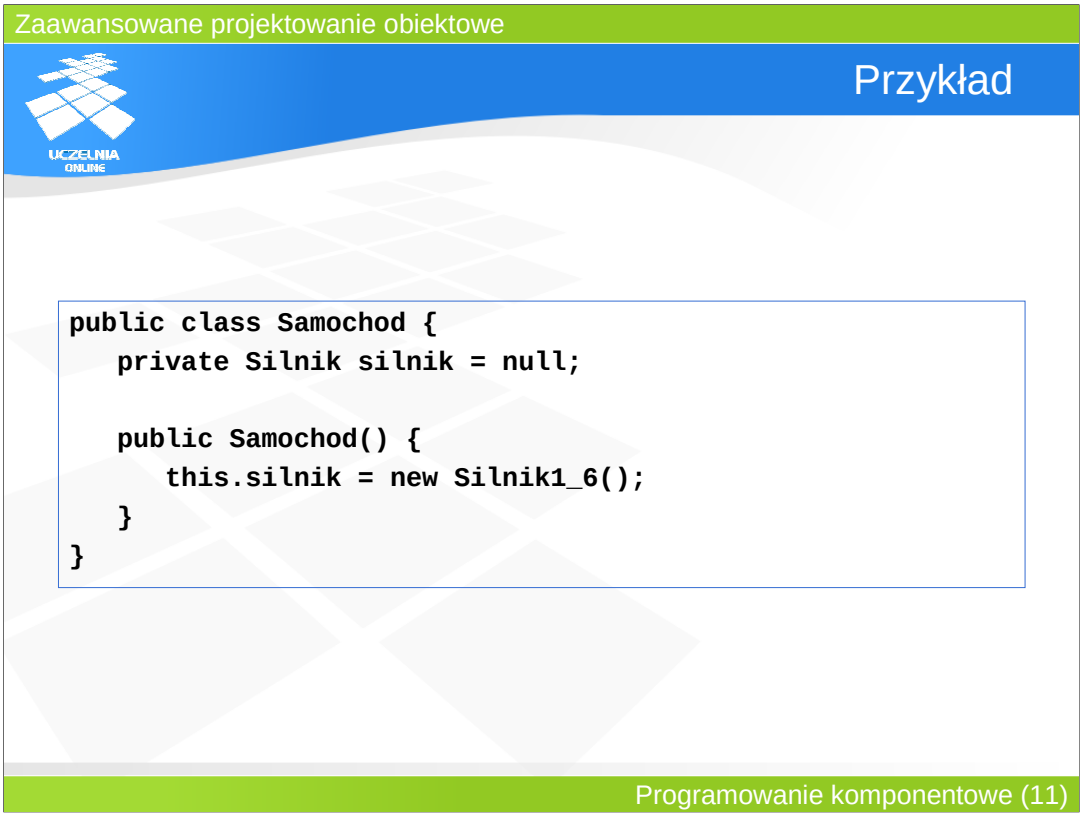
Z drugiej strony klient chce mieć wpływ na dobór przynajmniej niektórych podzespołów, które zostaną zamontowane w jego egzemplarzu (m.in. rodzaj silnika, typ opon, elementy bezpieczeństwa, dodatkowe wyposażenie etc).

Pogodzenie tych dwóch wymagań jest w niektórych sytuacjach trudne i jest istotą problemu definiowania zależności pomiędzy komponentami.

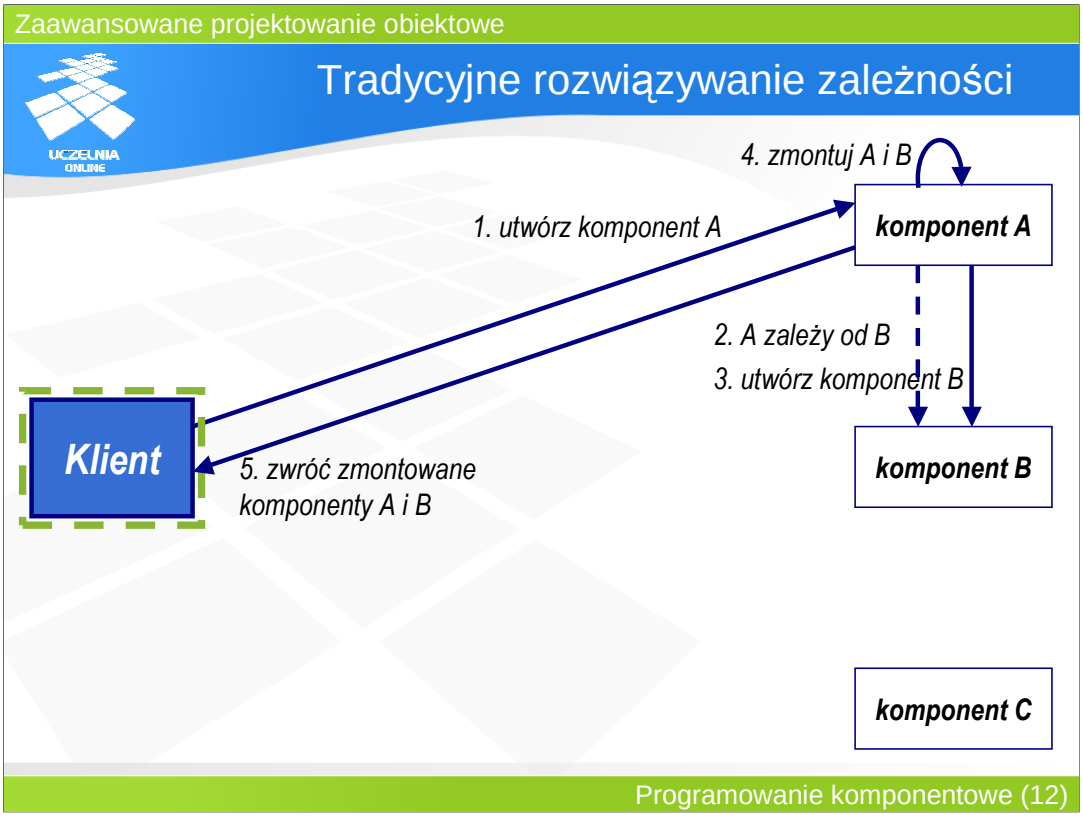


Typowe rozwiązanie przedstawionego problemu, które nie spełnia warunków stawianych komponentom, polega na hermetyzacji procesu tworzenia komponentu zależnego (jakim jest Silnik) przez Samochód. Innymi słowy, to Samochód decyduje o wyborze poszczególnych komponentów, natomiast Klient nie ma na to żadnego wpływu. W zamian otrzymuje prawidłowo zmontowany pojazd.

O ile z obiektowego punktu widzenia może to być rozwiązanie akceptowalne (Klient jest odseparowany od szczegółów tworzenia Samochodu), to z punktu widzenia komponentów – już nie. Rozwiązanie to tworzy bardzo silną zależność pomiędzy Samochodem i Silnikiem, na którą Klient nie ma żadnego wpływu: nie może np. zmienić rodzaju Silnika bez modyfikacji klasy Samochód. Zależność ta definiuje nie tylko kontrakt dotyczący cech Silnika (czyli interfejs), ale narzuca też instancję konkretnego modelu Silnika.



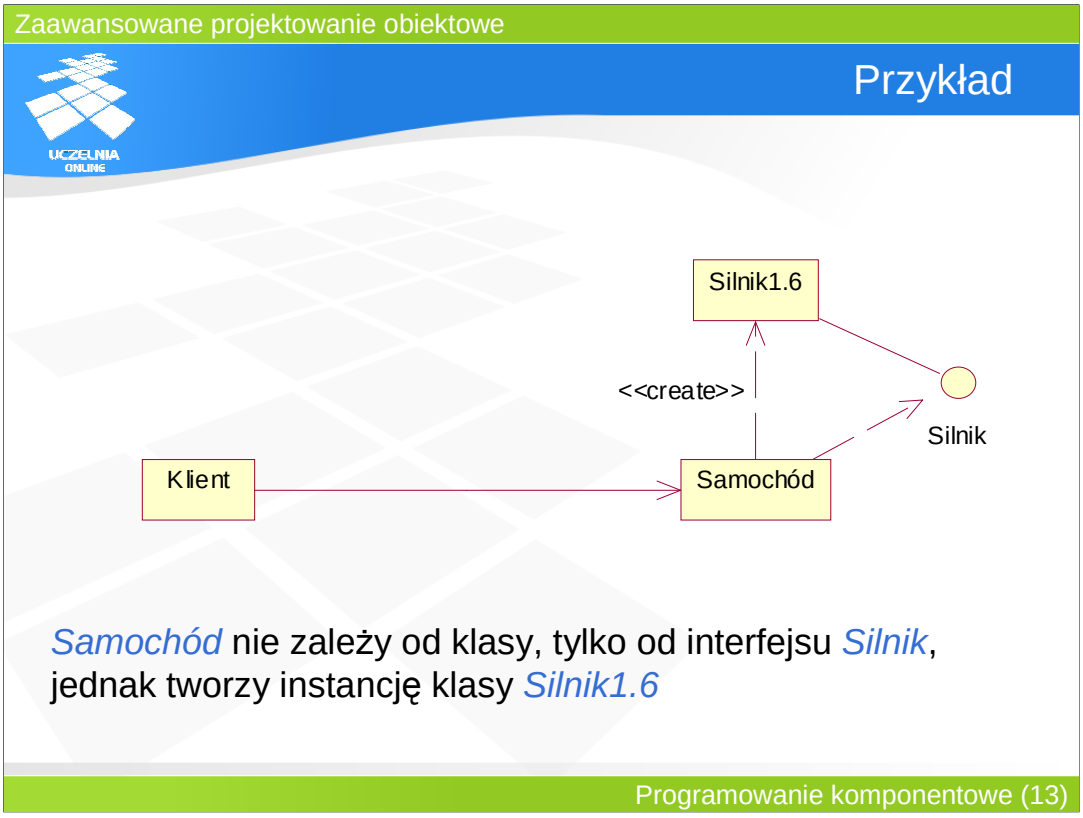
Slajd przedstawia przykładowy kod realizujący opisane wcześniej powiązanie pomiędzy Samochodem a Silnikiem. Samochód wewnętrznie tworzy instancję obiektu Silnik1_6, co wprawdzie pozwala odseparować Klienta od tworzenia Silnika, jednak jednocześnie pozbawia Klienta wpływu na wybór Silnika. Za montaż samochodu odpowiedzialny jest zatem obiekt Samochód, pozostający poza bezpośrednią kontrolą Klienta.



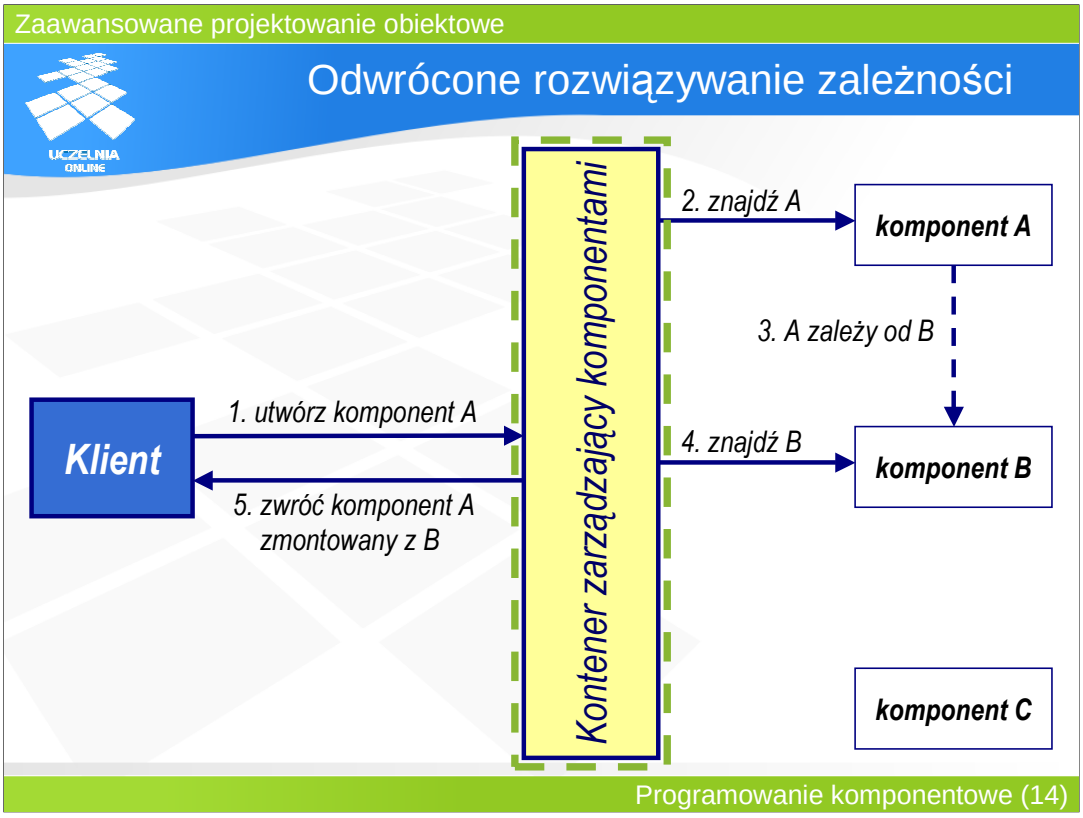
Taki uogólniony proces tworzenia obiektu z obiektem zależnym przebiega w następujących krokach:

1. Klient żąda utworzenia instancji komponentu A
2. Komponent A posiada zależność od komponentu B, dlatego nie może być utworzony przed nim
3. Dlatego komponent A musi utworzyć instancję komponentu B
4. oraz zmontować je razem
5. Ostatnim krokiem jest przekazanie zmontowanego zestawu klientowi

W efekcie klient otrzymuje żądaną instancję komponentu A, jednak nie ma żadnego wpływu na sposób montażu A i B ani na wybór zależnego komponentu B.



Aby osłabić rodzaj zależności pomiędzy komponentami, można wprowadzić pomiędzy nie interfejs. W tym przypadku komponent Samochód nie zależy bezpośrednio od klasy Silnik1_6, ale od implementowanego przez nią interfejsu Silnik. W praktyce jednak musi także utworzyć instancję klasy Silnik1_6, zatem po wprowadzeniu interfejsu Samochód nadal w pewien sposób będzie zależał od konkretnej klasy Silnik1_6, a ponadto powstanie zależność od interfejsu, który ta klasa implementuje. Zatem takie rozwiązanie nadal nie spełnia swojego zadania.



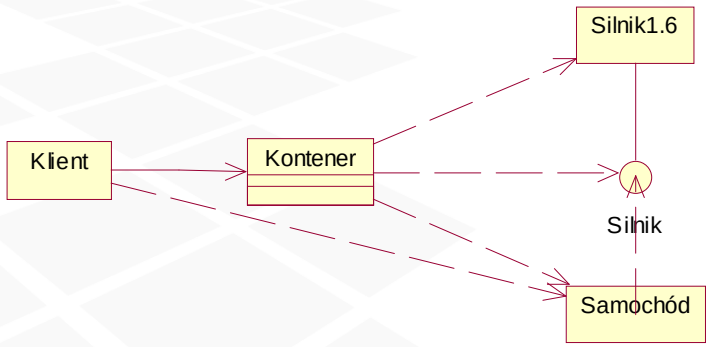
Rozwiązaniem zdecydowanie zmieniającym układ zależności jest wprowadzenie odwróconego sterowania (ang. *Inversion of Control,* IOC). Polega ono na innym przydziale odpowiedzialności do poszczególnych komponentów.

Komponenty stają się pasywne, tj. nie tworzą samodzielnie innych komponentów, tylko je odbierają od innego obiektu. W ten sposób są pozbawione wpływu na sposób zarządzania ich instancjami.

Ponadto, dominującą rolę przejmuje nowy uczestnik interakcji – kontener, nazywany często kontenerem IoC. Jego rola polega na zarządzaniu tworzeniem wszystkich komponentów oraz dostarczaniu gotowych instancji klientowi. Komponenty, chcąc rozwiązać swoje zależności, zlecają tę czynność kontenerowi, który zna wszystkie komponenty (przechowuje ich konfigurację w rejestrze) i może dostarczyć ich instancje.

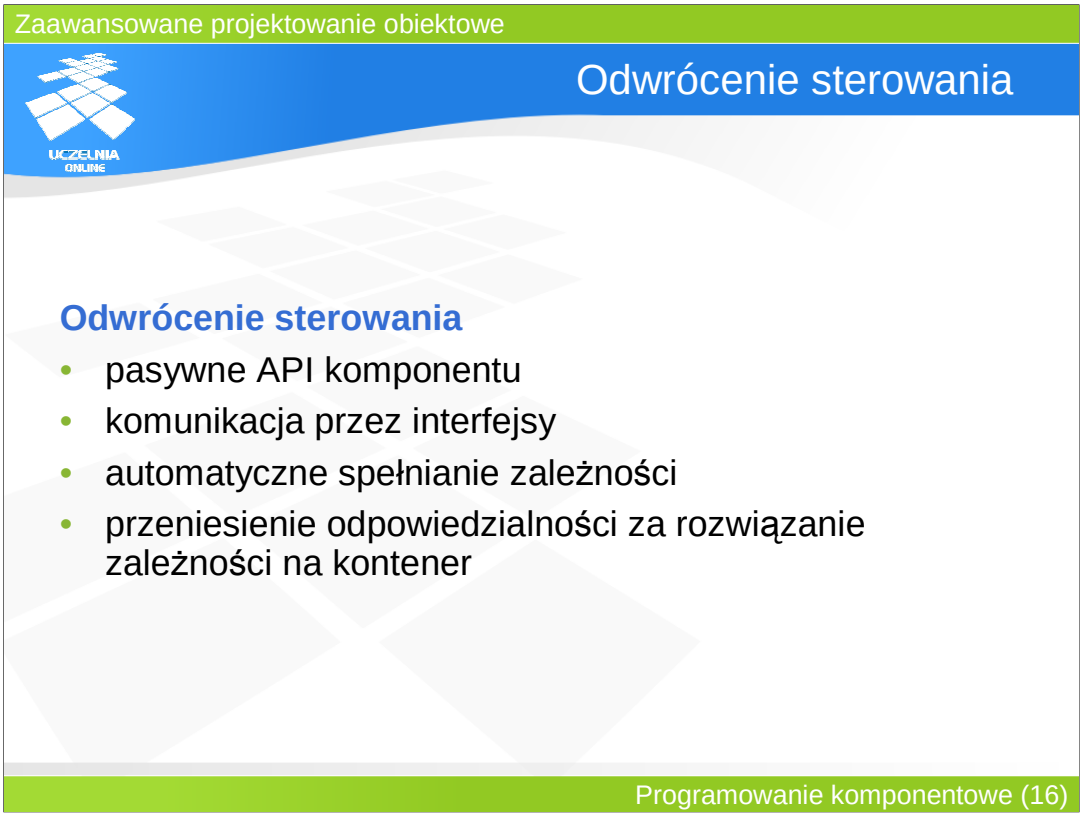
W tym układzie rola klienta nie ulega zmianie, jednak możliwość konfiguracji i wpływu na zachowanie komponentów jest znacznie poprawiona. Zamiast bezpośrednich powiązań pomiędzy implementacjami komponentów, zależą one od kontenera, którego konfiguracja decyduje o utworzeniu ich instancji.

Przykład



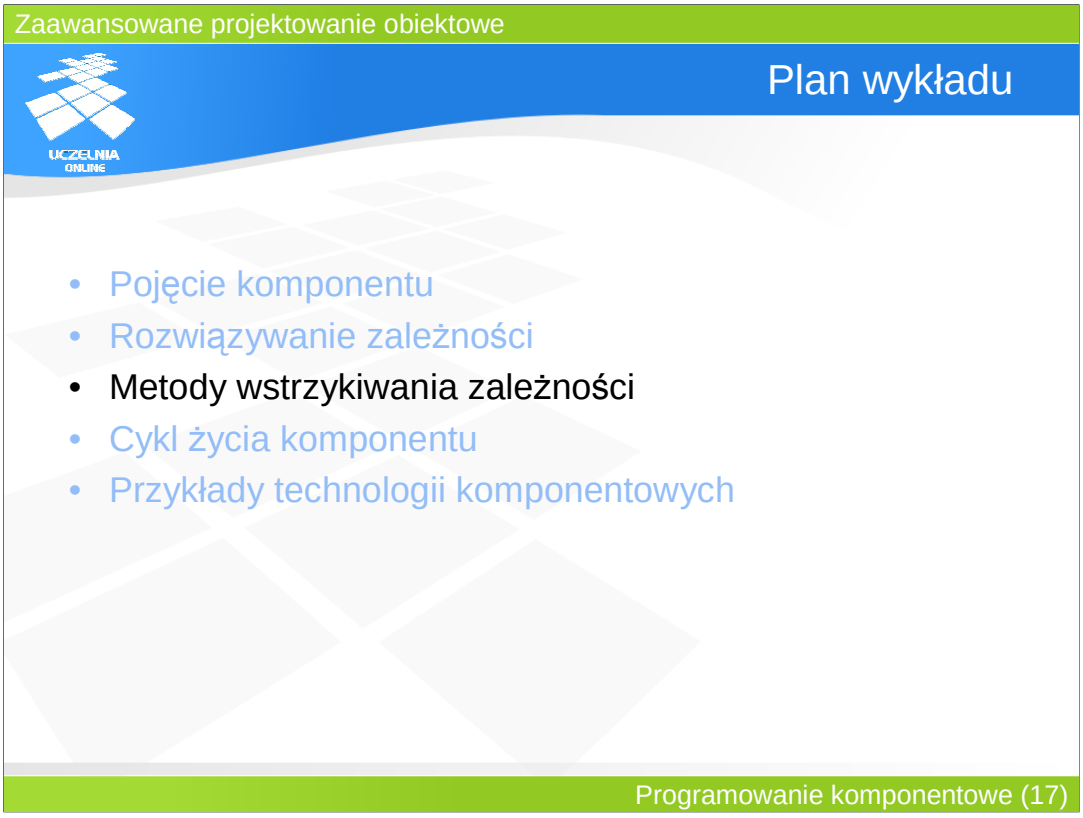
Klient odwołuje się do *Kontenera* w celu uzyskania zmontowanej instancji *Samochodu*. Zależności wewnętrzne są ukryte przed *Klientem*.

Po wprowadzeniu do omawianego przykładu kontenera powiązania między komponentami uległy osłabieniu. Zależności polegające na tworzeniu instancji obiektu zostały usunięte z warstwy komponentów i przeniesione na styk kontenera i komponentów. Jedyne zależności, jakie mogą wystąpić w warstwie komponentów, to żądane lub oferowane przez nie interfejsy, dzięki czemu komponenty stają się bardziej deklaratywne. Rozwiązywaniem zależności i tworzeniem instancji komponentów zajmuje się jedynie kontener, który jest także dla klienta rodzajem interfejsu umożliwiającego dostęp do komponentów.



Na czym zatem polega odwrócenie sterowania (ang. *Inversion of Control, IoC*)? Na przesunięciu odpowiedzialności za tworzenie i konfigurowanie komponentów z nich samych na kontener oraz usunięciu mocnych powiązań z warstwy komponentów. W efekcie program staje się bardziej abstrakcyjny i deklaratywny, co poprawia jego pielęgnowalność i testowalność.

W systemie z odwróconym sterowaniem komponenty nigdy nie sterują same sobą. Jedynym sposobem komunikacji pomiędzy nimi jest deklarowanie wymaganych zależności, które są rozwiązywane i spełniane przez kontener. Posiada on pełną władzę nad komponentami, także w zakresie tworzenia ich instancji oraz zarządzania cyklem ich życia.

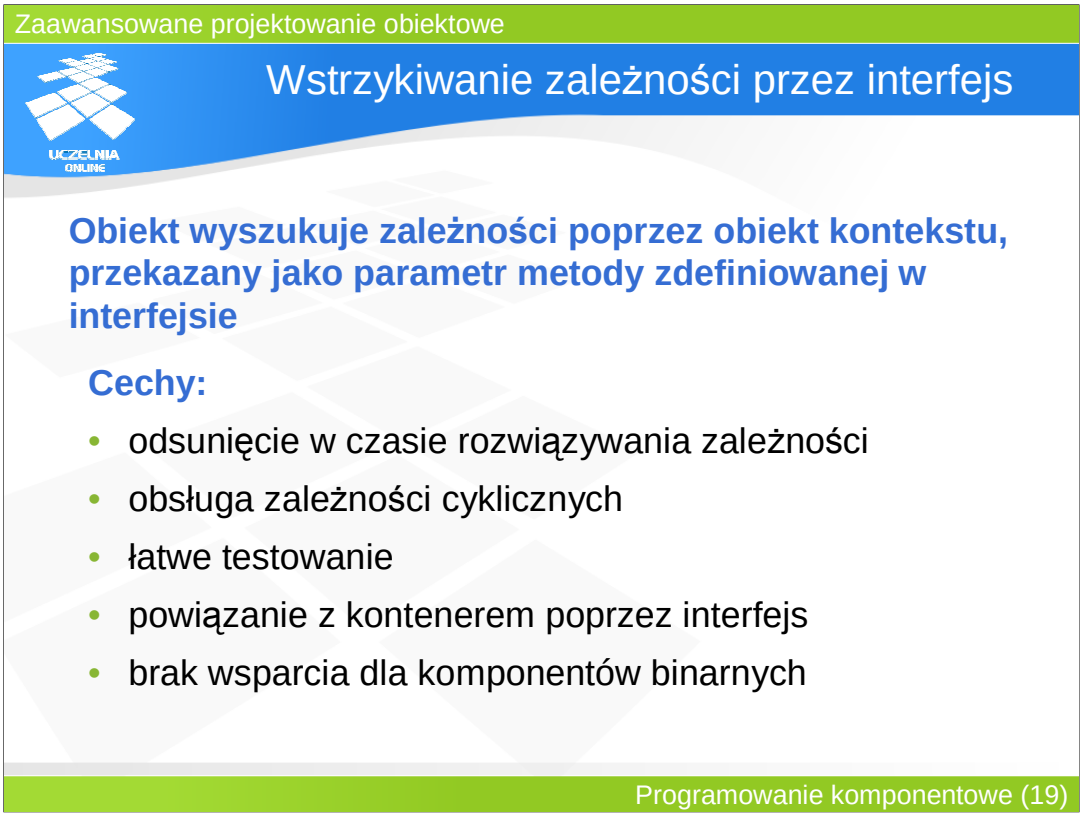


W tej części wykładu omówione zostaną metody wyszukiwania zależności pomiędzy komponentami, jakie oferują kontenery IoC. W opozycji do tradycyjnej metody, polegającej na aktywnym wyszukiwaniu instancji komponentów w rejestrze kontenera, przedstawiona zostanie grupa rozwiązań znanych pod nazwą wstrzykiwania zależności (ang. *dependency injection*).

Metody rozwiązywania zależności można podzielić ze względu na sposób ich przekazania do komponentu na trzy grupy:

- wstrzykiwanie przez interfejs, polegające na aktywnym wyszukaniu przez komponent wymaganych zależności,
- wstrzykiwanie przez konstruktor, w którym wszystkie zależności komponentu muszą zostać spełnione w momencie jego tworzenia, a ich rozwiązaniem i spełnieniem zajmuje się kontener,
- wstrzykiwanie przez właściwości obiektu, czyli przekazanie przez kontener referencji do zależnego komponentu przez metody typu *setXXX()*.

Zastosowanie każdej z tych grup niesie za sobą określone konsekwencje i ograniczenia, które pokrótce zostaną omówione.

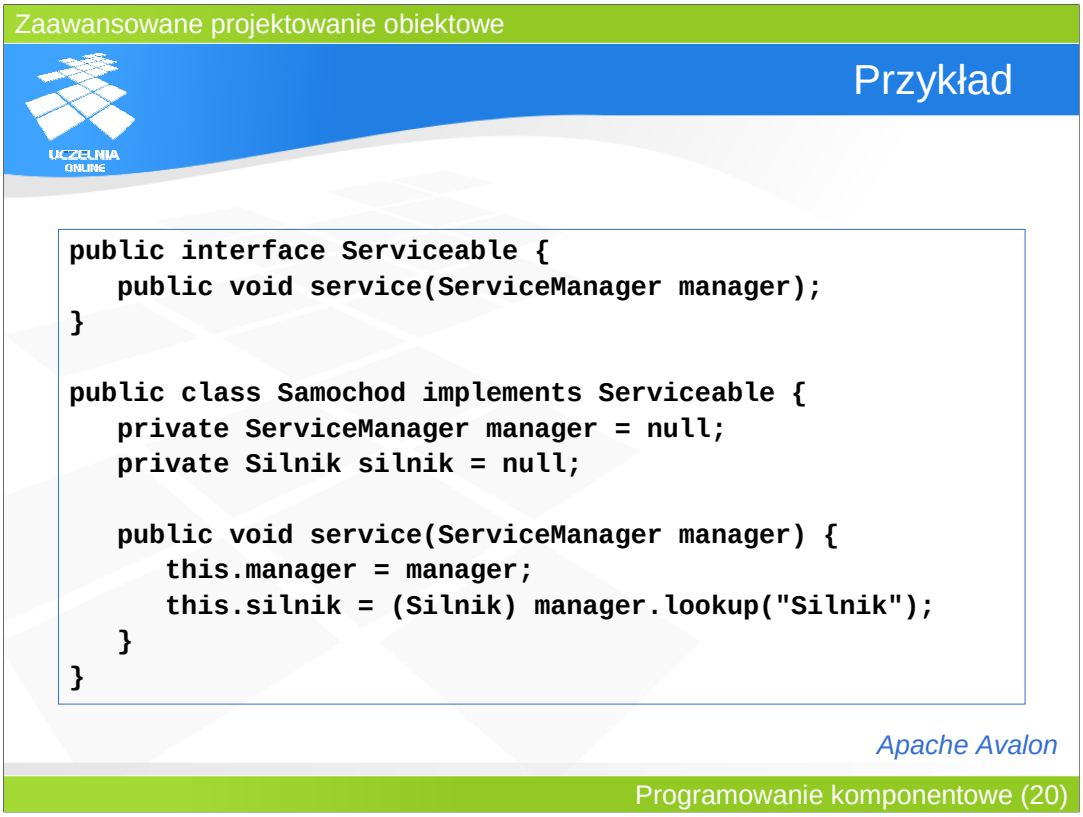


Wstrzykiwanie zależności poprzez interfejs w zasadzie nie polega na wstrzykiwaniu (co sugeruje pasywną rolę komponentu), a na aktywnym wyszukiwaniu komponentów w rejestrze kontenera. Jest to ciągle najpopularniejsza metoda rozwiązywania zależności, stosowana powszechnie m.in. w technologii J2EE. Przykładami jej zastosowań w praktyce są mechanizmy JNDI znany z języka Java oraz Naming Service stosowana w specyfikacji CORBA.

Metoda ta polega na implementacji w komponencie specjalnego interfejsu, stanowiącego znacznik wskazujący, że komponent ten wymaga pewnych zależności. Interfejs definiuje metodę posiadającą parametr, który jest obiektem kontekstu – wyszukiwarką innych obiektów. Komponent w momencie utworzenia otrzymuje od kontenera instancję kontekstu, któą może zapamiętać, a następnie wykorzystać do rozwiązania swoich zależności.

Poważną wadą tego rozwiązania jest silne powiązanie z konkretnym kontenerem, jakie wynika z konieczności implementacji przez komponent specyficznego interfejsu. W niektórych sytuacjach – np. gdy dostępny jest jedynie komponent w wersji binarnej – użycie tego sposobu rozwiązywania zależności jest niemożliwe.

Dlatego technika ta, mimo swojej popularności, powoli traci obecnie znaczenie na rzecz innych, prostszych metod, przede wszystkim wstrzykiwania zależności



Przykładowa implementacja mechanizmu wstrzykiwania przez interfejs stosowana w kontenerze Apache Avalon Fortress została przedstawiona na slajdzie. Klasa Samochód deklaruje implementację interfejsu *org.apache.avalon.framework.service.Serviceable*. Interfejs ten definiuje metodę *service()*, przyjmującą parametr typu ServiceManager. Parametr, pełniący rolę obiektu-kontekstu, ten zostanie przekazany komponentowi przez kontener w momencie jego tworzenia poprzez wywołanie metody *service().*

Obiekt ServiceManager jest wyszukiwarką wyszukującą instancje komponentów w rejestrze kontenera. W tym celu oferuje metodę *lookup()*, która korzystając z przekazanego jej klucza komponentu wyszukuje go w rejestrze kontenera. Dzięki temu komponent Samochód może uzyskać referencję do zależnego komponentu Silnik.

Aktywna rola komponentu w wyszukaniu zależności polega na całkowitym uzależnieniu tego procesu od niego. Komponent, który nie wywoła metody *lookup()*, nie otrzyma wymaganej zależności.

Wstrzykiwanie zależności przez konstruktor

Obiekt otrzymuje wszystkie zależności w postaci parametrów konstruktora.

Cechy:

- silna zależność pomiędzy komponentem i zależnościami
- rozwiązywanie zależności jest niepodzielne
- łatwe testowanie
- stan spójny bezpośrednio po utworzeniu (są *dobrymi obywatelami*)

Kolejne dwa sposoby rozwiązywania zależności stanowią istotną zmianę w stosunku do poprzedniej metody. Wyszukiwanie zależności nakładało na komponent obowiązek samodzielnego spełnienia swoich zależności, natomiast rola kontenera sprowadzała się do utrzymywania rejestru komponentów i dostarczenia obiektu kontekstu komponentowi implementującemu interfejs *Serviceable*.

Wstrzykiwanie zależności, bez względu na użytą technikę, zmienia ten stan rzeczy. Komponent, zamiast samodzielnie zdobywać potrzebne zasoby, jedynie deklaruje potrzebę ich wykorzystania w sposób, który jest czytelny dla kontenera i pozwala mu rozwiązać zależności w imieniu tego komponentu. Odpowiedzialność za dostarczenie zależności jest przeniesiona na kontener.

W przypadku wstrzykiwania przez konstruktor deklaracją taką są listy parametrów konstruktorów. Aby utworzyć instancję komponentu, kontener musi dostarczyć obiekty, które mogą posłużyć jako implementacje tych parametrów, i przekazać je do konstruktora tego komponentu. Jeżeli korzystając z zarejestrowanych komponentów nie może tego uczynić, wówczas utworzenie komponentu zależnego także nie jest możliwe i jest zgłaszane w postaci wyjątku. W przypadku gdy komponent posiada wiele konstruktorów, z zasady kontenery próbują utworzyć go korzystając z konstruktora najbardziej specyficznego (tj. o najdłuższej liczbie parametrów), którego zależności mogą rozwiązać.

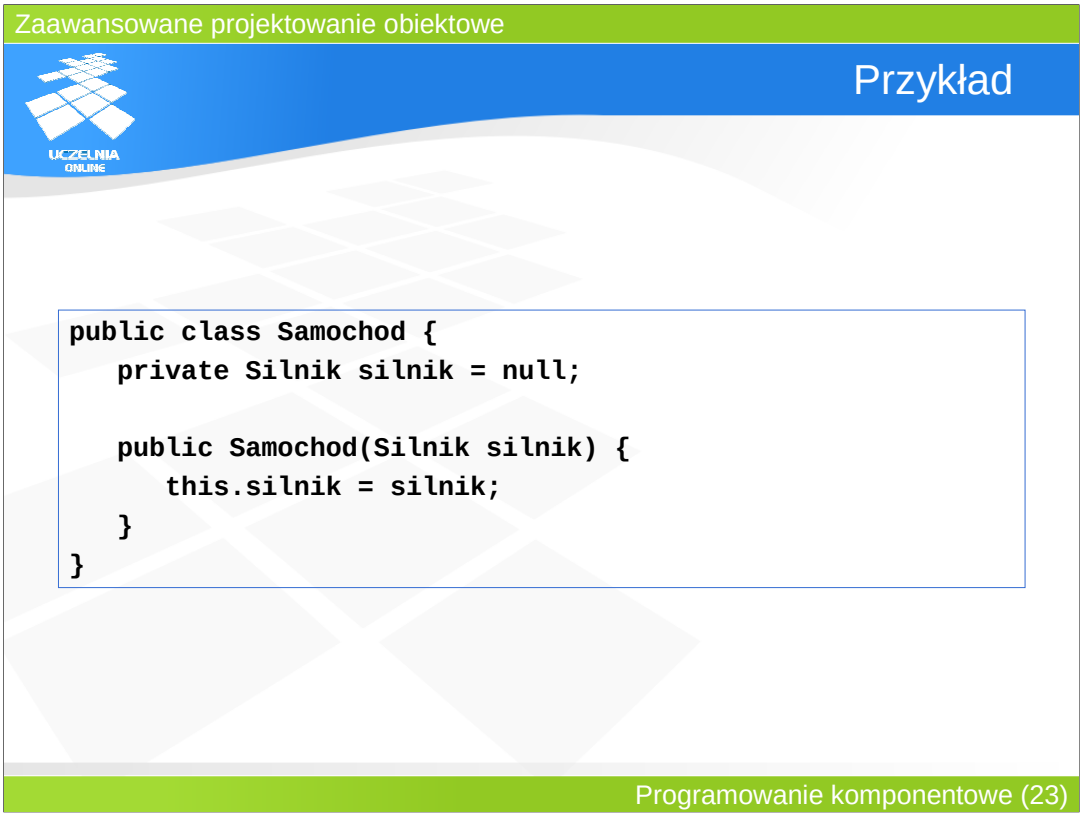
Wstrzykiwanie zależności przez konstruktor

Obiekt otrzymuje wszystkie zależności w postaci parametrów konstruktora.

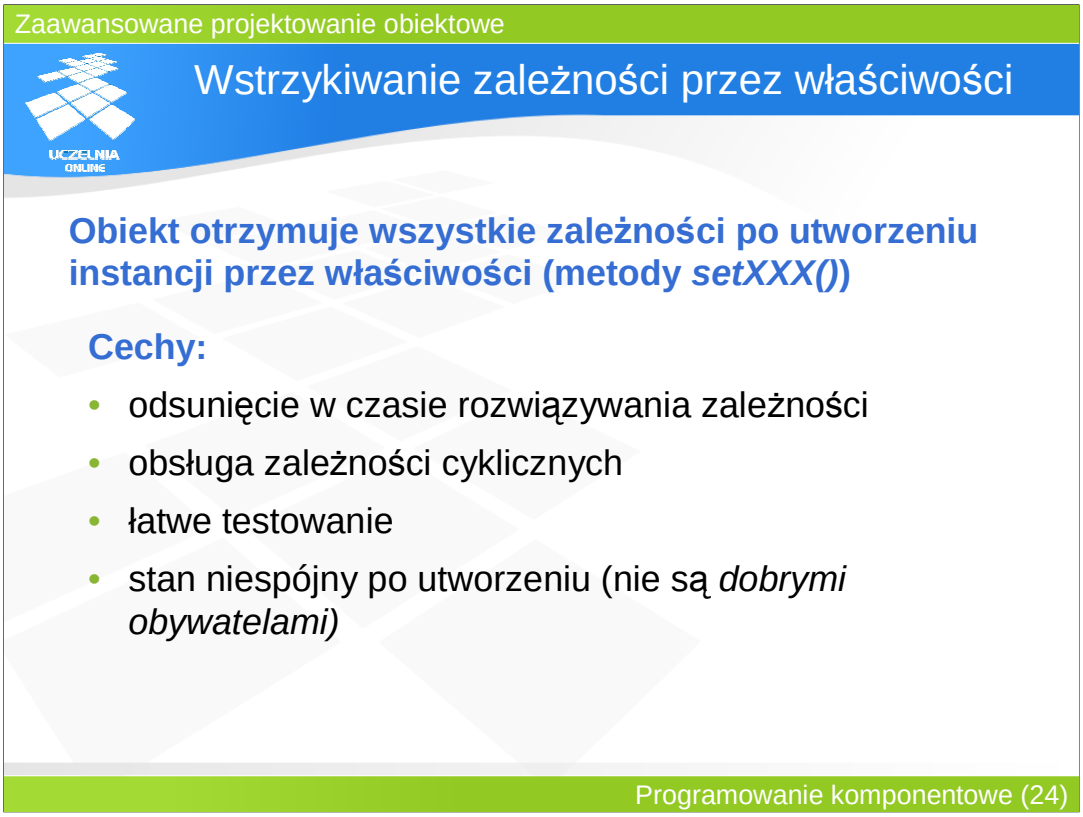
Cechy:

- silna zależność pomiędzy komponentem i zależnościami
- rozwiązywanie zależności jest niepodzielne
- łatwe testowanie
- stan spójny bezpośrednio po utworzeniu (są *dobrymi obywatelami*)

Najważniejszą cechą tej techniki jest zapewnienie spójnego stanu komponentu w każdym momencie jego istnienia (co jest jednym z warunków tzw. *dobrego obywatelstwa*), także bezpośrednio po wywołaniu konstruktora przez komponent. Ceną za uzyskanie tej własności jest brak możliwości zrealizowania zależności cyklicznych, tj. takich, w których komponenty zależą od siebie wzajemnie.



W kodzie programu mechanizm ten wymaga jedynie stworzenia w klasie komponentu Samochód konstruktora przyjmującego obiekt o interfejsie Silnik jako parametr. Kontener, tworząc instancję klasy Samochód, musi najpierw utworzyć obiekt typu Silnik, żeby przekazać go do właściwego konstruktora.



Trzecia metoda rozwiązywania zależności jest obecnie najbardziej popularna. Podobnie jak w przypadku wstrzykiwania przez konstruktor, tutaj także wszystkie zależności są przez komponent jedynie deklarowane, a ich spełnieniem zajmuje się kontener. Jednak tym razem do przekazania instancji obiektów zależnych nie służy konstruktor (który w tym przypadku zwykle jest bezparametrowy i służy niemal jedynie do fizycznej alokacji pamięci dla obiektu), a metody typu *setXXXX()*, znane z technologii Java Beans, które przyjmują zależności jako parametry.

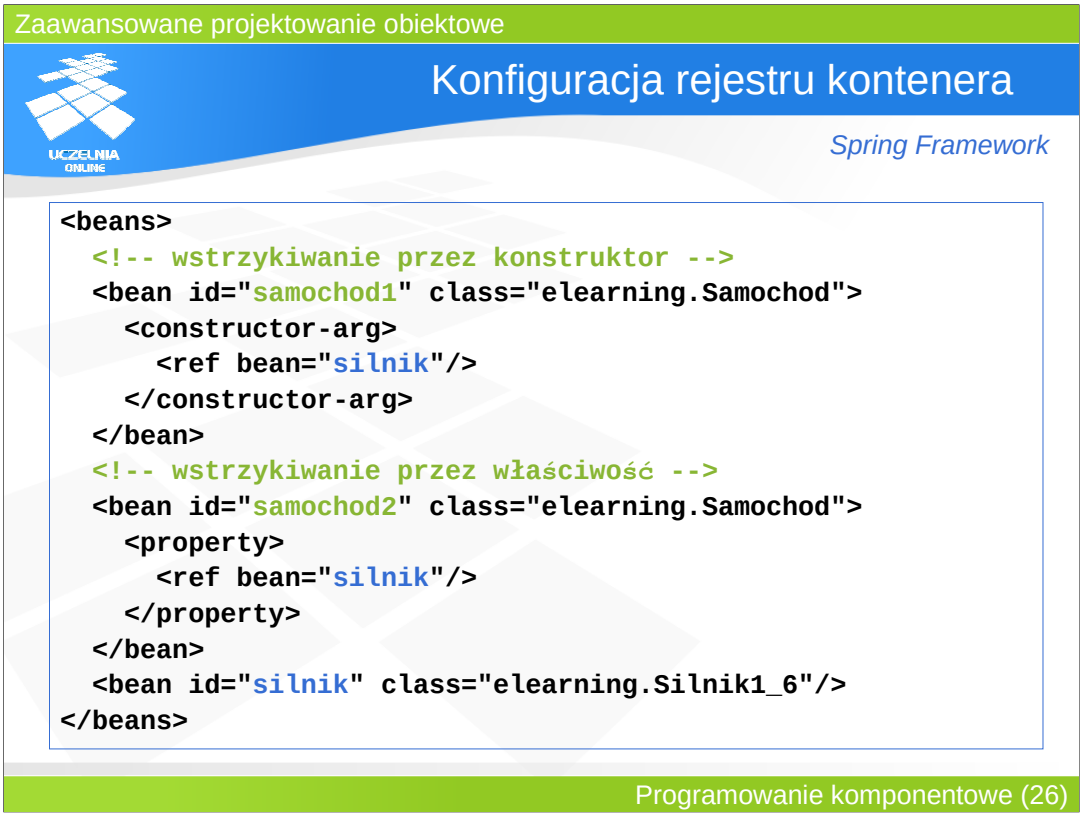
Wśród najważniejszych cech tego sposobu warto wymienić możliwość odsunięcia w czasie momentu rozwiązywania zależności, dzięki czemu komponent może zostać w pełni skonfigurowany dopiero w momencie pierwszego użycia. Wiąże się z tym możliwość obsługi zależności cyklicznych, co było niemożliwe w przypadku wstrzykiwania przez konstruktor.

Drugą konsekwencją stosowania tej metody jest możliwość utworzenia komponentów niespójnych, które wymagają odrębnej inicjacji. Wprawdzie proces ten zwykle jest realizowany przez kontener, który zapewnia poprawność komponentu w momencie jego wykorzystania, jednak w szczególnych okolicznościach może to spowodować nieprawidłowe zachowanie programu.

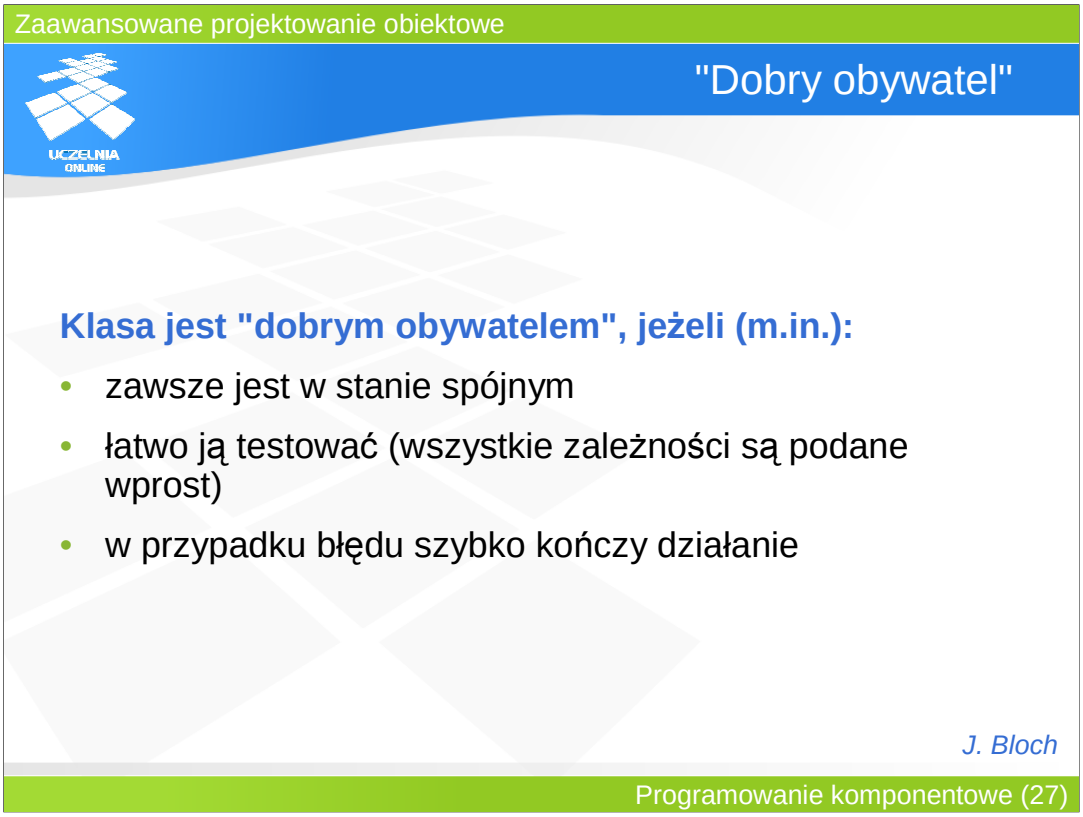
Przykład

```
public class Samochod {
   private Silnik silnik = null;

   public Samochod() {
   }

   public void setSilnik(Silnik silnik) {
      this.silnik = silnik;
   }
}
```


Na slajdzie przedstawiono możliwą implementację komponentu Samochód, korzystającą z rozwiązywania zależności poprzez właściwości. Klasa Samochód posiada bezparametrowy konstruktor oraz metodę *setSilnik()*, która przyjmuje obiekt typu Silnik jako argument. Kontener wywoła tę metodę na komponencie, pozwalając mu zapamiętać referencję do przekazanego w postaci parametru obiektu.



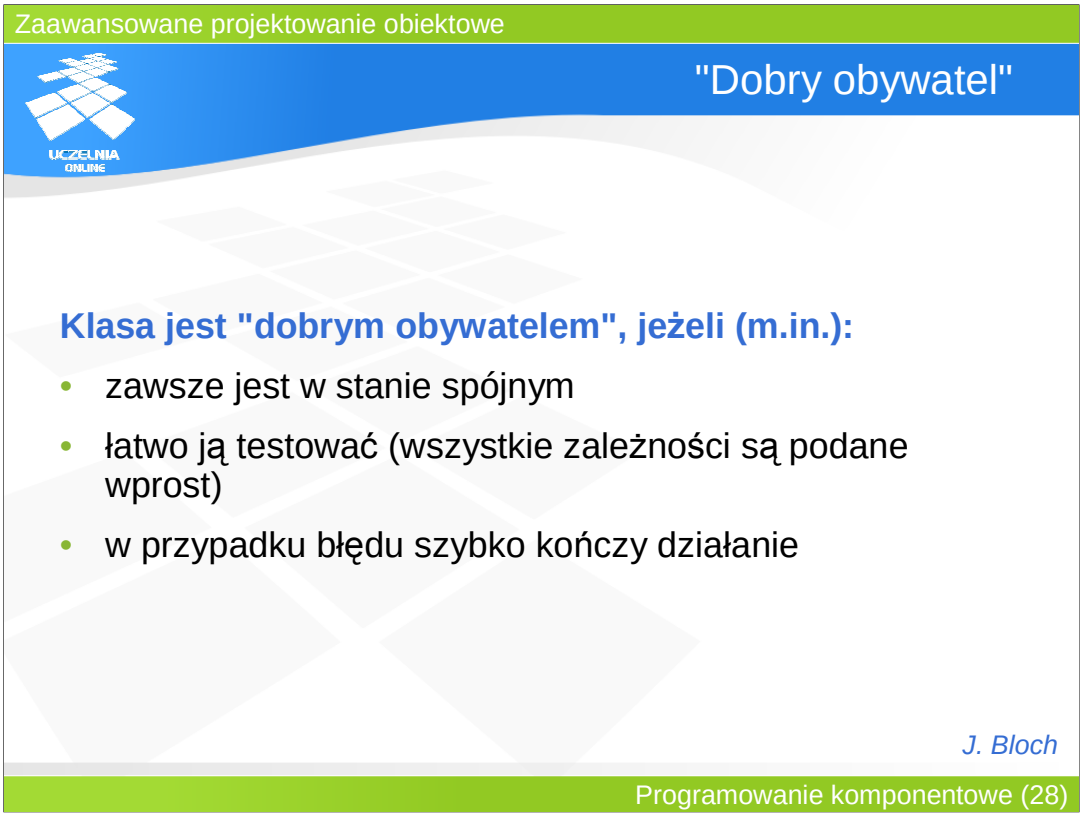
Dostęp do komponentów w rejestrze wymaga najpierw jego skonfigurowania. Na slajdzie przedstawiono przykładową konfigurację komponentów Samochód i Silnik dla kontenera Spring. Definiuje on dwa komponenty typu Samochód. Komponent dostępny pod nazwą *samochod1* wykorzystuje wstrzyknięcie komponentu Silnik przez konstruktor, natomiast pod nazwą *samochod2* znajduje się komponent korzystający z wstrzykiwania przez właściwość o nazwie *silnik*.



Korzystając z okazji, warto przypomnieć pojęcie *dobrego obywatela*, użyte przez J. Blocha (współarchitekta języka Java) w odniesieniu do klas spełniających pewne warunki. Klasa jest dobrym obywatelem, jeżeli postępuje w przewidywalny sposób, zatem korzystające z niej klienty mogą czynić pewne założenia dotyczące jej stanu.

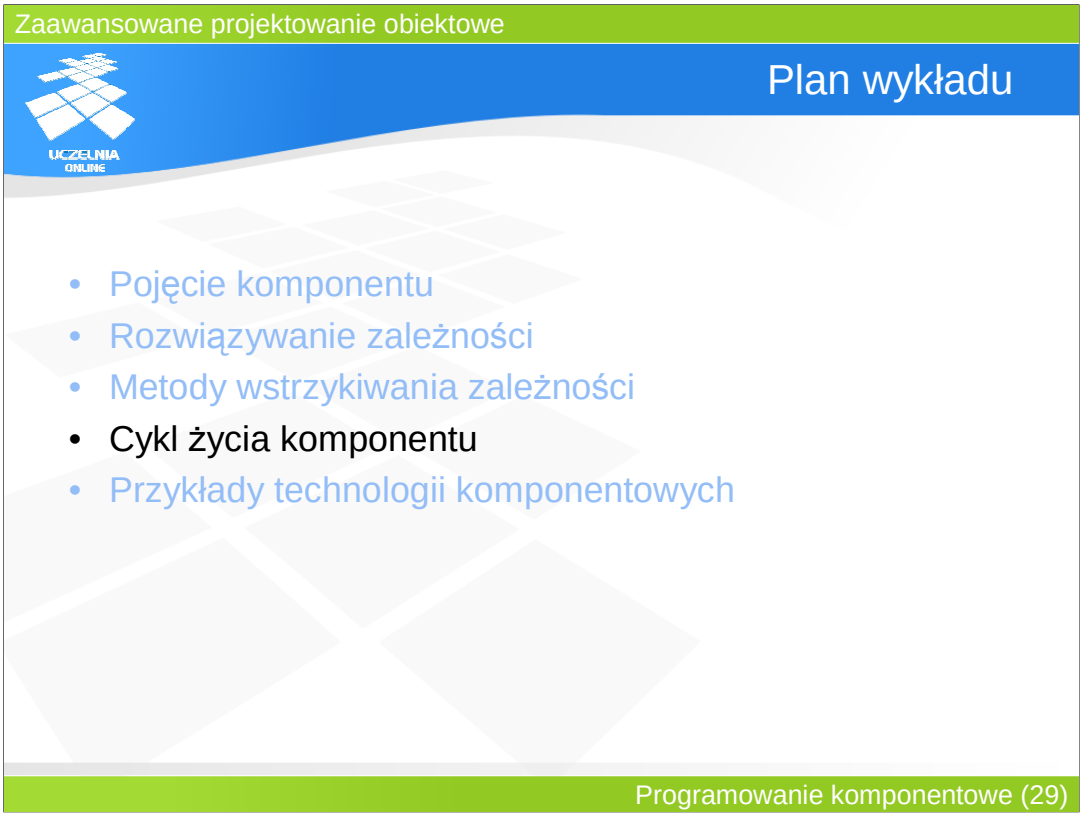
Wśród wielu atrybutów dobrego obywatelstwa, z zasadą odwróconego sterowania są związane trzy:

- **spójność stanu** oznacza, że konstruktor zawsze tworzy prawidłowy obiekt lub zgłasza wyjątek. Dodatkowe metody inicjujące, które muszą zostać wywołane po konstruktorze, są błędem projektowym;
- **testowalność**, oznaczająca, że klasa wszelkie zależności otrzymuje z zewnątrz. Dzięki temu można zależności zastąpić np. obiektami zastępczymi (ang. *mock objects;* zob. wykład nt. testowania jednostkowego);
- **szybkie zgłoszenie błędu**, które pozwala uniknąć niepotrzebnego zużycia zasobów na nieistotne w kontekście błędu operacje.



Warto zauważyć, że zasada dobrego obywatelstwa, choć odpowiada części warunków stawianych komponentom, jest naruszana przez wiele cieszących się dobrą opinią i dużą popularnością produktów i technologii komponentowych. Jest powszechną praktyką ograniczanie konstruktora obiektu wyłącznie do realizacji podstawowych operacji, które nie spowodują zgłoszenia wyjątku. Rolę "logicznego" konstruktora przejmuje w takim przypadku określona metoda, dzięki czemu można obsłużyć zgłaszane przez nią wyjątki bez konieczności tworzenia nowej instancji obiektu (co miałoby miejsce w przypadku zgłoszenia wyjątku przez konstruktor). Przykładem takiego rozwiązania jest technologia Java Servlets, w której cyklem życia serwletu zarządza kontener, i on fizycznie tworzy instancje tej klasy. Programista ma do dyspozycji jedynie metody związane z cyklem życia: *init(),* wykonywaną tuż po utworzeniu obiektu, oraz *destroy(),* związaną z jego finalizacją.

Dlatego stosowanie się do zasady dobrego obywatelstwa zależy od przyjętej filozofii projektowania i nie należy bezwzględnie jej przestrzegać.

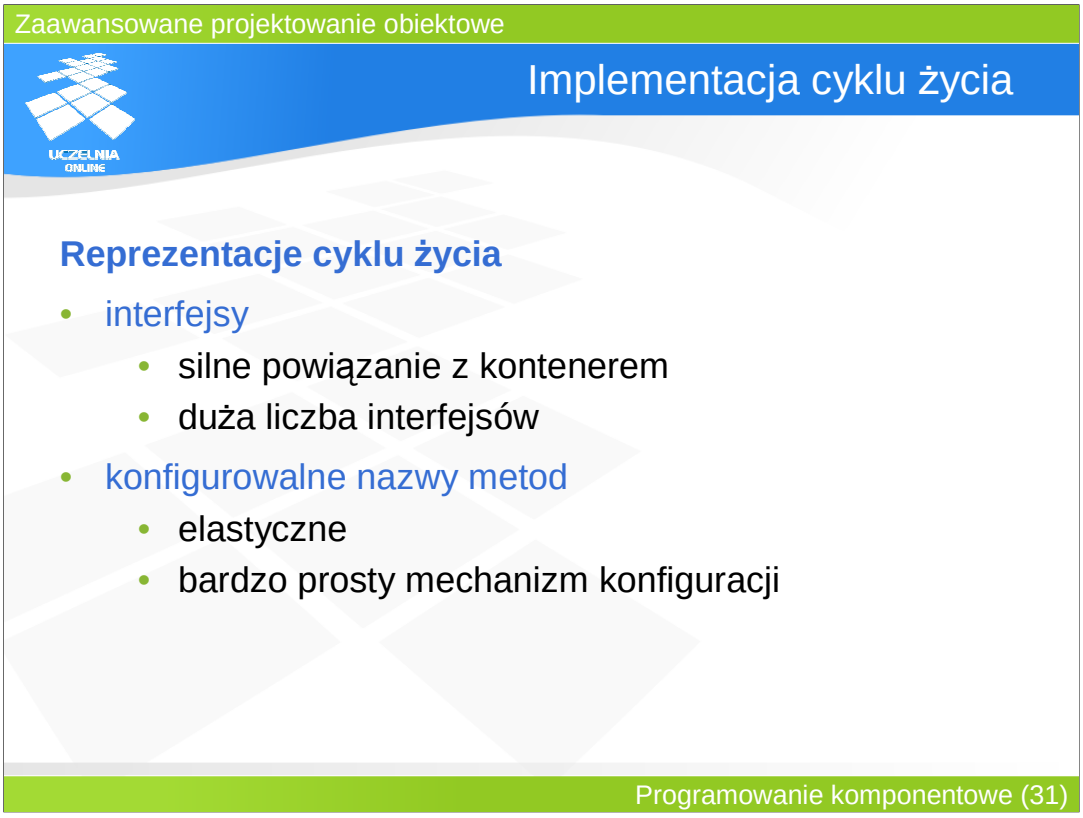


Kolejnym zagadnieniem związanym z komponentami jest ich cykl życia, który, podobnie jak inne aspekty istnienia komponentu, jest kontrolowany przez kontener.

Komponent zarządzany przez kontener jest od niego całkowicie zależny, jednak istnieje sposób pozwalający komponentowi na wykonanie pewnych działań w określonych etapach swojego życia. Służy do tego kontrakt pomiędzy komponentem deklarującym określone metody odpowiadające typ etapom, a kontenerem, który we właściwym momencie je wywoła. Dzięki temu komponent może mieć wpływ m.in. na sposób swojej inicjacji, konfiguracji i usuwania.

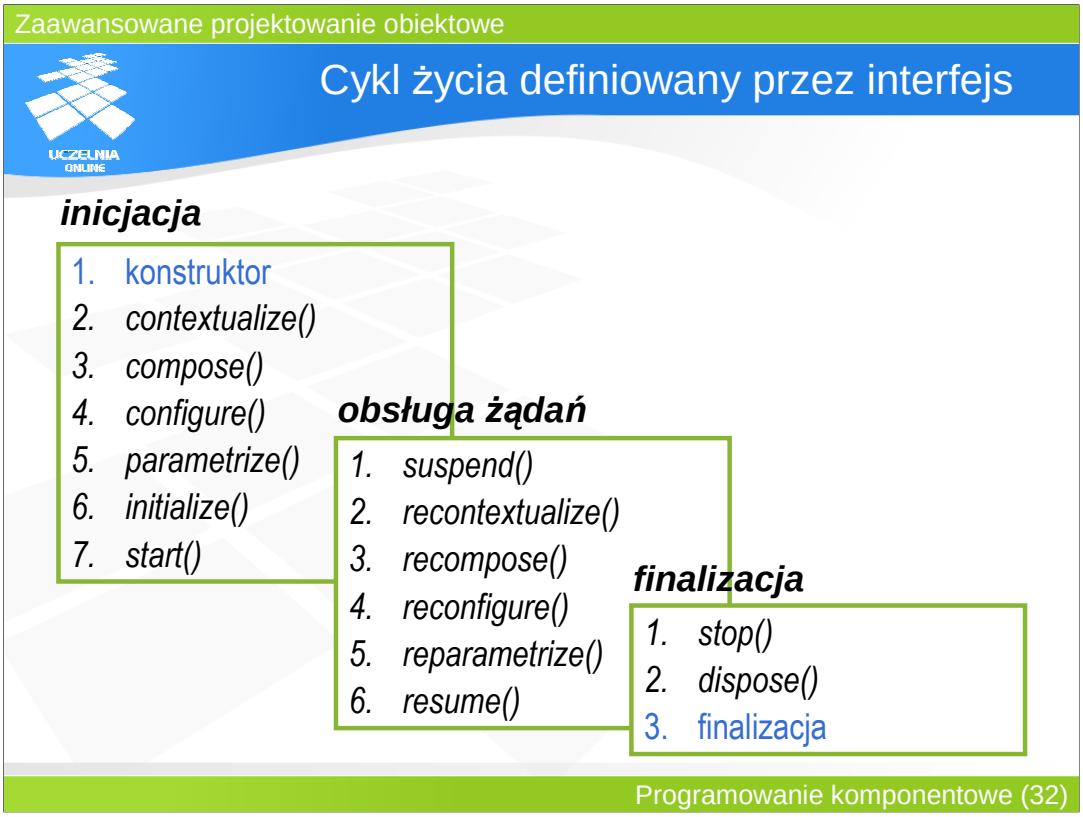
Drugim czynnikiem decydującym o cyklu życia komponentu jest kolejność wywoływania metod definiujących ten cykl. Jednak komponent ma wpływ jedynie na implementację metod, natomiast dokładnym określaniem momentu ich wywołania ich oraz samym wywoływaniem zajmuje się kontener.

Metody należące do cyklu życia, w zależności od etapu, który reprezentują, mogą być wywoływane jednokrotnie (np. inicjacja, konfiguracja) albo wielokrotnie (np. obsługa żądania, rekonnfiguracja).



W kontenerach IoC spotyka się dwie zasadnicze metody implementacji cyklów życia komponentów:

- poprzez specjalizowane interfejsy związane z danym kontenerem, które definiują metody reprezentujące poszczególne etapy życia komponentu, oraz
- poprzez bezpośrednie podanie nazw metod związanych z konkretną fazą życia komponentu podczas jego konfiguracji.



Na slajdzie tym przedstawiono przykładowe nazwy metod wchodzących w skład interfejsów definiujących cykle życia w kontenerze Apache Avalon Fortress. Liczba możliwych do wywołania metod jest bardzo duża, co daje programiście duży wybór i pozwala uwzględnić niemal każdy etap w cyklu życia komponentu, jednak z drugiej strony wiele z nich jest stosowanych jedynie incydentalnie. To, w powiązaniu z koniecznością implementowania wielu interfejsów (związanych z konkretnym kontenerem, a więc nieprzenoszalnych), przesądziło o niskiej popularności tego rozwiązania.

W praktyce obecnie jest ono stosowane w bibliotekach i szkieletach programowych, które narzucają ten schemat poprzez dziedziczenie (np. wspomniane już Java Servlets czy choćby JUnit, w których cykl życia komponentu składa się z trzech faz: inicjuj – działaj – finalizuj. W tym przypadku ma to jednak większy związek z usunięciem kodu inicjującego obiekt z konstruktora komponentu, dzięki czemu można taki obiekt rekonfigurować i stosować wielokrotnie.

Cykl życia definiowany przez nazwy metod

Cechy:

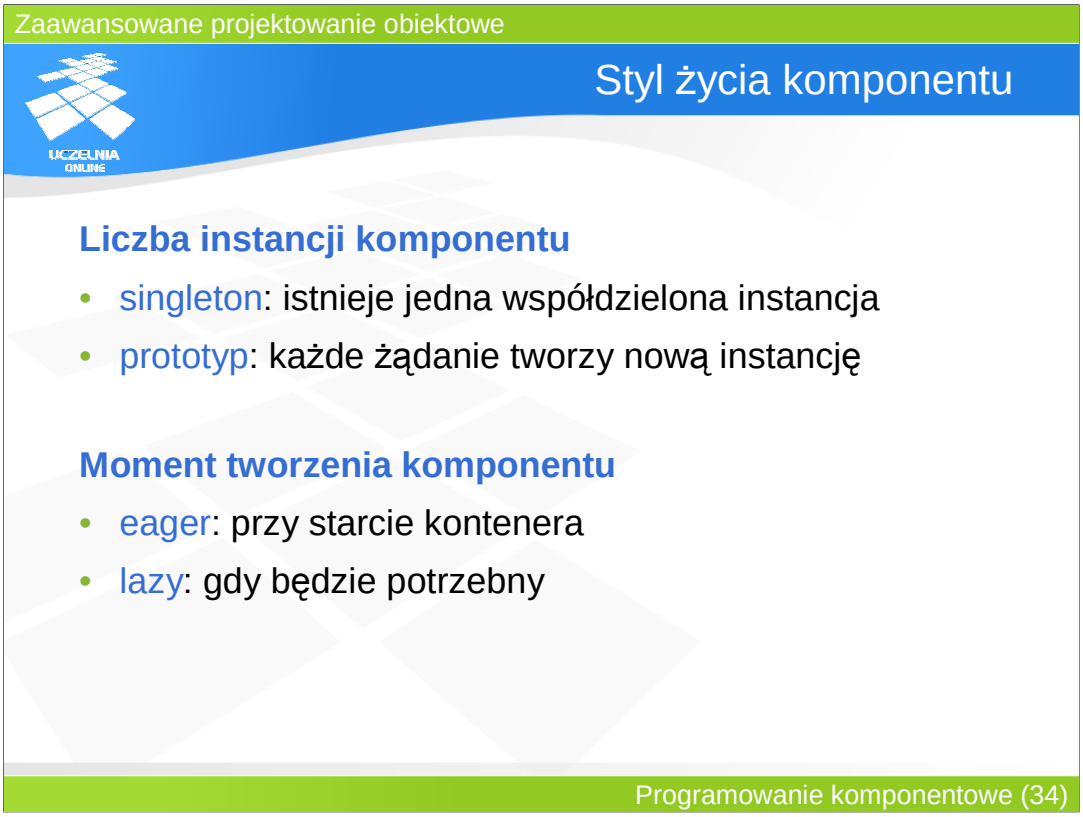
- wiązanie nazw metod z określonym punktem w życiu komponentu
- zwykle ograniczone do dwóch faz: inicjacji i usuwania

```
<bean id="Ksiazka" class="elearning.Ksiazka"
   init-method="inicjuj" destroy-method="usun">
      <property name="karta" />
</bean>
```


Drugie rozwiązanie jest stosowane wprawdzie tylko przez niektóre komponenty, jednak daje ono potencjalnie większe możliwości konfiguracji komponentu. Polega na określaniu nazw metod wywoływanych w typowych fazach życia komponentu i przekazaniu ich kontenerowi w postaci konfiguracji.

Metoda ta, mimo że nie wymaga implementacji interfejsów, dzięki czemu jest rozwiązaniem bardziej przenośnym, w przypadku wielu implementacji ma nieco ograniczone (choć należy przyznać, że w wielu przypadkach wystarczające) możliwości. Zwykle dotyczy jedynie dwóch faz: inicjacji i finalizacji komponentu, ponadto wymaga metod nie przyjmujących żadnych parametrów.

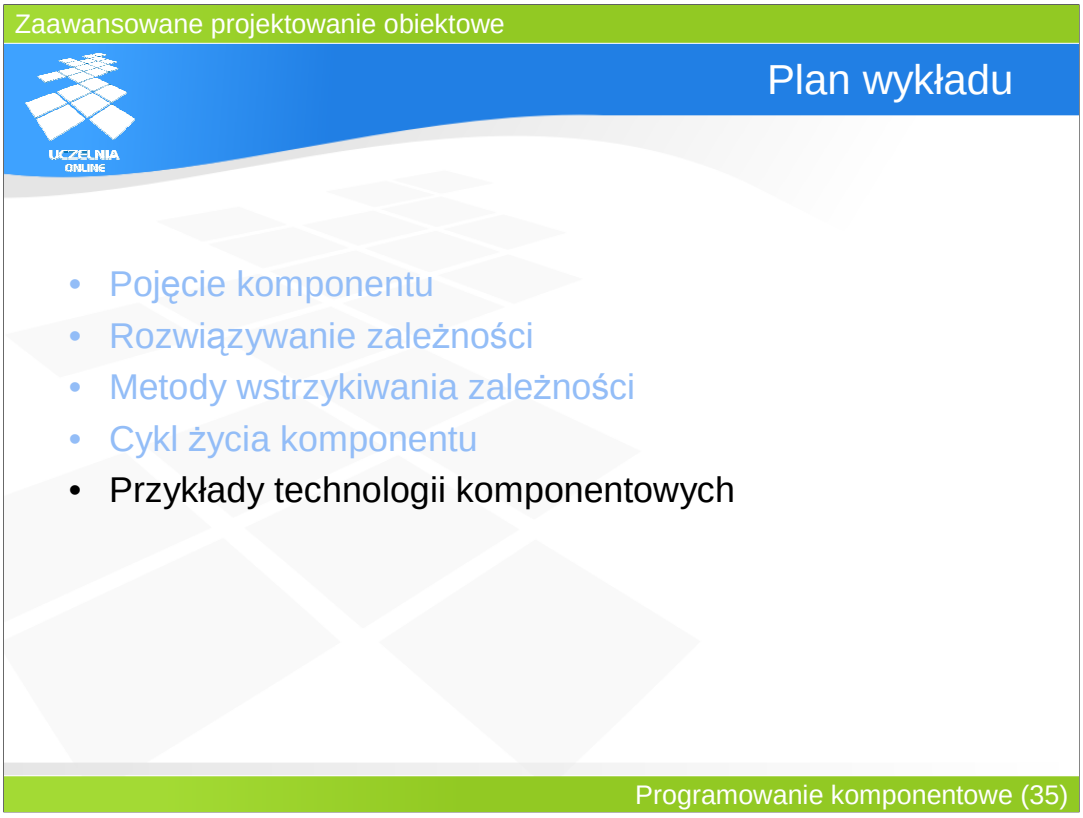
W przedstawionym fragmencie konfiguracji rejestru dla kontenera Spring komponent *Ksiazka* posiada metodę *inicjuj()* wywoływaną tuż po utworzeniu instancji obiektu oraz metodę *usun()* wykonywaną przed usunięciem instancji komponentu z rejestru.



Poza cyklem życia komponentu, opisującym graf jego stanów od momentu utworzenia do momentu usunięcia, komponenty mogą mieć swój styl życia. Pod tym pojęciem kryją się dodatkowe czynniki wpływające na sposób i moment tworzenia instancji komponentu.

Większość kontenerów pozwala na tworzenie komponentów-referencji (podobnych do obiektów-referencji), których jedyna instancja (singleton) jest współdzielona przez wszystkich żądających do niej dostępu klientów, oraz komponentów-wartości (również mających swój obiektowy odpowiednik), których instancje są tworzone w odpowiedzi na każde żądanie. Wybór stylu życia komponentu zależy utrzymywania przez niego stanu, współbieżnej obsługi żądań, żądanej wydajności etc.

Drugim ważnym czynnikiem określającym styl życia komponentu jest możliwość określenia momentu jego utworzenia. Komponenty są tworzone przez kontener w momencie uruchomienia, czyli zanim nastąpi potrzeba ich użycia (jest to tzw. chciwa inicjacja, ang. *eager initialization*), albo dopiero w momencie gdy są potrzebne (to tzw. leniwa inicjacja, ang. *lazy initialization*). Wybór jest – podobnie jak w poprzednim przypadku – podyktowany względami wydajnościowymi.



Ostatnia część wykładu obejmuje krótkie omówienie wybranych technologii komponentowych.

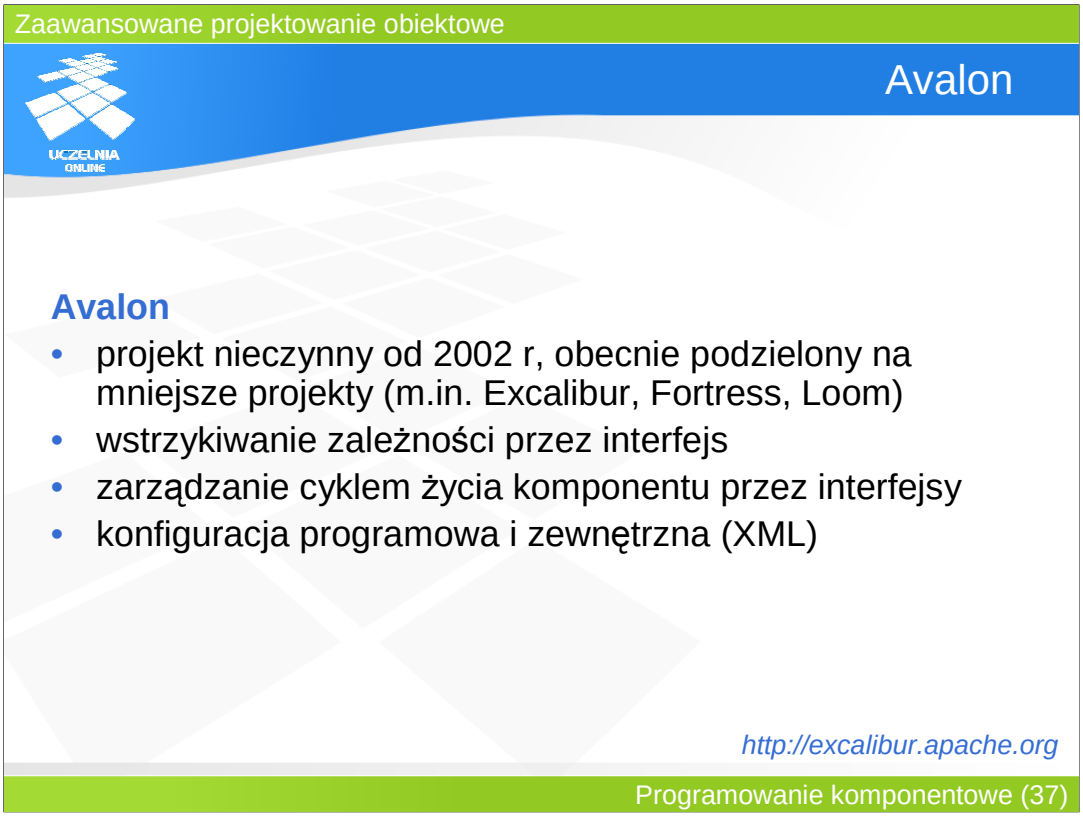
Lista kontrolna dla kontenerów IoC

Pożądane cechy kontenera IoC

- Sterowanie tworzeniem obiektów (bezpośrednio lub przez zdefiniowaną fabrykę)
- Szybkie i odwleczone inicjowanie obiektów
- Rozwiązywanie zależności
  - automatyczne, na podstawie typu zależności
  - na podstawie nazwy
  - poprzez konfigurację
- Wstrzykiwanie zależności
- Zarządzanie cyklem życia komponentu
- Obsługa zależności cyklicznych

M. Spille (2004)

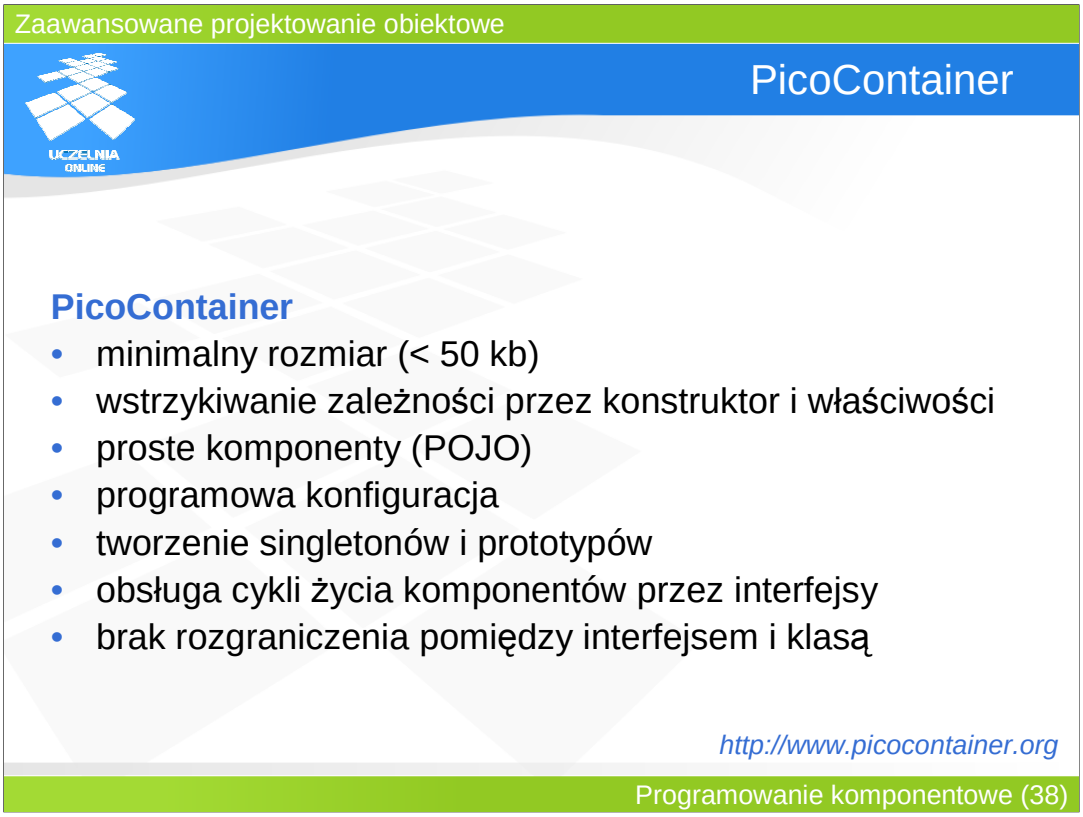
Slajd ten prezentuje najważniejsze elementy, które powinien posiadać kontener IoC. Wśród nich warto wymienić kontrolę nad tworzeniem obiektów z wykorzystaniem różnych technik, różne sposoby rozwiązywania zależności i ich wstrzykiwania do komponentu i przynajmniej proste zarządzanie cyklem życia komponentu.



Projekt Avalon rozwijany był do 1992 jako jeden z projektów Apache Software Foundation. Miał na celu stworzenie kontenera (lub serii kontenerów IoC) oraz zestawu wielokrotnie używalnych komponentów. Obecnie, wskutek zamknięcia projektu, jego dziedzictwo zostało przeniesione do niezależnych projektów tworzonych przez różne zespoły programistów.

Avalon dysponował tylko jedną metodą rozwiązywania zależności: wstrzykiwaniem poprzez interfejs. Umożliwiał także w ten sam sposób zarządzanie cyklem życia (definiował w tym celu dużą liczbę interfejsów, umożliwiającą obsługę wielu etapów w życiu obiektu). Rejestr kontenera był konfigurowany programowo (poprzez API) lub zewnętrznie, poprzez definicje komponentów zapisane w pliku XML.

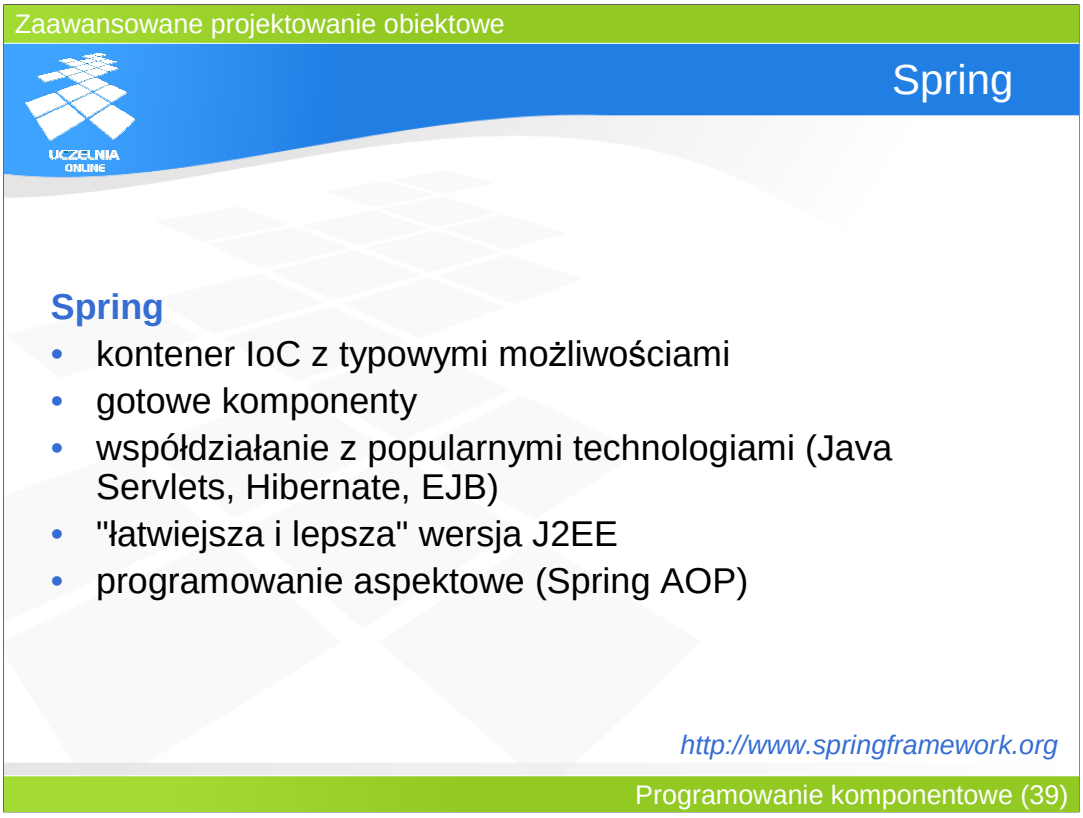
Avalon, z uwagi na ograniczone możliwości i niepraktyczny model rozwiązywania zależności, wiążący komponenty z konkretnym kontenerem, poza oprogramowaniem produkowanym przez ASF miał niewielkie znaczenie praktyczne. Obecnie warto o nim pamiętać właśnie ze względu na stosowany przez niego mechanizm wstrzykiwania przez interfejs.



PicoContainer jest – jak nazwa wskazuje – miniaturowym systemem zarządzania komponentami. Charakterystyczne jest, że nie posiada on żadnych zewnętrznych zależności (poza JDK 1.3), a objętość pliku JAR z jego kodem nie przekracza 50kb. Pico jest jedynie kontenerem, i nie udostępnia żadnych innych mechanizmów, obecnych m.in. w Springu. Umożliwia wstrzykiwanie zależności przez konstruktor i właściwości, co w większości przypadków jest wystarczającym rozwiązaniem. Z uwagi na niewielki rozmiar nie posiada możliwości konfiguracji zewnętrznej, dlatego wymaga programowgo zgłoszenia komponentów do rejestru. Wspiera podstawowy cykl życia komponentu (składający się metod *start()*/*stop()* oraz *dispose()*, implementowanych przez interfejsy), potrafi także tworzyć komponenty jako obiekty-referencje (singletony), wartości (prototypy) oraz hierarchie obiektów.

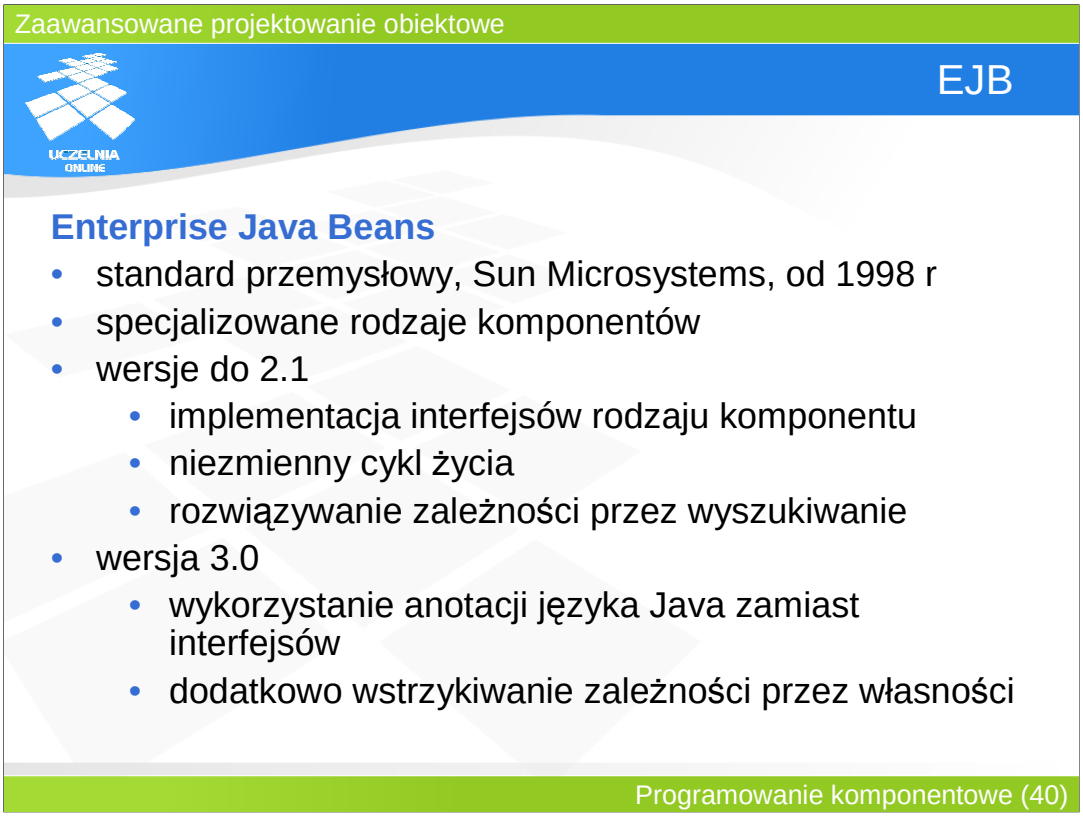
Wadą tego kontenera jest brak rozróżnienia pomiędzy interfejsem a klasą, co może prowadzić do niechlujnego programowania bez użycia abstrakcyjnych interfejsów.

Projekt jest interesujący z uwagi na stosunek możliwości do rozmiaru, jest jednak niszowym elementem sceny kontenerów IoC.



Dominującą rolę w świecie IoC odgrywa obecnie projekt Spring, reklamowany jako lekka (czytaj: prawidłowa) wersja J2EE. Spring to nie tylko kontener IoC, ale przede wszystkim duża liczba gotowych komponentów, obejmujących najważniejsze technologie. Dzięki temu budowanie aplikacji w Springu wymaga stosunkowo niewiele wysiłku, ponieważ pozwala na wykorzystanie istniejących i przetestowanych rozwiązań. Z tego powodu w opinii wielu osób jest łatwiejszą i lepszą wersją J2EE (biorąc pod uwagę kierunek ewolucji standardu EJB w wersji 3.0), trudno nie odmówić temu hasłu słuszności.

Poza standardowymi cechami większości kontenerów Spring posiada mechanizm tworzenia i użycia aspektów.



Mechanizm EJB jest promowaną przez Sun Microsystems technologią komponentową dla rozproszonych aplikacji klasy enterprise tworzonych w języku Java. Nie jest produktem, a jedynie specyfikacją, której implementacja zależy od twórców oprogramowania.

EJB specyfikuje obecnie trzy rodzaje komponentów, reprezentujących encje (jednostki danych), sesje (przetwarzanie danych) oraz obsługujące komunikaty. Podobnie jak w przypadku innych technologii, komponenty są tworzone i zarządzane przez kontener, który przejmuje także większość zadań związanych z zapewnianiem dostępu do danych, transakcyjnością etc.

Początkowy rozwój tej specyfikacji powodował gwałtowny wzrost jej złożoności, co odbijało się negatywnie na łatwości stosowania tej technologii. Pojawienie się lekkich kontenerów IoC, które powstały w odpowiedzi na problem związane z EJB, miał wpływ na dalsze losy także tej technologii. W wersji 3.0 wprowadzono wstrzykiwanie zależności oraz wykorzystano anotacje języka Java do opisu przeznaczenia i konfiguracji komponentu. Pozwoliło to uprościć model komponentowy i ułatwić jego stosowanie w praktyce.

Podsumowanie

- Komponent jest konfigurowalną jednostką programową wielokrotnego użytku
- Odwrócenie sterowania pozwala ograniczyć powiązania pomiędzy komponentami
- Kontenery IoC wspierają różne metody tworzenia komponentów, rozwiązywania ich zależności i zarządzania ich cyklem życia

Podczas wykładu przedstawiono wstęp do programowania komponentowego. Określono jego rolę w porównaniu z programowaniem obiektowym, zdefiniowano pojęcie komponentu jako konfigurowalnej jednostki programowej, oraz omówiono problematykę rozwiązywania zależności. Szczególną uwagę skupiono na bardzo popularnej ostatnio metodzie rozwiązywania zależności przez ich wstrzykiwanie, omawiając jej zalety w stosunku do tradycyjnego podejścia, opartego na wyszukiwaniu komponentów w rejestrze. Zaprezentowano typowe cechy kontenerów IoC, ich możliwości i braki. W ostatniej części przedstawiono kilka wybranych technologii komponentowych obecnych na rynku oprogramowania.